Computational Stylistics

Jack Doyle

April 2015

B.A. (Mod.) Computer Science, Linguistics and a Language
Final Year Project
Supervisor: Dr. Carl Vogel
Second Reader: Dr. Eamonn O’Nuallain
Acknowledgements

I would like to thank Dr. Carl Vogel for his support throughout the project, and indeed throughout the last four years I have spent at Trinity College.

Special thanks are due to my classmates, with whom some very special times were had.

Lastly, a very special thank you goes to my parents, for their emotional and financial support throughout my college years.
Contents

1 Introduction 5
   1.1 Why a web app? .................................................. 5
   1.2 Aims ................................................................. 6
   1.3 Methods ............................................................ 6

2 Background Material 6
   2.1 Computational Stylometry ....................................... 6
   2.2 Authorship Attribution
      2.2.1 Feature Selection ........................................... 7
      2.2.2 Existing Tools ............................................... 10
   2.3 Document Similarity
      2.3.1 Vector Space Model ....................................... 11
      2.3.2 N-grams .................................................... 12
      2.3.3 Cosine Similarity ......................................... 12
      2.3.4 TF-Idf Weighting ......................................... 13
   2.3.5 Chi-Square Similarity ..................................... 15

3 Application Structure 16
   3.1 High-Level Overview ......................................... 16
   3.2 Text Processing
      3.2.1 Tokenization ............................................... 17
      3.2.2 Vectorization and Weighting ............................... 18
   3.3 Arithmetic
      3.3.1 Cosine Similarity ......................................... 19
      3.3.2 Chi Square Similarity .................................. 20
   3.4 The Comparator Module ...................................... 20
   3.5 The Flask Framework
      3.5.1 Overview .................................................. 21
      3.5.2 The stylo Module ......................................... 21
   3.6 Program Flow .................................................. 22
   3.7 Corpus ................................................................ 23
      3.7.1 Database .................................................... 23

4 Future Work 23
   4.1 Extending the Project ......................................... 23
   4.2 Other Applications ............................................. 24

5 Conclusion 25
1 Introduction

“Which author do you write like?” is a question whose point, on the surface, perhaps seems a little trite – especially in the context of an academic project – and one that might sound better rephrased: “How similar are certain features of one’s writing style to those of a given author?” In essence this question is one which pertains to the field of computational stylistics and stylometry, the study and measure of features of writing style using computational methods. This project is an attempt to answer that question using methods in computational stylistics, through the medium of a web application.

1.1 Why a web app?

In the last decade or so, the Internet has started to evolve from what was arguably a collection of static websites to a more user interaction-based environment. The emergence of dynamic content meant that websites could provide an interactive interface for the user, and allow the user to interact with the technology at a closer level. The relation between Internet users and websites has moved from ‘browsing’ to ‘using’.

This change in web interaction is exciting because the user is now able to engage directly with the functioning of a website or web application, be it a social media platform, a cloud storage app, or simply an email app. The user is no longer separated from the underlying layer of technology of a website; with the advent of interactivity on the web, that curtain has been partially lifted. What results is a raise in the user’s awareness to the underlying operation of an app during interaction, even if that interaction is as simple as a change of theme.

It is this interactivity which may enable an average user to recreationally use, and hopefully consider the workings of, a tool which they previously would not have come across, or considered employing the use of. The app proposed in this project is fundamentally a language processing tool. It is not unreasonable to assume that a scientific tool – especially one concerned with the analysis of data – will rarely be used by a person outside of its respective field. Thus, implementing it as a web app with flexible features appeals to a wider user base, including the average Internet user described above.

In other words, implementing this tool as a web app may reach more people who are not familiar with stylistics or indeed computational linguistics, and these people will hopefully take a new interest in one or both of these. Additionally, there is a sort of ‘fun’ side to the app, arising from the nature of the question “Which author do you write like?” which appeals to the curiosity of anyone who has ever asked themselves that question.
1.2 Aims

The principal aim of this project is to provide an interactive tool which employs established methods to compute similarity between texts, whose back-end is modular and easily extensible for use in other projects. Preliminary research did not uncover any existing app of this nature, hence a secondary aim is to bring Internet users into contact with stylistics, and by extent, the field of computational linguistics.

The eventual aim of this project, however, is to provide an all-encompassing stylistics comparison tool with more features, which draws upon existing research in authorship attribution studies. Eventually this will hopefully produce a tool with all the interactivity and novelty of a web app, but which will also be supported by current research in order to guarantee accuracy and credibility.

1.3 Methods

The app essentially receives a text as user input, and compares it to each of a set of documents by various authors stored in a database, obtaining for each document a similarity score. The similarity is calculated using chi-square measures, and cosine similarity. The documents are pre-processed and tokenized into word n-grams or character n-grams depending on the user’s choice, and then represented as vectors, making use of the Vector Space Model (VSM). The program returns to the user the name of the author whose text received the highest similarity score between itself and the user’s input.

2 Background Material

2.1 Computational Stylometry

As the development of the project is essentially a task in stylometry, a good starting point in the discussion of background material would probably be a brief overview of stylometry, including its origin and applications.

Stylometry dates as far back as the 19th century, when Thomas Mendenhall investigated a remark made by logician Augustus DeMorgan in a letter to a friend. DeMorgan posed the question of whether one text “does not deal in longer words” than another text. Building upon this idea, Mendenhall (1887) proposes the characteristic curve, which is “a graphical representation of an arrangement of words according to their length and to the relative frequency of their occurrence.” He claims that if this spectrum persists in form across works by the same author, then that spectrum could be characteristic of that author.

In 1901 Mendenhall carried out a study investigating the word length frequency distributions in works by Shakespeare and Bacon, with the goal of discerning whether or not works attributed
to the former could have been written by the latter. It was this analysis of writing style through study of features present in a text which contributed to the fundamental concept of stylometry.

In essence, stylometry can be described as the inference of the characteristics of an author’s style based on the characteristics of a text by that author. It revolves around the hypothesis that every author has a stylistic ‘fingerprint’, which cannot be consciously regulated, but which is comprised of a number of characteristics which are quantifiable and distinct. The task of the stylometrist is therefore to establish a countable feature, or set thereof, in the text which is conducive to determining the author’s style. The advent of technology means that counts of features in a text no longer have to be done manually; tools such as concordancers and parsers make much of the data readily available.

2.2 Authorship Attribution

The concept of authorship attribution builds upon the hypothesis referenced above; that every author’s style can be described by a set of quantifiable stylistic features which is characteristic to them. Also known as authorship verification or authorship identification, authorship attribution seeks to infer or verify the author of a text based on these stylistic features, using methods in computational stylometry.

Although in the scope of this project, the proposed app does not classify texts as having been written by such and such an author, its functioning does draw upon concepts from authorship attribution, so a brief overview of the field would be useful.

The set of methodological models and frameworks used in authorship attribution is vast and diverse (Juola, 2008). As such, authorship attribution is still a problem for which a well-defined solution does not exist. The dilemma lies in the choice of method to be used in the analysis of a text. Which features in the text are most reflective of an author’s style?

2.2.1 Feature Selection

An initial step in an authorship attribution problem involves defining a set of style markers to be used in the analysis. These are then counted in the text manually or with the help of a computational tool. Most modern authorship attribution methods are based on a measure of three types of features:

1. Lexical features, which are based on frequencies of occurrence of individual words, or which tend to represent the vocabulary richness of an author. (Stamatatos et al, 2001). These features include function words, and Yule’s K value for measuring vocabulary richness, among others.

2. Syntactic features such as parts-of-speech distribution (Baayen et al, 1996).
3. Low-level features, such as word-length (Mendenhall, 1887; Brinegar, 1963), syllables (Fucks, 1952) and sentence-length (Yule, 1938).

Over the last century of research in authorship attribution, the number of possible style markers has risen into the thousands. The sheer number of possible ways to extract meta-knowledge from text means that no single feature is likely to be universally ‘best’ in inferring an author’s style. Rather, an analysis of a combination of features tends to be more successful.

Early works employed a univariate approach to measuring style. As mentioned above, Mendenhall (1887) proposed word-length as a possibly distinguishing characteristic of an author. The findings of his analysis of works by Bacon, Shakespeare and Marlowe suggest that word-length is not a reliable indicator of style, and subsequent research has confirmed this (Smith, 1983).

Fucks (1952) measured syllables in a study on English and German authors. He calculated the average number of syllables per word, the relative frequencies of the i-syllabled words, and their distribution in the text (Holmes, 1994). However, this method proved to be more suited to discriminating between languages rather than authors (Fucks and Lauter, 1965).

Yule (1938) proposes sentence-length as a characteristic feature of an author, and uses that in his approach to the case of disputed authorship of The Imitation of Christ. He concludes, however, that sentence-length is not a very reliable indicator of one’s writing style; one disadvantage being that it is a variable which is under conscious control of the author. Smith (1983) backs up this conclusion – he asserts that sentence-length is not reliable as a stand-alone technique.

An early experiment in authorship attribution was carried out by Mosteller and Wallace (1964), in their study of the Federalist Papers. In the case study they use word counts as discriminating features. Since the topic of a text heavily influences which words are used in that text, a number of topic-independent words were chosen as features (a, and, to, upon . . . ). These function words which include prepositions, conjunctions and articles tend to be useful as a stylometric measure, as they are much more common than contextual words. Furthermore, one pays less attention to their own use of function words due to their pervasiveness in English, and would find it difficult to consciously alter how they use them. Burrows (1987) uses frequencies of sets of the 30-50 most common words in his sample, without making the distinction between context words and function words. This both minimizes the computational cost, and achieves good results for a wider variety of authors (Stamatatos et al, 2001).

One stylistic feature that has been put forward is a measure of vocabulary richness or diversity. The idea is that an author has a stock of words in his/her vocabulary, and that he/she will favour the use of some words over others, and that this may be reflected in a sampled text. If the function of all the vocabulary frequencies in the sample frequency profile produces a single measure, and if that measure is characteristic of the sample frequency distribution, then that measure could be used for comparative purposes (Holmes, 1994). One such vocabulary richness measure is the Type/Token ratio, which is the ratio of the number of word occurrences in the
text (N) to the number of lexical units which comprise the vocabulary in the sample (V). One problem associated with the Type/Token ratio is that while N can increase infinitely, V will likely be finite. Accordingly, it seems that the ratio may only be useful in analyses where N is fixed at a certain value. Yule (1944) proposes the K characteristic, a measure of vocabulary richness based on the assumption that the occurrence of a given word is based on chance, and that this can be modeled as a Poisson distribution (Holmes, 1994). However, Tallentire (1972) asserts that K is not suitable for authorship attribution problems.

Baayen et al (1996) remark upon the high efficacy of function words as feature types in works by Burrows (1992, 1993). As function words relate directly to syntax, Baayen et al consider the study of syntax directly in a text as a possible solution to authorship attribution. They use a syntactically annotated corpus to determine the discriminatory potential of syntactic rewrite rules. They conclude that analysis of syntactic rewrite rules yields higher discriminatory power than traditional word-based methods such as vocabulary richness and word occurrence frequencies. They add that through an additional analysis, it was observed that the use of rewrite rules is less variable in a text than the use of words themselves.

With so many feature types having been proposed for authorship attribution, a round-up and analysis of them is useful in any attempt to compare their discriminatory potential. Grieve (2007) provides a comprehensive evaluation of thirty-nine of the most commonly used textual measurements in authorship attribution. Each set of textual measurements is inserted into the same attribution algorithm and tested on the same data set to ensure a fair comparison. The measurements which were tested include word-length, sentence-length, vocabulary richness, grapheme frequency, word frequency, punctuation mark frequency, collocation (word n-gram) frequency and character n-gram frequency. Within these last two, collocation 2- and 3-grams are tested, and character 3- through 9-grams are tested. A brief explanation of n-grams is given in section 2.3.2. The results showed that character 2- and 3-gram frequency proved to be some of the most accurate techniques tested, distinguishing between two possible authors with 94% accuracy.

The features that were selected for use in the proposed app are character n-grams and word n-grams. While the lexical, syntactic and low-level features described above are certainly interesting in that they are more stylistic than character and word n-grams, they are, as has been mentioned, more prone to conscious manipulation by the author, while character n-grams are just a consequence of the words chosen by the author. Additionally, there have been encouraging results in the use of character n-grams as a style marker (Grieve, 2007; Vogel, 2007). Lastly, it has been suggested that character n-grams would be the most reliable style marker in a forensic analysis of a text using corpus linguistics if it is to satisfy the Daubert test of admissibility of expert testimony in criminal court (Vogel, 2007). Word n-grams are equally interesting as a style marker, as a chi-square analysis can reveal where exactly two documents are similar or dissimilar. Document similarity will be discussed more in section 2.3.
2.2.2 Existing Tools

A number of tools designed specifically for authorship attribution and by extent, stylometric analysis of texts, have been developed in the last few years. These tools have been developed with the average user in mind, allowing them to perform their own comparisons of texts. One of these tools is Signature, a program developed by Peter Millican of Oxford University. Version 2.00 of the tool is still in development, and it will have features for processing of text (tokenization, frequent word/phrase lists, etc.) as well as statistical analyses (Principal Component Analysis, Burrows’ Delta analysis, and chi-square analysis).

Juola (2006) asserts that the area of authorship attribution has not been accepted by scholars despite almost 120 years of research, due partly to the limited perceived range of applicability, the inaccuracy of many methods, and the mathematical complexity of those methods. He claims that these problems can be addressed in some part by increasing the number of potential users of authorship attribution technology, and proposes JGAAP, a prototype for an authorship attribution tool. Presently the tool provides a modular and easily extensible framework for importing, processing and scoring of documents, and offers an estimated 20,000 authorship attribution techniques, when taking all combinations of analysis options into account. The architecture of the tool mirrors the theoretical authorship attribution framework proposed by Juola (2006), the structure of the framework being:

1. Canonicization: refers to data type preprocessors used to convert one filetype to another – usually plain text – as well as text preprocessors which prepare the text for analysis by removing punctuation and HTML tags, converting the text to lower case, etc.

2. Determination of the event set: refers to the tokenization of the resulting text, either into words, characters or parts-of-speech, amongst others.

3. Statistical inference: an analysis of the event set is performed using a variety of statistical methods, and then classified.

The design of the app proposed in this project is in the same spirit as the framework above, with the object-orientation paradigm granting a structured, pipelined approach to development. Details of the app structure can be found in section 3.

A notable example of attribution which found success in using lexical features for analysis is in the work of Burrows (1992, 1993). Burrows (2002) presents a measure of stylistic difference (‘Delta’) building upon his previous work, which has been considered to perform very well in attribution tasks. Although not a tool per se, Delta has been implemented as a spreadsheet for personal download and use.

Many more tools exist, however only some of the most salient are covered here. Interestingly, only one case of a stylometric tool implemented as a web app was found during research. I Write
Like is an app which is in the same vein as the one proposed here, in that the user inputs text, and they receive the name of the author whose style is supposedly most similar. Unfortunately, the soundness of the app’s algorithm seems to be a little shaky; while there is some merit in the fact that a Bayesian classifier was used, the author has stated in an interview: “There are a lot of works in academia dealing with writing analysis, but I used none of them.” From the interview, it seems that the algorithm measures low-level style markers such as average sentence length and punctuation. A little more digging around forums reveals some users’ experience with the app, many of them resulting in much confusion about results. Fortunately, the author has plans to improve the program, drawing on academic research. More importantly though, this leaves space for an app in a similar spirit as I Write Like, but perhaps with a more refined method of analysis.

2.3 Document Similarity

So far a brief overview of how to measure style in text has been outlined, with reference to stylometry and authorship attribution. In a way that mirrors the flow of the app, an appropriate topic to discuss after feature selection would be how to measure similarity between documents of text. Textual similarity is principally used in information retrieval, in the ranking of search results from a search engine in order of how similar they are to a user’s query. Below is a brief outline of how calculating textual similarity can be achieved.

2.3.1 Vector Space Model

The Vector Space Model is a mathematical model for representing text documents (or indeed, any object) as a vector (list) of integers. Each dimension of a vector corresponds to a single term in the document, and each value in the vector is the number of occurrences of that term in the document (also known as term frequency). A term can be one of a number of features, for example words, characters, parts-of-speech, etc. If the terms are words, then the dimensionality of the vector will be equal to the number of words in the vocabulary (the set of all the words appearing in the corpus). If the terms are characters, the dimensionality will be equal to the number of characters in the ‘vocabulary’ i.e., the alphabet. This means that if the terms are characters, each letter $a$, $b$, $c$, $d$ . . . will represent a dimension of the vector (in the case of English).

An intuitive way to visualize a document as a vector is to imagine words as terms. The vector representation of the sentence *the sun in the sky* is presented below.

<table>
<thead>
<tr>
<th></th>
<th>sun</th>
<th>in</th>
<th>the</th>
<th>sky</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>
The dimensionality of the vector above is four, as the number of words in the vocabulary is four. In the sentence *the sun in the sky*, the term *the* appears twice, with all the other terms *sun*, *in*, and *sky* appearing once each. Therefore, the sentence can be represented by the term vector \([1, 1, 2, 1]\). It should be noted that when converting a text document into a vector representation, all information about the order and position of words is lost; it is simply a representation of term frequencies. The above sentence could also be represented as \([2, 1, 1, 1]\) or indeed any permutation of those four integers. By converting all the text documents in a corpus to vectors of term frequencies, vector operations can be used to calculate the similarity between two documents.

### 2.3.2 N-grams

As mentioned in section 2.2.1, the two style markers to be analysed within the app are word n-grams and character n-grams. An n-gram is a sequence of \(n\) terms from a given sequence of terms, where, again, terms can be a number of things. Obtaining all the n-grams in a sequence of terms can be imagined as placing a window in which only \(n\) terms of the sequence are visible at any one time over the sequence of terms, and then moving that window term by term through the sequence. N-grams when \(n\) is equal to one are known as unigrams, when \(n\) is equal to two they are known as bigrams. Taking the example in the previous section, *the sun in the sky*, the word unigrams of that sentence will be *the*, *sun*, *in*, *the*, and *sky*. The word bigrams for the sentence will be *the sun*, *sun in*, *in the*, and *the sky*. Illustrative examples can be seen below.

- Character unigrams: \([t, h, e, s, u, n, i, n, t, h, e, s, k, y]\)
- Character bigrams: \([th, he, su, un, ni, in, nt, th, he, es, sk, ky]\)
- Word unigrams: \([the, sun, in, the, sky]\)
- Word bigrams: \([the sun, sun in, in the, the sky]\)

Note that this variation of character bigrams above does not take into account spaces. Indeed, if only letters are of interest – as is the case for the app – whitespace and punctuation will not be included.

At this stage we have a way of representing the documents in the corpus within the app – as term vectors of \(d\) dimensions, where \(d\) is equal to the number of distinct terms in the corpus, and the terms are either word or character n-grams.

### 2.3.3 Cosine Similarity

Cosine similarity is a measure of similarity between two vectors which measures the cosine of the angle between them. Two vectors pointing in the same direction will have a cosine of 1 and
contain the same terms, while two vectors at 90° will have a cosine of 0, and share no terms. It is a useful similarity measure because it takes into account the orientation of a vector rather than the magnitude. For example, a document containing the term cat 250 times and another document containing the term cat 20 times will point in the same direction. The angle between them will be small, and their similarity will be high. If instead of cosine similarity, Euclidean distance were used as a similarity measure, the two documents would be ranked as less similar, as the first document would have a higher magnitude than the second, and as a result the distance between the vectors would be greater.

Cosine similarity is calculated using the dot product and magnitude of two vectors. The dot product of two vectors \( A \) and \( B \) is the sum of the products of the corresponding entries in \( A \) and \( B \), or in mathematical terms:

\[
A \cdot B = \sum_{i=1}^{n} A_i B_i = A_1 B_1 + A_2 B_2 + \cdots + A_n B_n
\] (1)

The magnitude of a vector is the square root of the sum of the square of each entry in the vector, or:

\[
\|A\| = \sqrt{\sum_{i=1}^{n} A_i^2} = \sqrt{A_1^2 + A_2^2 + \cdots + A_n^2}
\] (2)

The cosine similarity of two vectors is the dot product of the vectors divided by the product of the two vectors’ magnitudes. The equation for cosine similarity of two vectors \( A \) and \( B \) is defined as:

\[
similarity = \cos \theta = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n}(A_i)^2} \times \sqrt{\sum_{i=1}^{n}(B_i)^2}}
\] (3)

Two texts in a corpus can end up being very sparse (most elements equal to 0) if they do not contain many of the terms in the corpus vocabulary. In the calculation of cosine similarity, only the non-zero dimensions of a vector are considered. Thus, cosine similarity is popular as a similarity measure due to its computational efficiency.

### 2.3.4 Tf-Idf Weighting

One pitfall of representing documents as term frequency vectors is that very common terms such as the, is, on, etc. will cause two documents to appear very similar to each other, as both documents will likely contain high frequencies of these words. A solution is to create a stopword list, which usually consists of function words, and remove those stopwords from the documents.
Although it is a plausible solution for terms such as the ones above, other common terms which are not common enough to be considered function words cannot readily be handled by a stopword list.

Tf-idf weighting provides a solution to this problem. Tf-idf is a statistic that indicates how important a word is to a document in a corpus. The basic idea is that the tf-idf value of a term $t$ in a document $d$ increases proportionally to the number of times $t$ appears in $d$, but is offset by the frequency of $t$ in the corpus. In other words:

1. If a term appears frequently in a document, it’s important. Give the term a higher score.

2. But if a term appears in many documents, it’s not as important. Give the term a lower score.

Tf-Idf is short for term frequency–inverse document frequency. As such, to calculate the tf-idf value of a term, these two values must first be calculated. The frequency of a term $t$ in a document $d$ is the number of times $t$ appears in $d$. However, simply taking the frequency of $t$ may not always be reliable, as it does not factor in the length of the document. It is usually a good idea to normalize the frequency value of a term, by dividing it by the number of terms in the document. This prevents bias towards longer documents in the corpus.

The inverse document frequency is a statistical representation of term specificity, conceived by Karen Spärck-Jones (1972). Essentially it is an indicator of how rare a term is across a set of documents, which is potentially an indicator of how much information it provides. Although inverse document frequency has not yet been completely justified in information theory, it works well as a heuristic. The inverse document frequency of a term $t$ can be calculated by dividing the total number of documents in the corpus by the number of documents containing $t$, and then taking the logarithm of that quotient. The formula is defined as:

$$idf(t, D) = \log \frac{N}{|\{d \in D : t \in d\}|}$$

where

$N$: the total number of documents in the corpus

$|\{d \in D : t \in d\}|$: the number of documents in the corpus in which $t$ appears.

All that remains after calculating term frequency and inverse document frequency is to multiply them together to get the tf-idf value of a term. For each document in the corpus, the tf-idf value of each term in the corpus vocabulary, with respect to that document, is calculated.
2.3.5 Chi-Square Similarity

An alternative method to cosine similarity as a similarity measure is using chi-square tests. Given two documents A and B, for each term \( t \) in the vocabulary, the chi-square value for \( t \) is calculated. This is done through the use of a 2x2 contingency table, the cells of which represent the number of occurrences and non-occurrences of \( t \) in A and B. Pictured below is the contingency table.

<table>
<thead>
<tr>
<th>term</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>term</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: 2x2 contingency table

For a given term \( t \), the table is filled with the frequency of \( t \) in document A, the frequency of \( t \) in document B, the number of times \( t \) does not appear in A, and the number of times \( t \) does not appear in B. These are the observed values – the frequencies we have counted in the text. From these the expected values are calculated for each cell, with the formula \((\text{row total} \times \text{column total}) / n\), where \( n \) is the total of all observed values. Once the observed values and expected values have been calculated, it is simply a matter of plugging them into the chi-square formula, defined as:

\[
\chi^2 = \sum \frac{(O - E)^2}{E}
\]  

(5)

This formula is applied to each term in the vocabulary, obtaining a \( \chi^2 \) value for each term.

To calculate document similarity using chi square, an average is taken of the \( \chi^2 \) value of each term in the vocabulary. That is, the sum of the \( \chi^2 \) values of all of the n-grams tested is divided by the number of n-grams tested. One advantage of using chi-square similarity over cosine similarity is the ability to pinpoint exactly where two documents are similar or dissimilar. Since a \( \chi^2 \) value has been assigned to each term in the vocabulary, simply observing which terms have the largest \( \chi^2 \) values will reveal the differences between the documents. Conversely, observing the terms with a smaller \( \chi^2 \) value will reveal the similarities between the documents.
3 Application Structure

This section gives a detailed overview of the architecture of the application, and each step of the execution flow of the program is explained. Choices that were made with respect to the design of the app are discussed, and alternatives considered. First a high-level overview is given, and then a low-level look at the code is presented.

3.1 High-Level Overview

It would seem that a top-down approach to looking at the app’s design is convenient, rather than jumping straight into the code itself. In this way, it is easier to gain an understanding of the sub-systems that constitute the app, and subsequently looking at each of those sub-systems’ roles in the system. The app itself is written in Python – the reasons for this are primarily down to its readability, ease of use, object-orientation capability, and its web frameworks (specifically Flask, discussed in section 3.5).

The app can loosely be described by three components – the text processing component, the arithmetic component, and the ‘operation’ component. The text processing component provides classes for the tokenization of text and the weighting of terms according to a given weighting factor. The arithmetic component is simply a module which provides functions for calculations, namely those associated with cosine similarity and chi-square similarity. The operation component, for want of a better term, is the ‘brain’ of the application. It includes the module for initialising the app within the web framework, as well as a Comparator module which essentially compares a set of documents.

A diagram of the app structure is provided in Figure 1 below, and a higher resolution version is available in the Appendix.
The app was designed with modularity and extensibility in mind, such that another person could plug in one of the modules into their own program if they were so inclined, without much refactoring of code. For example, the Tokenizer classes can immediately be used with any text in another program. Following is a deeper look into the code of the app.

### 3.2 Text Processing

As mentioned above, the text processing component of the program consists of a set of classes facilitating the tokenization of text into character or word n-grams, and the conversion of a set of these tokens into a weighted term vector.

#### 3.2.1 Tokenization

Two classes are provided for text tokenization, and are stored in the `Tokenizer` module. A module is simply a file which allows the grouping of classes to facilitate organisation and pragmatics. The two tokenizer classes are CharTokenizer and WordTokenizer. When the user inputs text in the app’s interface and chooses the parameters they wish to use in the analysis, an instance of one of the two above classes is created. More detail on this is provided later. Both classes have a `tokenize` method, which employs the use of regular expressions to tokenize the text. In the case of WordTokenizer, the method splits the text by whitespace and removes any punctuation, and...
in the case of CharTokenizer, the method removes any character that is not alphabetic. From these tokens, a set of n-grams is generated.

The regular expressions used in each class’s tokenize method are seen below.

CharTokenizer

```python
re.sub(r'[^a-z]+', '', text.lower())
```

WordTokenizer

```python
re.sub(r'[
\[\].\-!?#\%\*()]++', '', text.lower())
```

In addition to the tokenize method, the WordTokenizer class defines a constructor, into which a stopword list can be passed as an argument. These stopwords are then removed from the resulting tokens of the tokenize function. An English stopword list is already included in the app package, a copy of which can be found in the Appendix.

### 3.2.2 Vectorization and Weighting

After the text has been tokenized, if cosine similarity is to be used as a similarity measure, it is necessary to convert the list of tokens into weighted term vector. Here, the term ‘vectorization’ is used loosely; there is no need to use a Vector object to represent the documents in Python, as the built-in list data structure is sufficient. Regardless, the decision was taken to implement a Vectorizer class to ‘convert’ a list of tokens into a weighted term vector. The reason for this is purely to aid in conceptualising the structure of the program; just as the Tokenizer module produced a list of tokens from some text, the Vectorizer module produces a term vector from a list of tokens, even though there is no ‘conversion’ happening as such.

To illustrate the vectorization of tokens, it is best to start with the Weighter module. As the name implies, the module provides classes to calculate the weight of a term in a document. Two classes are included, with more to be added in the future. The first is the Weighter class, whose only method is `factory`, which receives a collection of documents, and the type of Weighter to be created as input. The `factory` method creates a different type of Weighter based on the argument passed in. For example, the method call:

```python
weighter = Weighter.factory(document_list, 'tf-idf')
```

will create an instance of a TfidfWeighter, which itself is a subclass of Weighter. As different types of Weighter classes are added to the module in the future, only a quick update of the factory method is required so that they can be used. The TfidfWeighter class is concerned with calculating the tf-idf of a term in a document. Since the calculation of tf-idf requires information about the corpus – a global variable – an instance of the class is initialised with a document list
as an argument. The methods provided by the class include:

- \textit{freq(term, document)}: calculates the frequency of the term in the document.
- \textit{term\_freq(term, document)}: normalises the frequency by dividing it by the number of documents in the corpus. Calls the \textit{freq} method above.
- \textit{n\_containing(term)}: calculates the number of documents containing the term.
- \textit{idf(term)}: calculates the idf of the term. Calls the \textit{n\_containing} method above.
- \textit{tfidf(term, document)}: calculates the tf-idf value of a term. Calls the \textit{term\_freq} and \textit{idf} methods above.

While it is of course possible to compute tf-idf within a single function, splitting the steps up into their own functions facilitates easy reading, understanding and editing of the code, and is congruent with the idea of modularity expressed earlier.

The Vectorizer class imports the Weighter module, and simply takes a list of tokens as input and weights them, returning a list of weighted term frequencies. A single method is provided by the class, which is \textit{weight\_terms}. It accepts a tokenized document, a tokenized corpus and a weighting factor as arguments. A vocabulary is generated from the corpus of documents, which is then used by the instance of Weighter created by the Vectorizer, to generate a list of weighted term frequencies for each term in the vocabulary. As mentioned earlier, there is no real need to implement this step as a class in itself, but it does help with understanding the flow of the application, and is analogous to the role of Tokenizer described above.

3.3 Arithmetic

Described here is the Arithmetic module, a class-less module which provides all the necessary functions for the document similarity calculations.

3.3.1 Cosine Similarity

The calculation of the cosine similarity between two vectors is relatively simple, and rests solely on two other calculations, namely the dot product of two vectors and the magnitude of each vector. As with the implementation of tf-idf calculation, the steps for calculating cosine similarity are split into individual functions, for the same reasons as above. The functions are therefore:

- \textit{dot\_prod(vec1, vec2)}
- \textit{magnitude(vector)}
- \textit{cosine(vec1, vec2)}
with the \textit{cosine} function calling the other two functions. Python’s rich syntax means that these functions are each only one line of code long, with no sacrifice to readability.

\subsection*{3.3.2 Chi Square Similarity}

Three functions are provided for the calculation of chi square similarity. The \texttt{chi\_square} function computes the $\chi^2$ value for a given term with respect to a pair of documents. The \texttt{chi\_average} function computes the average of the $\chi^2$ values of all the terms with respect to a pair of documents – this function calls the \texttt{chi\_square} function.

The calculation of chi-square similarity as described earlier in section \ref{2.3.5} requires only these two functions. However, a third function was added, \texttt{chi\_statistic}, following a proposal by Chen \& Chen (2010) that a chi statistic can be used to calculate the similarity between two term frequency vectors. Unlike the method using the 2x2 contingency table described in section \ref{2.3.5}, the chi statistic method does not take into account the presence and absence of each term in the vocabulary; only the frequencies of the terms themselves are used in the calculation.

The \texttt{chi\_statistic} function is defined as follows:

\begin{equation}
\chi^2 = h \left[ \sum_{i=1}^{n} \frac{x_i^2}{\text{sum}(x)(x_i + y_i)} + \sum_{i=1}^{n} \frac{y_i^2}{\text{sum}(y)(x_i + y_i)} \right] - h \tag{6}
\end{equation}

where

\begin{align*}
\text{sum}(x) &= \sum_{i=1}^{n} x_i, \\
\text{sum}(y) &= \sum_{i=1}^{n} y_i,
\end{align*}

and

\begin{align*}
h &= \text{sum}(x) + \text{sum}(y)
\end{align*}

The \texttt{chi\_statistic} function is, at the moment, not used in the app, but is included in the module in case this changes in the future, and/or if the code is to be eventually released as open source (likely).

\subsection*{3.4 The Comparator Module}

The purpose of the Comparator module is simply to compare two documents within a corpus. This is the module that the top layer of the app will ‘speak’ to. When a user submits their text to the app, they specify a number of settings to be used in the calculation of similarity. When this happens, an instance of the Comparator class is initialised with these settings. During initialisation, the Comparator object creates a Tokenizer object and tokenizes all the documents in the corpus according to the user’s choice of feature (word or character n-grams).
The class provides two methods for the comparison of two documents: `get_cosine_similarity` and `get_chisimilarity`. In the case of the user choosing cosine similarity as a similarity measure, the Comparator object creates a Vectorizer object, and weights the terms of both documents according to the weighting factor specified by the user.

In the case of the user choosing chi square similarity, the Comparator object simply calls the `chi_average` function from the arithmetic module presented earlier, and returns the result.

### 3.5 The Flask Framework

#### 3.5.1 Overview

Flask is a web microframework tailored mainly to small-scale web applications (larger applications tend to favour Django as a framework). It is termed a microframework as it does not assume that a developer should want to use a specific tool or library; Flask does not come with any database abstraction layer, form validation, or any other components for which third-party libraries provide functionality, however it does support third-party extensions.

The following example should illustrate the simplicity of how a basic app is structured, compared to larger and more complex frameworks. A ‘Hello World’ program can be implemented as follows:

```python
def hello():
    return 'Hello World!'

if __name__ == '__main__':
    app.run()
```

Flask uses the Jinja2 template engine to display web pages. A template engine allows a Python application to dynamically generate markup, allowing values from the app to be passed to the template. The templates were written in HTML, supported by CSS. Images of the web pages are included in the appendix.

#### 3.5.2 The stylo Module

The stylo module is where the app is initialised (‘Stylo’ being a working title for the app), and represents the top layer of the app, which the user directly interfaces with. All the web-related functionality happens in this module, such as connecting to the database and rendering HTML.
templates for the user. The following functions are provided by the module:

- `connect_db()`: connects to the database specified in the app’s configuration file (config.py). The database management system used is SQLite3.

- `get_texts()`: retrieves the corpus of texts from the database.

- `get_most_similar(query, doclist, feature_set, system, n, stopwords, weighting_factor)`: takes the user’s choice of settings as input arguments and initialises a Comparator object with these settings. For each document in the corpus, the Comparator computes the similarity between it and the user’s query. The name of the author whose document yielded the highest similarity is returned.

- `home()`: this is an app routing function which is called when a user navigates to the homepage. The `get_texts` function is called here, to retrieve the corpus of texts from the database.

- `display_results()`: this is another app routing function, which renders the results page template to the user, once they click submit. The user’s settings are passed to the template so that they may be displayed on the results page, along with the name of the author whose document scored highest in similarity against the user’s query.

### 3.6 Program Flow

The ‘brain’ of the app lies in the interaction between the stylo module and the Comparator class. As described above, the Comparator class represents an object whose exclusive role is to take a corpus and two documents as input, and use cosine or chi-square similarity to calculate the similarity between the two documents. Much like the Vectorizer class, there is no explicit need to implement this functionality as a class in itself, but again it aids in conceptualising the flow of the program and the roles each component plays.

The app's functioning revolves around the creation of one Comparator object for each user query submitted to the app. The reason for this is that for each user query, that user specifies the parameters used in the computation of similarity: selection of features for tokenization (character or word n-grams), the n value for those n-grams, the method of calculating similarity (chi-square or cosine), any stopwords they wish to use, and the weighting factor (in the case of cosine similarity). The Comparator object is initialised with these settings, and compares the user’s query to each document in the corpus one by one, using these settings. All the tokenization, vectorization and weighting is handled internally by the Comparator object; it creates its own Tokenizer and Vectorizer (which itself creates its own Weighter) based on the user’s choice of settings. It is hopefully clear that this is much more preferable than having numerous control
flow operators in the main function evaluate the settings chosen by the user and performing such-and-such an operation based on those settings. This means that the only other module that the stylo (main) module interacts with is the Comparator module. What results is a compositional and compartmentalised app structure, with each component having its own specific, singular role – achieving what is hoped to be a certain degree of good object-oriented design.

3.7 Corpus

A corpus of 30 excerpts of books by various classic authors is used in the app, with about two or three texts per author. These texts were downloaded from Project Gutenberg, a free digital library. Early tests of the app revealed computational inefficiency when performing calculations on whole books – as such, 1000-word excerpts of the books are stored in the database for the first version of the app. More discussion about this follows in sections 4 and 5. A full listing of the texts in the corpus can be found in the Appendix.

3.7.1 Database

For each text in the corpus, the following data is stored in the database:

- a unique identifier (id)
- the name of the author
- the name of the text
- the text itself

In the future, information about the text’s terms will be stored in the database, namely lists of the text’s word unigrams, bigrams and trigrams, and character unigrams, bigrams and trigrams. Vector representations of these will also be stored. In this way, the corpus will no longer need to be tokenized or vectorised at runtime – the database will simply be queried for the necessary information, saving much computation time.

4 Future Work

Much future work is planned for this project, involving mainly the addition of features but also a possible restructuring of the app. Some of these changes are discussed in this section.

4.1 Extending the Project

If the eventual aim of the project as discussed in section 1.2 is to be achieved, much work needs to be done; many features need to be added in order for the app to be considered as something
more than a novelty tool. Virtually all of the modules could do with more features, as the current set is a little limiting:

- Functions for counting low-level and syntactic features would be useful.
- The Tokenizer module can be extended to include parts-of-speech tokenization.
- The Weighter module can be extended to include more types of weighting factors.
- The Vectorizer and Comparator modules may be eliminated altogether, so as to avoid superfluous code. Their functionality may be expressed elsewhere in the app.
- If more functions are added to the arithmetic module, it could make sense to start grouping the functions into classes, e.g. a chi-square class.
- Creation of Document and Collection classes was undertaken at one point, but due to time constraints these updates had to be rolled back. They may be added again in the future.
- As mentioned above, storing each text’s tokens and term vectors for word and character n-grams in the database would improve performance. This was also implemented but a rollback was necessary, again due to time constraints.
- Allowing the user a closer inspection of the results, with data visualisations. Using chi-square analysis, the user could see which exact terms they have in common with the author they were matched to.
- File upload to allow users to perform their own authorship attribution experiments. Statistical methods used for classification would therefore need to be added.
- File upload to the database itself, resulting in a user-contributed database of texts.
- Last but not least, a full rewrite of the app in Javascript has been considered. This is mainly to avoid the necessity of a web framework altogether. Increase in performance is also expected.

### 4.2 Other Applications

The use of this app as an interesting tool to compare one’s style of writing to those of famous authors, could easily be extended to fit other applications. One which is mentioned briefly above is that of an authorship attribution tool, allowing users to upload their own corpus of texts and conduct their own experiments, without the need to download a program.

Another potentially exciting application is utilising the (eventual) stylometric comparison power of the app and integrating it as part of an author or book recommendation tool, as a sort of literary version of Spotify Discover or iTunes Genius. Although book recommendation tools
do exist online, most, if not all of them make their recommendations based on user reviews of
books and authors. Taking a different approach based on meta-knowledge of the text would be
interesting. Clustering authors based on genre and/or stylistic features, and visualising the data
as node-link diagrams or treemaps, among others, is also an exciting prospect.

5 Conclusion

A prototype for a stylometry tool in the form of a web app has been proposed. However, much
work needs to be done in order to guarantee accuracy and credibility of the tool. The integration
of the tool into other applications in the future is an exciting prospect. This project will be
continued, and hopefully it can gain traction among Internet users.
Appendices
Appendix A

Code Listings

A.1 Tokenizer.py

```python
import re
import string

class CharTokenizer(object):
    def tokenize(self, text, n=1):
        text = re.sub(r'[a-z]+', '', text.lower())
        n_grams = [text[i:i+n] for i in range(len(text) - n+1)]
        return n_grams

class WordTokenizer(object):
    def __init__(self, stopwords=None):
        if stopwords is not None:
            with open(stopwords) as file:
                self.stopwords = [line.strip() for line in file]
        else:
            self.stopwords = None

    def tokenize(self, text, n=1):
        text = re.sub(r'[^a-zA-Z\-.,!?\:;\%\(\)\+\-,\s]', '', str(text))
        tokens = text.lower().split()

        if self.stopwords is not None:
            tokens = [word for word in tokens if word not in self.stopwords]
```

27
n_grams = [''].join(x) for x in [tokens[i:i+n] for i in range(len(tokens) - n+1)]
return n_grams

A.2 Weighter.py

```python
import math

class Weighter(object):
    def factory(doclist, type):
        if type == 'tf-idf' or 'tfidf':
            return TfidfWeighter(doclist)
        factory = staticmethod(factory)

class TfidfWeighter(Weighter):
    def __init__(self, doclist):
        self.doclist = doclist

    def freq(self, term, document):
        return sum(1.0 for w in document if w == term)

    def term_freq(self, term, document):
        return self.freq(term, document) / len(document)

    def n_containing(self, term):
        return sum(1 for doc in self.doclist if term in doc)

    def idf(self, term):
        return math.log((len(self.doclist) / 1 + (self.n_containing(term))), 2)

    def tfidf(self, term, doc):
        tfidf = self.term_freq(term, doc) * self.idf(term)
```
from .Weighter import Weighter

class Vectorizer(object):

def weight_terms(self, document, doclist, weighting_factor='tfidf '):

    weighter = Weighter.factory(doclist, weighting_factor)
    vocabulary = set([term for doc in doclist for term in doc])
    vector = [weighter.weight(term, document) for term in vocabulary]

    return vector


from math import fsum, sqrt

def dot_prod(vec1, vec2):
    return sum(x * y for x, y in zip(vec1, vec2))

def magnitude(vector):
    return sqrt(sum(x * x for x in vector))

def cosine(vec1, vec2):
    return dot_prod(vec1, vec2) / (magnitude(vec1) * magnitude(vec2))

def chi_square(term, doc1, doc2):

a = doc1.count(term)
b = doc2.count(term)
c = len(doc1) - a
d = len(doc2) - b

# observed frequencies
o_freqs = [a, b, c, d]
total = a + b + c + d

# expected frequencies
e_freqs = [
    ((a + b) * (a + c)) / total,
    ((b + a) * (b + d)) / total,
    ((c + a) * (c + d)) / total,
    ((d + b) * (d + c)) / total
]

# chi square formula
chi_square = fsum(((x - y) * (x - y)) / (1 + y) for x, y in zip(o_freqs, e_freqs))
return chi_square

def chi_average(doc1, doc2):
    vocab = set(doc1 + doc2)
    chi_total = 0

    for word in vocab:
        chi_total += chi_square(word, doc1, doc2)

    return chi_total / len(vocab)

def chi_statistic(doc1, doc2):
    h_value = sum(doc1) + sum(doc2)
```python
sum_values_one = 0
sum_values_two = 0

for x, y in zip(doc1, doc2):
    sum_values_one += (x * x) / (sum(doc1)) * (x + y)
    sum_values_two += (y * y) / (sum(doc2)) * (x + y)

chi_statistic = (h_value * (sum_values_one + sum_values_two)) - h_value
return round(chi_statistic, 4)
```

A.5 Comparator.py
```python
from .arithmetic import cosine, chi_average
from .Tokenizer import CharTokenizer, WordTokenizer
from .Vectorizer import Vectorizer

class Comparator(object):
    
def __init__(self, doclist, feature_set, n=1, stopwords=None):
        self.n = n

        # initialises a Tokenizer based on arguments
        if feature_set in ("word", "words", "w"):
            self.tn = WordTokenizer(stopwords)
        elif feature_set in ("character", "characters", "char", "c"):
            self.tn = CharTokenizer()

        # tokenizes each document in the collection
        self.doclist = [self.tn.tokenize(doc, self.n) for doc in doclist]

def get_cosine_similarity(self, doc1, doc2, weighting_factor='tfidf'):
```
v = Vectorizer()

# tokenize the documents
doc1 = self.tn.tokenize(doc1, self.n)
doc2 = self.tn.tokenize(doc2, self.n)

vec1 = v.weight_terms(doc1, self.doclist, weighting_factor)
doc2 = v.weight_terms(doc2, self.doclist, weighting_factor)

return round(cosine(vec1, vec2), 4)

def get_chisimilarity(self, doc1, doc2):
    return round(chi_average(doc1, doc2), 4)

A.6 stylo.py

import sqlite3
from flask import Flask, request, session, g, redirect, url_for, abort, render_template, flash
from contextlib import closing
from bin import Comparator

# create the application
app = Flask('stylo')
app.config.from_object('config')
texts = {}

# connects to database
def connect_db():
    return sqlite3.connect(app.config['DATABASE'])

# Retrives the corpus of texts from the database and loads it into a dictionary.
def get_texts():
with closing(connect_db()) as db:
    c = db.cursor()
    c.execute('SELECT * FROM documents')
    rows = c.fetchall()

    for row in rows:
        text = row[3]
        author = row[2]
        texts[text] = author

    return texts

def get_most_similar(query, doclist, feature_set, system, n=1,
                      stopwords=None, weighting_factor='tfidf'):
    c = Comparator(doclist, feature_set, n, stopwords)
    similarities = {}

    if system == 'cos' or 'cosine':
        for text, author in doclist.items():
            # get cosine similarity between the user's text and each
text in the corpus
            similarities[author] = c.get_cosine_similarity(query,
                                                         text,
                                                         weighting_factor)
            print('{}: \t{}: {}'.format(author, text,
similarities[author]))

    elif system == 'chi' or 'chi square':
        for text, author in doclist.items():
            # get the average chi square value between the user's text
and each text in the corpus
            similarities[author] = c.get_chi_similarity(query, text)

        return max(similarities, key=similarities.get)

    # homepage
@app.route('/')
```python
@app.route('/index.html')
def home():
    texts = get_texts()
    return render_template('index.html')

# display results to user
@app.route('/results', methods=['POST'])
def display_results():
    settings = {
        'user_query': request.form['text-box'],
        'system': request.form['system'],
        'feature_set': request.form['feature-set'],
        'weighting_factor': request.form['weighting-factor'],
        'n-value': int(request.form['n-value']),
        'stopwords': request.form['stopwords']
    }

    settings['stopwords'] = None if settings['stopwords'] == "None" else 'docs/stopwords/' + settings['stopwords']
    author = get_most_similar(settings['user_query'], texts, settings['feature_set'], settings['system'], settings['n-value'], settings['stopwords'], settings['weighting_factor'])

    return render_template('results.html', author=author, settings=settings)

if __name__ == '__main__':
    app.run()
```

A.7 config.py

```python
DATABASE = 'tmp/stylo.db'
USERNAME = 'admin'
PASSWORD = 'default'
```
A.8 index.html

```html
<!DOCTYPE html>
<html>
<head>
  <title>Stylo</title>
  <link rel=stylesheet type=text/css href="{{ url_for('static', filename='css/styles.css') }}">
  <link href='http://fonts.googleapis.com/css?family=Varela' rel='stylesheet' type='text/css'>
  <link href='http://fonts.googleapis.com/css?family=Cabin:400,700' rel='stylesheet' type='text/css'>
  <link href='http://fonts.googleapis.com/css?family=Lato:400,700,900' rel='stylesheet' type='text/css'>
  <link href='http://fonts.googleapis.com/css?family=Bitter' rel='stylesheet' type='text/css'>
  <link href='http://fonts.googleapis.com/css?family=Montserrat:400,700' rel='stylesheet' type='text/css'>
  <meta content= "text/html; charset=UTF-8" http-equiv= "content-type">
  <meta name="description" content="Stylo: A Computational Stylometry Tool">
</head>
<body>
  <div id="section">
    <div id="container">
      <h1><a href="index.html">Stylo</a></h1>
      <h2>A Computational Stylistics Tool</h2>
      <p>See which author’s writing style is most similar to yours with this tool, which analyses the stylistic features of your writing and compares them to those of famous authors. For more accurate results, use a sample of at least a few paragraphs. Paste your text below and then click submit to see a run-down of the results.</p>
      <form method="POST" action="{{ url_for('display_results') }}">
        <textarea name="text-box" placeholder="Lolita, light of my life, fire of my loins. My sin, my soul. Lo-lee-ta: the tip of the tongue taking a trip of three steps down the palate to tap, at three, on the teeth. Lo. Lee. Ta. She was Lo, plain Lo, in the morning, standing four feet ten in one sock. She was Lola in slacks. She was Dolores on the dotted line. But in my arms she was always Lolita.">Lolita, light of my life, fire of my loins. My sin, my soul. Lo-lee-ta: the tip of the tongue taking a trip of three steps down the palate to tap, at three, on the teeth. Lo. Lee. Ta. She was Lo, plain Lo, in the morning, standing four feet ten in one sock. She was Lola in slacks. She was Dolores on the dotted line. But in my arms she was always Lolita.
```
```
Did she have a precursor? She did, indeed she did. In point of fact, there might have been no Lolita at all had I not loved, one summer, a certain initial girl-child. In a princedom by the sea. Oh when? About as many years before Lolita was born as my age was that summer. You can always count on a murderer for a fancy prose style. Ladies and gentlemen of the jury, exhibit number one is what the seraphs, the misinformed, simple, noble-winged seraphs, envied. Look at this tangle of thorns.
This tool compares your text to a corpus of 30 other texts by classic authors, and computes which author’s style is most similar to your own. In order to perform the comparisons, your text is broken down into certain features (word or character). After that, the program performs either a cosine similarity or chi square analysis of your text against the corpus of texts in the database.

You may change the settings of the program and select which analysis you want to use, along with the features to extract, and more, by clicking on the red 'Optional Parameters' button above.

To understand more about how the tool works, please refer to the following links:

- Vector Space Model
- TF-IDF Weighting
- Combining TFIDF and Cosine Similarity
- Stop Words
A.9 results.html

```html
<!DOCTYPE html >
<html >
<head >
  <link rel=stylesheet type=text/css href="{{ url_for('static', filename='css/styles.css') }}">
  <link href='http://fonts.googleapis.com/css?family=Varela' rel='stylesheet' type='text/css'>
  <link href='http://fonts.googleapis.com/css?family=Lato:400,700,900' rel='stylesheet' type='text/css'>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type">
</head>
<body >
  <div id="section">
    <div id="container">
      <div id="header">
        <form action="index.html">
          <input type="submit" id="home-button" value="Home">
        </form>
      </div>
      <div class="results-section">
        <h2>Results</h2>
        <p style="text-align: center;">The author you write most like is: 
          <span style="font-weight: 900; color: #2D8A8E">{{ author }}</span>
        </p>
      </div>
    </div>
  </div>
  <div id="break"></div>
  <div class="results-section">
    <h2>Settings</h2>
    <p style="text-align: center;">You can find here the settings used in the computation:
    </p>
    <table id="settings">
      <tr>
        <td>System</td>
        <td>{{ settings.system }}</td>
      </tr>
      <tr>
        <td>Feature Set</td>
        <td>{{ settings.feature_set }}</td>
      </tr>
      <tr>
        <td>N-gram Value</td>
        <td>{{ settings.n_value }}</td>
      </tr>
    </table>
  </div>
</body>
</html>
```
{% if settings.system == "cosine" %}
<tr>
    <td>Weighting Factor</td>
    <td>{{ settings.weighting_factor }}</td>
</tr>
{% endif %}
</table>
<div id="break"></div>
</div>
</div>
</body>
</html>

A.10 styles.css

body {
    background-color: #DECFB9;
}

h1 {
    color: #403027;
    font-family: 'Montserrat', sans-serif;
    font-size: 48px;
    font-weight: 700;
    margin: 60px 0px 5px 0px;
    text-align: center;
}

h2 {
    color: #403027;
    font-family: 'Montserrat', sans-serif;
    font-size: 22px;
    font-weight: 700;
    margin: 0px 0px 20px 0px;
    text-align: center;
}

h3 {
    color: #E74C3C;
    font-family: 'Cabin', sans-serif;
form {
  width: 944px;
  margin: 0px auto 30px auto;
}

.buttons {
  text-align: center;
}

input[type="submit"] {
  width: 100px;
  height: 40px;
  text-align: center;
  margin: 0px auto 5px 250px;
  font-family: 'Lato', sans-serif;
  font-weight: 700;
  font-size: 14px;
  color: #2D8A8E;
  background-color: #DECFB9;
  border: 2px solid #2D8A8E;
  display: inline-block;
  -webkit-transition: background-color 400ms ease;
  transition: background-color 400ms ease;
}

#home-button {
  width: 100px;
  height: 40px;
  text-align: center;
  font-family: 'Lato', sans-serif;
  font-weight: 700;
  font-size: 16px;
  color: #403027;
  margin: 0px;
  background-color: #DECFB9;
  border: 2px solid #403027;
  -webkit-transition: background-color 400ms ease;
  transition: background-color 400ms ease;
}

#home-button:hover, #home-button:hover a {
  background-color: #403027;
  color: white;
cursor: pointer;
-webkit-transition: background-color 400ms ease;
transition: background-color 400ms ease;
}

input[type="radio"] + label, input[type="text"] + label {
  font-family: 'Lato', sans-serif;
  font-weight: 700;
  color: #50ba88;
  display: inline;
  font-size: 14px;
}

input[type="text"] {
  width: 20px;
  margin: 10px 0px 0px 0px;
  text-align: center;
  border: 2px solid #403027;
  font-family: 'Montserrat', sans-serif;
  color: #403027;
}

input[type="submit"]:hover {
  background-color: #50ba88;
  border: none;
  color: white;
}

#toggle, #info {
  position: absolute;
  top: -100%;
  left: -100%;
  cursor: pointer;
}

#toggle + label, #info + label {
  cursor: pointer;
}
175  label[for="toggle"], label[for="info"] { 
176    font-family: 'Montserrat', sans-serif; 
177    font-size: 14px; 
178    font-weight: 700; 
179    text-align: center; 
180    display: inline-block;; 
181    width: 165px; 
182    padding: 4px 0px 4px 0px; 
183    -webkit-transition: background-color 500ms ease; 
184    transition: background-color 500ms ease; 
185    margin: 0px 0px 10px 0px; 
186  } 
187 
188  label[for="toggle"] { 
189    float: left; 
190    color: #E74C3C; 
191    border: 2px solid #E74C3C; 
192  } 
193 
194  label[for="info"] { 
195    float: right; 
196    color: #50ba88; 
197    border: 2px solid #50ba88; 
198  } 
199 
200  #toggle:checked ~ .options { 
201    opacity: 1; 
202  } 
203 
204  #toggle:checked + label { 
205    background-color: #E74C3C; 
206    color: white; 
207  } 
208 
209  #info:checked ~ .how-it-works { 
210    opacity: 1; 
211  } 
212 
213  #info:checked + label { 
214    background-color: #50ba88; 
215    color: white; 
216  } 
217 
218  .options { 
219    width: 100%; 
220    display:table; 
221  }
transition: all 500ms cubic-bezier(0.460, 0.155, 0.460, 0.755); /* custom */
-webkit-transition-timing-function: cubic-bezier(0.460, 0.155, 0.460, 0.755);
-moz-transition-timing-function: cubic-bezier(0.460, 0.155, 0.460, 0.755);
-o-transition-timing-function: cubic-bezier(0.460, 0.155, 0.460, 0.755);
transition-timing-function: cubic-bezier(0.460, 0.155, 0.460, 0.755); /* custom */
## Appendix B

### Miscellaneous Files

### B.1 Stopword List

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>about</td>
</tr>
<tr>
<td>above</td>
<td>after</td>
</tr>
<tr>
<td>again</td>
<td>against</td>
</tr>
<tr>
<td>all</td>
<td>am</td>
</tr>
<tr>
<td>an</td>
<td>and</td>
</tr>
<tr>
<td>any</td>
<td>are</td>
</tr>
<tr>
<td>as</td>
<td>at</td>
</tr>
<tr>
<td>be</td>
<td>because</td>
</tr>
<tr>
<td>been</td>
<td>before</td>
</tr>
<tr>
<td>being</td>
<td>below</td>
</tr>
<tr>
<td>between</td>
<td>both</td>
</tr>
<tr>
<td>but</td>
<td>by</td>
</tr>
<tr>
<td>can</td>
<td>did</td>
</tr>
<tr>
<td>do</td>
<td>does</td>
</tr>
<tr>
<td>doing</td>
<td>don</td>
</tr>
<tr>
<td>down</td>
<td></td>
</tr>
</tbody>
</table>

47
during each few for from further had has have having he her here hers herself him himself his how i if in into is isn it its itself just me more most my myself no nor not now
of
off
on
once
only
or
other
our
ours
ourselves
out
over
own
s
same
she
should
so
some
such
t
than
that
the
their
theirs
them
themselves
then
there
these
they
this
those
through
to
too
under
until
up
very
was
we
were
what
when
where
which
while
who
whom
why
will
with
you
your
yours
yourself
yourselves
A Computational Stylistics Tool

Stylo
Stylo
A Computational Stylistics Tool

See which author’s writing style is most similar to yours with this tool, which analyses the stylistic features of your writing and compares them to those of famous authors. For more accurate results, use a sample of at least a few paragraphs.

Paste your text below and then click submit to see a run-down of the results.

Lolita, light of my life, fire of my loins. My sin, my soul, Lo-lee-ta, the tip of the tongue taking a trip of three steps down the palate to tap, at three, on the teeth. Lo. Lee. Ta. She was Lo, plain Lo, in the morning, standing four feet ten in one sock. She was Lola in slacks. She was Dolly at school. She was Dolores on the dotted line. But in my arms she was always Lolita. Did she have a precursor? She did, indeed she did. In point of fact, there might have been no Lolita at all had I not loved, one summer, a certain initial girl-child. In a princely by the sea. Oh when? About as many years before Lolita was born as my age was that summer. You can always count on a murderer for a fancy prose style. Ladies and gentlemen of the jury, exhibit number one is what the seraphs, the misinformed, simple, noble-winged seraphs, envied. Look at this tangle of thorns.

Optional Parameters

- System:
  - cosine similarity
  - chi square

- Feature Set:
  - word n-grams
  - character n-grams
  - p-value

- Stopwords:
  - None

- Weighting Factor:
  - tf-idf

How It Works

This tool compares your text to a corpus of 30 other texts by classic authors, and computes which author’s style is most similar to your own. In order to perform the comparisons, your text is broken down into certain features (word or character n-grams). After that, the program performs either a cosine similarity or chi square analysis of your text against the corpus of texts in the database.

You may change the settings of the program and select which analysis you want to use, along with the features to extract, and more, by clicking on the red ‘Optional Parameters’ button above. To understand more about how the tool works, please refer to the following links:

1. Vector Space Model
2. TF-IDF Weighting
3. Combining TFIDF and Cosine Similarity
4. Stop Words
B.5 Corpus

1. A Christmas Carol by Charles Dickens
2. David Copperfield by Charles Dickens
3. Great Expectations by Charles Dickens
4. The Picture of Dorian Gray by Oscar Wilde
5. Lord Arthur Savile's Crime; The Portrait of Mr. W.H., and Other Stories by Oscar Wilde
6. A House of Pomegranates by Oscar Wilde
7. The Gambler by Fyodor Dostoyevsky
8. The Brothers Karamazov by Fyodor Dostoyevsky
9. Crime and Punishment by Fyodor Dostoyevsky
10. Anna Karenina by Leo Tolstoy
11. War and Peace by Leo Tolstoy
12. The Age of Innocence by Edith Wharton
13. The Reef by Edith Wharton
14. Sanctuary by Edith Wharton
15. The Mysterious Affair at Styles by Agatha Christie
16. The Secret Adversary by Agatha Christie
17. Pride and Prejudice by Jane Austen
18. Sense and Sensibility by Jane Austen
19. Emma by Jane Austen
20. Beyond Good and Evil by Friedrich Nietzsche
21. Thus Spake Zarathustra by Friedrich Nietzsche
22. A Portrait of the Artist as a Young Man by James Joyce
23. Dubliners by James Joyce
24. Adventures of Huckleberry Finn by Mark Twain
25. The Prince and the Pauper by Mark Twain
26. This Side of Paradise by F. Scott Fitzgerald
27. The Beautiful and the Damned by F. Scott Fitzgerald
28. Tales of the Jazz Age by F. Scott Fitzgerald
29. Night and Day by Virginia Woolf
30. The Voyage Out by Virginia Woolf
Bibliography


Mendenhall, T. C. 'The Characteristic Curves Of Composition'. Science ns-9.214S (1887):


