'Stylo': a package for stylometric analyses

Maciej Eder
Pedagogical Univ. of Kraków

Jan Rybicki
Jagiellonian University

Mike Kestemont
University of Antwerp

February 13, 2018

Abstract

The 'stylo' package (Eder et al. 2013) provides easy-to-use implementations of various established analyses in the field of computational stylistics, including non-traditional authorship attribution, genre recognition, style development ("stylochronometry"), etc. The package includes a number of explanatory methods provided by the function `stylo()` (multidimensional scaling, principal component analysis, cluster analysis, bootstrap consensus trees). Addtionally, a number of supervised machine-learning methods are available via the function `classify()` (Delta, support vector machines, naive Bayes, k-nearest neighbors, nearest shrunken centroids). The `rolling.delta()` function analyses collaborative works and tries to determine the authorship of fragments extracted from them. The function `rolling.classify()` offers a more flexible interface to sequential classification of collaborative works. The `oppose()` function performs a contrastive analysis between two given sets of texts: among other things, it generates lists of words significantly *preferred* and *avoided* by one or more authors in comparison to the texts by another author (or a set of them).

Keywords

stylometry, computational stylistics, authorship attribution, cluster analysis, dendrogram, bootstrap consensus trees, PCA, MDS, k-NN, SVM, NSC, naive Bayes, Delta, Zeta, rolling stylometry

Contents

1 Introduction **3**

2 Installation **4**

3 Functions provided **5**

4 stylo() 7
4.1 Corpus preparation . . . 7
4.2 Starting the function . . . 9
4.3 Options available on GUI . . . 9
4.3.1 Input . . . 10
4.3.2 Language . . . 11
4.3.3 Features . . . 11
4.3.4 MFW settings . . . 12
4.3.5 Culling . . . 13
4.3.6 Statistics . . . 14
4.3.7 Distances . . . 15
4.3.8 Sampling . . . 17
4.3.9 Graphs . . . 18
4.3.10 Plot area . . . 19
4.3.11 PCA/MDS . . . 19
4.3.12 PCA flavour . . . 20
4.3.13 Various . . . 20

5 classify() 21
5.1 Corpus preparation . . . 21
5.2 Calling the function . . . 22
5.3 Options . . . 22
5.3.1 Options inherited from `stylo()` . . . 22
5.3.2 Statistics . . . 22
5.3.3 General . . . 23
5.3.4 SVM options . . . 23
5.3.5 k-NN options . . . 23
5.3.6 Output: general . . . 23

6 rolling.delta(), rolling.classify() 24
6.1 Corpus preparation . . . 25
6.2 Calling the function . . . 25
6.3 Options . . . 25
6.3.1 Features . . . 25
6.3.2 Sampling . . . 25
6.3.3 Colors . . . 25

7 oppose() 26
7.1 Corpus preparation . . . 26
7.2 Calling the function . . . 26
7.3 Options . . . 26

8 Options unavailable on GUI 27
8.1 Cluster analysis linkage . . . 27
8.2 Network analysis support . . . 27
8.3 Undocumented arguments . . . 28

9 Advanced topics **29**
9.1 Batch mode 29
9.2 Custom splitting rule 31
9.3 Splitting rule and batch mode combined 31
9.4 Custom similarity measures 32
9.5 Large-scale stylometric tests 33

10 Error messages and troubleshooting **34**

References **34**

1 Introduction

Stylometric studies, in all their variety of material and method, have two features in common: the electronic texts they study have to be coaxed to yield numbers, and the numbers themselves have to be processed via statistics. Sometimes, the two actions are two independent parts of a given study. To give the simplest example, one piece of software is used solely to compile word frequency lists; then, one of the many commercial statistics packages takes over to extract meaning from this mass of words, draw graphs etc.

Yet, as stylometrists have begun to produce statistical methods of their own – to name but a few, Burrows's Delta, Zeta and Iota (Burrows, 2002, 2007) and their modifications by other scholars (Argamon, 2008, Craig and Kinney, 2009, Hoover, 2004a, 2004b) – commercial software, despite its wide array of accessible methods, becomes something of a straightjacket. This is why a number of dedicated stylometric solutions have appeared, targeting the specific analyses frequently used in this community. Hoover's Delta, Zeta and Iota Excel spreadsheets are pioneering examples of this approach (Hoover, 2004b). Constantly developed since 2004, they have at least two major assets: they do exactly what the stylometrist wants (with several optional procedures) and they only require fairly standard – although proprietary – spreadsheet software. This has been especially helpful for uses in specialist workshops and classrooms: the student only needs additional (and, often, free) software to produce word frequency lists and (s)he is ready to go. Yet Excel imposes one important limitation: it is very demanding from the point of view of memory usage. Moreover, the two-stage nature of the process (a separate piece of software prepares word lists that can be later automatically imported into the spreadsheet) might be problematic, because it takes an experienced Visual Basic programmer to make Excel itself extract the various frequency dictionaries needed.

In this respect, Juola's JGAAP can directly import texts in a variety of formats and perform a variety of authorship attribution tasks using an impressive variety of statistical methods (Juola et al., 2008). These can be further expanded by experienced programmers in Java. Java is also the language of another software solution which takes an even broader approach: Craig's Intelligent Archive is able to perform a number of standard stylometric procedures, but it can also be used as a corpus organizer. Once the initial work of registering texts is done, it enables a versatile combination of individual texts and groups of texts (Craig and Kinney, 2009).

Since contemporary stylometry uses either stand-alone dedicated programs custom-made by stylometrists, or applies existing software, the `stylo` package can be situated somewhere in-between: the powerful open-source statistical programming environment

R provides, on the one hand, the opportunity of building statistical applications from scratch, and, on the other, allows less advanced researchers to use ready-made scripts and libraries (cf. http://www.R-project.org). In our own stylometric adventure with R, one of the aims was to build a tool (or a set of tools) that would combine sophisticated state-of-the-art algorithms for classification and/or clustering with a user-friendly, "point-and-click" interface. In particular, we wanted to implement a number of popular multidimensional methods to be used by scholars without advanced programming skills. It soon became evident that once our R scripts were provided with a graphical user interface and modest documentation, they lent themselves well to classroom use. In our experience, this suite of tools offers an excellent way to work around R's typically steep learning curve, without losing anything of the power of the environment – namely R's considerable computing power and speed.

A crucial point in building the interface was to make sure that all stages of a typical stylometric analysis – from loading texts to visualizing the results – could be performed from within a single function. The `stylo()` function, for instance, does all the work: it processes electronic texts to create a list of all the words used in all texts studied, with their frequencies in the individual texts; normalizes the frequencies with z-scores (if applicable); selects words from the desired frequency ranges; performs additional procedures that might improve attribution, such as Hoover's (2004a, 2004b) automatic deletion of personal pronouns and "culling" (automatic removal of words too characteristic for individual texts); compares the results for individual texts; performs a variety of multivariate analyses; presents the similarities/distances obtained in tree diagrams; and finally, produces a bootstrap consensus tree (a new graph that combines many tree diagrams for a variety of parameter values). It was our aim to develop a general platform for multi-iteration stylometric tests; for instance, an alternative script derived from the function `classify()` produced heatmaps to show the degree of Delta's success in attribution at various intervals of the word frequency ranking list (Rybicki and Eder, 2011).

The last stage of the interface design was, firstly, to add a GUI (since some humanists might be allergic to the command-line mode provided by R) and, secondly, a host of various small improvements (like saving and loading the parameters for the most recent analysis, a wide choice of graphic output formats, etc.). Nevertheless, advanced users could still easily switch off the GUI and embed the functions provided by the "stylo" library in their own scripts.

2 Installation

Make sure you are connected to the internet. Launch R. Type `install.packages("stylo")` in the console. Whenever you start a new R session, type `library(stylo)`. This will automatically load all the functionality provided in the package (see below). If you are very lazy and only use R for stylometric purposes, you can find your `Rprofile.site` configuration file (in `R/R-<your R version here>/etc`), open it with administrator privileges and insert the line `library(stylo)` there. In this case, the "stylo" library will be loaded at the start of each R session and you can start invoking the particular functions right away.

3 Functions provided

The most important tools included in this package are distributed over the following functions:

- `stylo()`
- `classify()`
- `oppose()`
- `rolling.delta()`
- `rolling.classify()`

The next sections of this manual describe these four functions together will all the different input options they can take. If you want to get a general overview of these four functions, type `help(stylo)`, `help(classify)`, etc., and a help window will appear. More advanced users might be interested in some other functions provided by the library. Generally speaking, they are a great deal of lower-level functions which are called automatically from inside the upper-tier functions, such as `classify()`, `oppose()`, etc. This lower-level functionality can of course be used for developing your own scripts and functions. These include:

- `assign.plot.colors`
- `define.plot.area`
- `delete.markup`
- `delete.stop.words`
- `dist.argamon`
- `dist.cosine`
- `dist.delta`
- `dist.eder`
- `dist.simple`
- `draw.polygons`
- `gui.classify`
- `gui.oppose`
- `gui.stylo`
- `load.corpus.and.parse`
- `load.corpus`
- `make.frequency.list`
- `make.ngrams`

- `make.samples`
- `make.table.of.frequencies`
- `parse.corpus`
- `parse.pos.tags`
- `perform.culling`
- `perform.delta`
- `perform.knn`
- `perform.naivebayes`
- `perform.nsc`
- `perform.svm`
- `stylo.default.settings`
- `stylo.pronouns`
- `txt.to.features`
- `txt.to.words.ext`
- `txt.to.words`
- `zeta.chisquare`
- `zeta.craig`
- `zeta.eder`

In most cases, these lower-level functions provide very basic processing functionality and they are therefore not intended to be invoked by everyday users. Hence, they will not be discussed in this manual. However, if you are interested how they work and how to use them, you can invoke the help pages for these functions: `help(load.corpus)`, `help(make.ngrams)`, etc. Help pages routinely contain some insightful examples as to how to use the code: refer to them if you want to understand what a particular function does. The examples can be copy-pasted into an active R console. (Don't be afraid of the lines '`## Not run`' – they prevent R to run some automatic checks on interactive functions; you can use these examples safely).

Apart from functions, the package '`stylo`' (ver. ¿0.6.1) contains three datasets that can be used to start playing with stylometric methods without any actual texts. The datasets are as follows:

- `novels`
- `galbraith`
- `lee`

The first dataset contains 9 full-size novels by Jane Austen and the Brontë sisters. The second and the third set contains computed tables of word frequencies for 26 and 28, resp., contemporary novels that for copyright-related reasons could not be made available in their original format. A detailed description of the datasets can be retrieved via `help(novels)`, `help(galbraith)` and `help(lee)`.

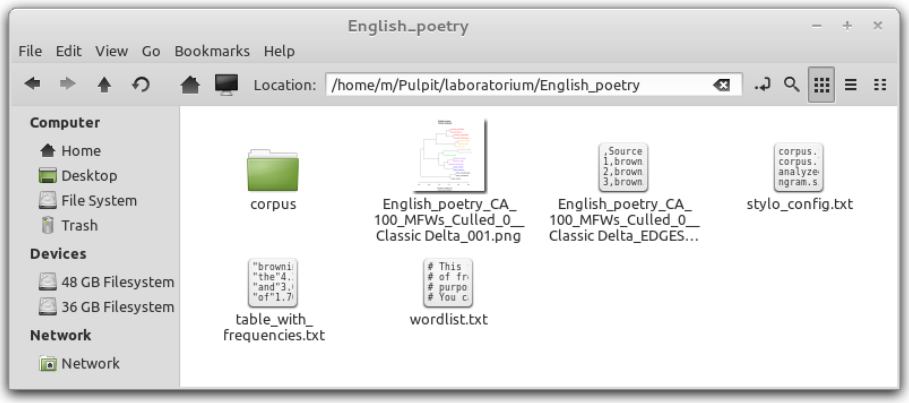

Figure 1: Working directory containing a subdirectory `corpus` and some files generated by the function `stylo()`.

4 stylo()

This is currently the main tool in the package. The function `stylo()` is meant to enable users to automatically load and process a corpus of electronic text files from a specified folder, and to perform a variety of stylometric analyses from multivariate statistics to assess and visualize stylistic similarities between input texts. This function provides explanatory analyses; any users interested in machine-learning supervised methods might want to skip this section and go to `classify()`, below.

`stylo()` will typically be used to produce a most-frequent-word (MFW) list for the entire corpus. Next, it will acquire the frequencies of the MFWs in the individual texts to create an initial matrix of words (rows) by individual texts (columns): each cell will contain a single word's frequency in a single text. Subsequently, it will normalize the frequencies: it selects words from the the desired frequency ranges for an analysis (this is also saved to disk as `table_with_frequencies.txt` and it will perform additional processing procedures (automatic deletion of personal pronouns and culling, see 4.3.5 below) to produce a final wordlist for the actual analysis (this information is saved to disk in the current working directory as `wordlist.txt`. It then compares the results for individual texts, performing e.g. distance calculations and using various statistical procedures (cluster analysis, multidimensional scaling, or principal components analysis). Finally, the function will produce graphical representations of distances between texts and it will write the resulting authorship (or similarity) candidates to a logfile (`results.txt`) in the current working directory. When the `consensus tree` option is selected, the script produces virtual cluster analyses for a variety of parameters, which then produce a final diagram that reflects a compromise between the underlying cluster analyses.

4.1 Corpus preparation

The procedure of loading corpora as described immediately below is probably the best way to start doing your first analyses. However, experienced users of R sooner or later will discover that input data structures (corpora, vectors of features, tables of frequencies) can be passed as R objects directly from, say, other functions, without any interaction with texts files. Refer to section 9.1 for details.

Each project requires a separate and dedicated working folder. You will want to give it a meaningful name (like `SanskritPoetry11` rather than `Blah-blah`), since the

name of the folder will appear as the title in your graphs generated by the function. By default, the results of your analyses and other useful files will be written automatically to this folder. The actual text files for your analyses must be placed in a subfolder in the working directory, named `corpus` (Note: all file names are case sensitive!). All functions in this tool suite expect to find at least two input texts for their analyses.

The text files need to follow the following naming syntax: `category_title.txt`. For people working in authorship attribution, the `category` will capture a text's authorial signature; other users, perhaps interested to compare a translators' styles, should name their files `translatorname_title.txt`. Likewise, if you are looking for stylistic similarities between writers of the same gender, use `gender_title.txt`, etc. It is really important to use an underscore "_" (underscore) as a delimiter: e.g. colors on the final graphs will also be assigned according to strings of characters up to the first underscore in the input files' names. (For further details and examples, type `help(assign.plot.colors)`). Consider the following examples, in which the classes are the authors' names and authors' gender, respectively:

```
ABronte_Agnes.xml
ABronte_Tenant.xml
Austen_Emma.xml
Austen_Pride.xml
Austen_Northanger.xml
Conrad_Nostromo.xml
Conrad_Lord.xml
Dickens_Pickwick.xml
...
```

```
M_Conrad_Lord_Jim.txt
M_Joyce_Dubliners.txt
F_Woolf_Night_and_day.txt
F_Woolf_Waves.txt
...
```

Everything that comes after the underscore (say, the short titles of novels) can be followed by any other information. Be careful with long names, however, since these might not fit in the graphs that will be generated. The texts must either be *all* in plain text format, or *all* in HTML, or *all* in TEI-XML (the latter two options have not been extensively tested so far, and should be used carefully).

A concise remark about possible encoding issues should also be added. If the operating system you use is Linux or Mac, you just need to make sure the texts are all in UTF-8 (aka Unicode). If your operating system is Windows, you have two options. Firstly, you might want to save all the texts in ANSI codepage, but you have to tread carefully if your machine runs one charset, say, Central European (1250) and your texts are in the Western European codepage (1252); in this respect, for instance, French is notoriously difficult (*nous sommes vraiement désolés*). Alternatively, you can convert your texts into Unicode (a variety of freeware converters are available on the internet), and to use an appropriate encoding option when launching the function, say, `stylo()` either by clicking the "UTF-8" button on GUI (beginners), or passing the argument `encoding = "UTF-8"` directly to the function (advanced users).

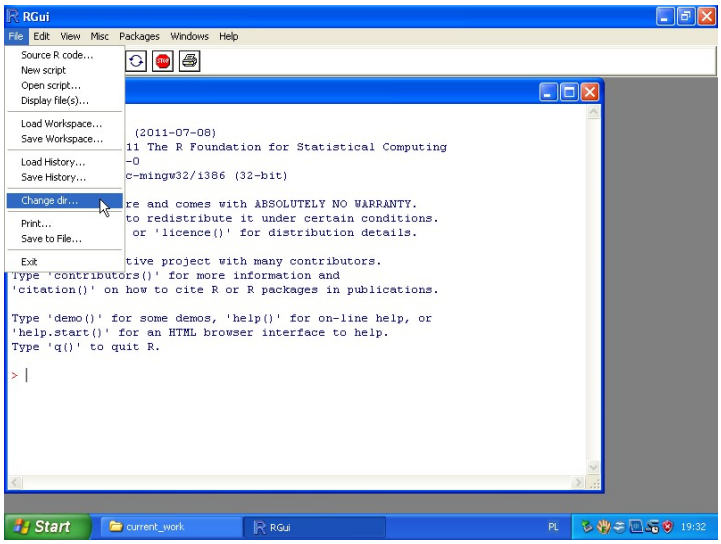

Figure 2: R console on Windows: easiest way to set working directory

4.2 Starting the function

Start up R. At the prompt (where you see the cursor blinking), move to your folder (the main folder you will be working in, *not* the `corpus` subfolder) using the command `setwd()`. E.g.:

```
setwd("/Users/virgil/Documents/disputed-works-of-mine")
```

You can use either absolute paths (as in the above example), or relative ones, i.e. you can navigate directly from the current working directory. If you want to go, say, two levels up and then descend to a folder `first_experiment`, type:

```
setwd("../../first_experiment")
```

You can always check you current working directory typing `getwd()`. (If you use R app for Windows, you can set your directory by clicking the *File* menu: see Fig. 1; Mac OS users – click the *Misc* menu on your R console). Call the function by typing `stylo()` at the prompt and hitting enter. After a while, you should see a GUI box appear on the screen. Change as many options as you need. Since there are multiple tabs in the GUI, make sure you only click the OK button after you've set the parameters in all the tabs. Shortly afterwards, you will see the names of the files processed appear in the R console, followed by other (technical) information. Depending on the size of your corpus, this step might take a few minutes. When the process is completed without major errors, you will typically see a diagram on your screen; otherwise, a graphic file (you can choose one or more format if you like) will be saved in your working directory (at better resolution than the onscreen version, so use this for your publication), and you can start exploring the other `stylo()` output files there.

4.3 Options available on GUI

As a first step, beginners should learn how to use the graphical user interface (GUI), which allows you to control the script's main parameters without having to tamper with the actual code. However, if you *do* prefer to tamper with the code, you can call the function in batch mode: `stylo(gui=FALSE)`. In that case, before you start, you might want to visit the help pages via typing the command `help(stylo)`. Also, you should

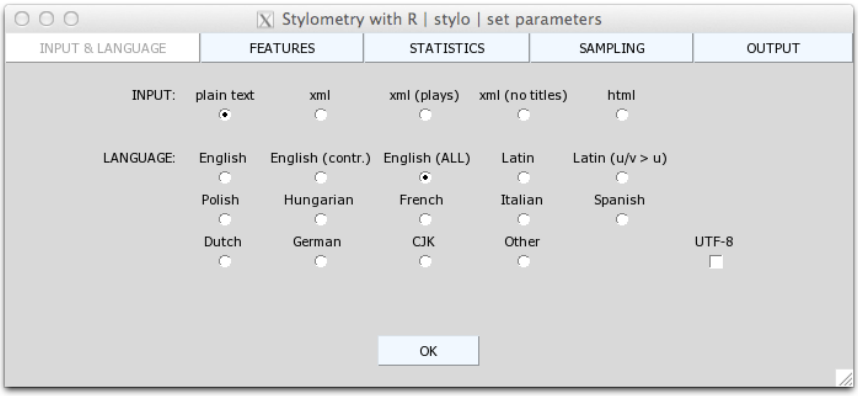

Figure 3: `stylo()` launched in its default mode: first tab on GUI

be familiar with additional options that can, or rather should, be passed as arguments; they are listed on the margins of this document.

Whenever you use the GUI, each successful execution or "run" of the script will generate a `stylo_config.txt` file (saved in your working folder) which you can review (for instance, should you have forgotten the parameters you used in your last experiment). The parameter settings specified in this file will be retrieved at each subsequent run of the script, so that the user won't have to *re*specify their favorite settings every time. Please note that when you hover your cursor over the labels of each of the entries in the GUI, tool tips will appear that will help you understand the GUI. In the following sections we will discuss each of the different tabs in the `stylo()` GUI.

No matter if you decide using GUI or not, you can pass additional arguments from command-line. If the graphic mode is on, these "new" values will appear in the GUI and thus they will be still modifiable. Some examples include:

```
stylo(mfw.min=300, mfw.max=300, analyzed.features="c", ngram.size=3)
stylo(gui=FALSE, analysis.type="MDS", write.png.file=TRUE)
stylo(mfw.min=100, mfw.max=1000, mfw.incr=100, analysis.type="BCT")
```

4.3.1 Input

This is where you specify the format of your corpus (see 4.1 above for more details about corpus preparation, and mind possible encoding issues). The available choices are: `corpus.format=`

- plain text: plain text files. `"plain"`
- xml: XML files; this option will remove all tags and TEI headers. `"xml"`
- xml (plays): XML files of plays; with this option, all tags, TEI headers, and speakers' names between `<speaker>...</speaker>` tags are removed. `"xml.drama"`
- xml (no titles): XML contents only: all tags, TEI headers, and chapter/section (sub)titles between `<head>...</head>` tags are removed. `"xml.notitles"`
- html: the option will attempt to remove HTML headers, menus, links and other tags. `"html"`

- UTF-8: if you use Linux or Mac, this option is immaterial; however, if your operating system is Windows, then you need to set it depending whether your dataset is encoded in Unicode (then check the option), or in ANSI (then leave it unchecked). `encoding= "UTF-8" "native.enc"`

4.3.2 Language

This setting makes sure that pronoun deletion (see below) works correctly. If you decide not to remove pronouns from your corpus (which is known to improve authorship attribution in some languages), this setting is immaterial (unless you are using English; see immediately below). `corpus.lang=`

- English: this setting makes sure that contractions (such as "don't") are *not* treated as single words (thus "don't" is understood as two separate items, "don" and "t"), and that compound words (such as "topsy-turvy") are *not* treated as one word (thus "topsy-turvy" becomes "topsy" and "turvy"). `"English"`
- English (contr.): this setting makes sure that contractions (such as "don't") *are* treated as single words (thus "don't" is understood as "don^t" and counted separately), but compound words (such as "topsy-turvy") are still *not* treated as one word (thus "topsy-turvy" becomes "topsy" and "turvy"). `"English.contr"`
- English (ALL): this setting makes sure that contractions (such as "don't") *are* treated as single words (thus "don't" is understood as "don^t" and counted separately), and that compound words (such as "topsy-turvy") *are* treated as one word (thus "topsy-turvy" becomes "topsy^turvy"). `"English.all"`
- Latin: this setting makes sure that "v" and "u" are treated as discrete character signs in Latin texts. `"Latin"`
- Latin.corr: since some editions do not distinguish between "v" and "u", this option provides a consistent conversion of both characters to "u" in each text. `"Latin.corr"`
- CJK: Chinese, Japanese and Korean scripts, provided that the input data is encoded in Unicode. `"CJK"`
- Other: non-Latin scripts: Hebrew, Arabic, Cyryllic, Coptic, Greek, Georgian, Latin phonetic, so far. Make sure your input data is in Unicode! `"Other"`

Please do note that for all other languages, apostrophes do *not* join words and compound (hyphenated) words are split. This is not the ideal solution and will be addressed as soon as we get to it.

4.3.3 Features

In many established approaches to stylometry, the (relative) frequencies of the most frequent words (MFW) in a corpus are used as the basis for multidimensional analyses. It has been argued, however, that other features are also worth considering, especially word and/or character n-grams. The general idea behind such n-grams is to combine a string of individual items into a partially overlapping, consecutive sequences of n of these individual items. Given a sample sentence "This is a simple example", the character 2-grams ("bigrams") are as follows: "th", "hi", "is", "s ", " i", "is", "s ", " a", "a ", " s", "si", "im", "mp", etc. The same sentence split into bigrams of words reads "this

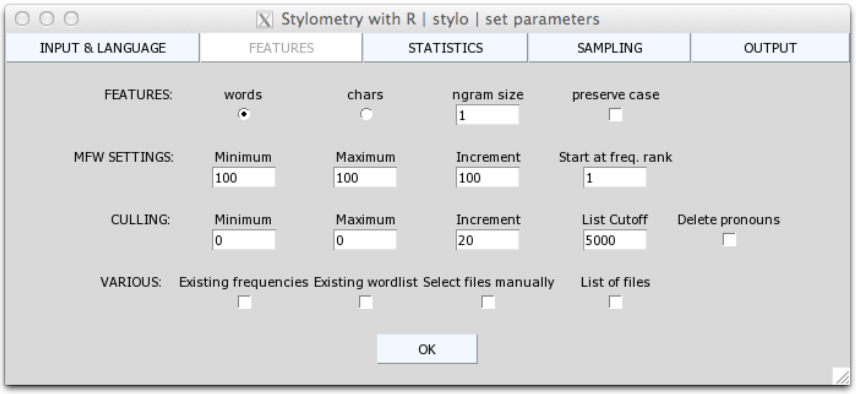

Figure 4: `stylo()` GUI, the second tab: Features, MFW Settings, Culling, Various

is", "is a", "a simple", "simple example". It has been heavily debated in the secondary literature whether the use of n-grams really increases the accuracy of stylometric tests (Hoover, 2002, 2003, 2012; Koppel et al., 2009; Stamatatos, 2009; Eder, 2011; Alexis et al., 2014). However, it has been shown (Eder, 2013) that character n-grams are impressively robust when one deals with a "dirty" corpus (one with a high number of misspelled characters, or one with bad OCR). The ideal combination of parameters in this section is another bone of contention between scholars; in fact, Eder and Rybicki (2013) maintain that this differs not only from language to language but also from one collection of text to another.

- words: words are used as the unit. Naturally, the higher the n you specify, the less repetitive your n-grams there will be, and this means poor statistics (data sparseness). `analyzed.features= "w"`
- characters: characters are used as the unit. `"c"`
- n-gram size: this is where you can specify the value of n for your n-grams. Certainly, setting this option to 1 makes sure that individual words/chars will be used instead of higher-order n-grams.; of course, single-letter counts do not seem like a good idea. `ngram.size= <integer>`
- preserve case: normally, all the words from the input texts are turned into lowercase, no matter if they are proper nouns or not – e.g. the sentence *The family of Dashwood had long been settled in sussex* will be turned into *the family of dashwood had long been settled in sussex*. In some situations, however, you might be interested in preserving the case. That's the option to do it. `preserve.case= TRUE|FALSE`
- select files manually: normally, the script performs the analysis on *all* files in your `corpus` subfolder. If this option is checked, a dialogue window will appear enabling the user the make a selection of input files from the subfolder. Obviously, you can achieve the same results by simply removing the unwanted texts from the `corpus` subfolder. Again, note that this function will expect you to select at least two different input files. `[TBD]`

4.3.4 MFW settings

This is where you specify the size of the most-frequent-word list that will be used for your analysis. Actually, the name is slightly misleading, since you are not at the mercy of

"most frequent *words*" only. You can use most frequent word pairs (bigrams), character sequences, etc. We keep the name "MFW" because... Well, we don't really remember why we keep it; probably, there was no-one around to propose a better solution.

- Minimum: this setting determines how many words (or features) from the top of the frequency list for the entire corpus will be used in your analysis in the first (and possibly, only) run of the function. With a value of 100 for this parameter, your analysis will be conducted on the 100 most frequent words (features) in the entire corpus. `mfw.min= <integer>`
- Maximum: this setting determines how many words from the top of the word frequency list for the entire corpus will be used in your analysis in the last (and possibly, only) run of the function. Thus, a setting of 1000 results in your (final) experiment being conducted on 1000 most frequent words in the entire corpus. (This parameter setting is especially important when working with the bootstrap consensus trees in `stylo()`, a procedure which involves running several analyses in a row. See immediately below under "Increment"). `mfw.max= <integer>`
- Increment: this setting defines the value by which the value of Minimum will be increased at each subsequent run of your analysis until it reaches the Maximum value. Thus, a setting of 200 (at a Minimum of 100 and a Maximum of 1000) provides for an analysis based on 100, 300, 500, 700 and 900 most frequent words. (As above, this parameter setting is especially important when working with the bootstrap consensus trees in `stylo()`, a procedure which involving running several analyses in a row). `mfw.incr= <integer>`
- Start at freq. rank: sometimes you might want to skip the very top of the frequency list. With this parameter, you can specify how many words from the top of the overall frequency rank list should be skipped. Normally, however, users will want to set this at 1. `start.at= <integer>`

N.B. For all statistical procedures (see 4.3.6 below) except the Consensus Tree, it is advisable to set Minimum and Maximum to the same value (this makes the Increment setting immaterial), unless you want to produce a large series of cluster analysis, multidimensional scaling or principal components analysis graphs in a row, for instance to observe how/if the results change for various lengths of the MFW list.

4.3.5 Culling

"Culling" refers to the automatic manipulation of the wordlist (proposed by Hoover 2004a, 2004b). The culling values specify the degree to which words that do not appear in all the texts of your corpus will be removed. Thus, a culling value of 20 indicates that words that appear in at least 20% of the texts in the corpus will be considered in the analysis. A culling setting of 0 means that no words will be removed; a culling setting of 100 means that only those words will be used in the analysis that appear in *all* texts of your corpus at least once.

- Minimum: this setting specifies the first (and possibly, only) culling setting in your analysis (cf. the minimum MFW setting). `culling.min= <integer>`
- Maximum: this setting specifies the last (and possibly, only) culling setting in your analysis (cf. the maximum MFW setting). (This parameter setting is espe- `culling.max= <integer>`

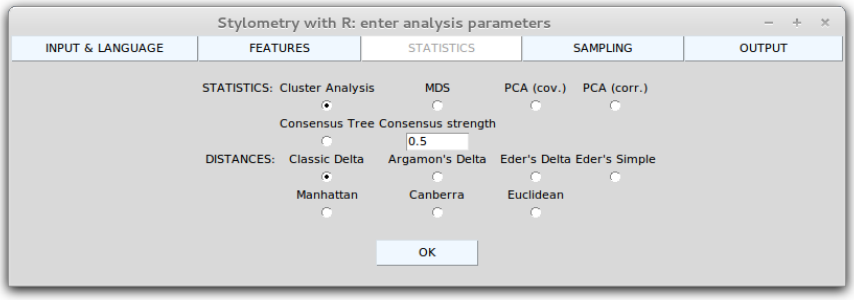

Figure 5: `stylo()` GUI, the third tab: Statistics, Distances

cially important when working with the bootstrap consensus trees in `stylo()`, a procedure which involves running several analyses in a row).

- Increment: this defines the increment by which the value of Minimum will be increased at each subsequent run of your analysis until it reaches the Maximum value. Thus a setting of 20 (at a Minimum of 0 and a Maximum of 100) provides for an analysis using culling settings of 0, 20, 30, 60, 80 and 100. (This parameter setting is especially important when working with the bootstrap consensus trees in `stylo()`, a procedure which involves running several analyses in a row). `culling.incr= <integer>`

- List cutoff: Usually, it is recommended to cut off the tail of the overall wordlist; if you do not want to cut the list and analyze vectors of thousands of words at once, then the variable may be set to an absurdly big number (although this can be computationally demanding for your machine). This setting is independent from the culling procedure. `mfw.list.cutoff= <integer>`

- Delete pronouns: (this setting too is independent of the culling procedure). If this option is checked, make sure you have selected the correct language for your corpus (see 4.3.2 above). This will select a list of pronouns for that language inside the script. Advanced users can use this part of the tool to remove any words they want. So far, we have pronoun lists for English, Dutch, Polish, Latin, French, German, Spanish, Italian, and Hungarian. `delete.pronouns= TRUE|FALSE` `corpus.lang= "English" "Dutch" ...`

N.B. As had been mentioned above, for all statistical procedures (see 4.3.6 below) except consensus trees, it is advisable to set Minimum and Maximum to the same value (this makes the Increment setting immaterial), unless you want to produce a large series of cluster analysis, multidimensional scaling or principal components analysis graphs etc. in a row.

4.3.6 Statistics

This is the very last moment to emphasize one important thing: the function `stylo()` provides a bunch of *unsupervised* methods used in stylometry, such as principal components analysis, multidimensional scaling or cluster analysis. The results are represented either on a scatterplot, or a tree-like diagram (dendrogram); the last stage of the analysis involves a human interpretation of the generated plots. The results obtained using these techniques "speak for themselves", which gives a practitioner an opportunity to notice with the naked eye any peculiarities or unexpected behavior in the analyzed corpus. Also, given a tree-like graphical representation of similarities between particular samples, one `analysis.type=`

can easily interpret the results in terms of finding out which group of texts a disputable sample belongs to. On the other hand, however, these methods cannot be *validated* in terms of an automatic verification of a given method's reliability. Thus, if you feel you'd better use one of *machine-learning* techiniques, refer to the funcion `classify()`, below.

- Cluster Analysis: Performs cluster analysis and produces a dendrogram, or a graph showing hierarchical clustering of analyzed texts. This option makes sense if there is only a single iteration (or just a few). This is achieved by setting the MFW Minimum and Maximum to equal values, and doing the same for Culling Minimum and Maximum. `"CA"`
- MDS: Multidimensional Scaling. This option makes sense if there is only a single iteration (or just a few). This is achieved by setting the MFW Minimum and Maximum to equal values, and doing the same for Culling Minimum and Maximum. `"MDS"`
- PCA (cov.): Principal Component Analysis using a covariance matrix. This option makes sense if there is only a single iteration (or just a few). This is achieved by setting the MFW Minimum and Maximum to equal values, and doing the same for Culling Minimum and Maximum. `"PCV"`
- PCA (corr.): Principal Component Analysis using a correlation matrix (and this is possibly the more reliable option of the two, at least for English). This option makes sense if there is only a single iteration (or just a few). This is achieved by setting the MFW Minimum and Maximum to equal values, and doing the same for Culling Minimum and Maximum. `"PCR"`
- Consensus Tree: this option will output a statistically justified "compromise" between a number of virtual cluster analyses results for a variety of MFW and Culling parameter values. `"BCT"`
- Consensus strength: For Consensus Tree graphs, direct linkages between two texts are made if the same link is made in a proportion of the underlying virtual cluster analyses. The default setting of 0.5 means that such a linkage is made if it appears in at least 50% of the cluster analyses. Legal values are $0.4 - 1$. This setting is immaterial for any other Statistics settings. `consensus.strength= <integer>`

4.3.7 Distances

This is where the user can choose the statistical procedure used to analyze the distances (i.e. the similarities and differences) between the frequency patterns of individual texts in your corpus. Although this choice is far from trivial, some of the following measures seem to be more suitable for linguistic purposes than others. On theoretical grounds, Euclidean Distance and Manhattan Distance should be avoided in stylometry based on word frequencies (unless the frequencies are normalized; see: Delta). Canberra Distance is quite troublesome but effective e.g. for Latin; it is very sensitive to rare vocabulary, and thus might be a good choice for inflected languages, with sparse frequencies (it should be combined with careful culling settings and a limited number of MFWs taken into analysis). For English, usually Classic Delta is a good choice: mathematically speaking (Argamon, 2008), it is simply Manhattan distance applied to normalized (z-scored) word frequencies. A theoretical explanation of the measures implemented in this function is pending. The available distance measures are as follows: `distance.measure=`

- Euclidean Distance: basic and the most "natural". It is an obvious choice when your variables are similarly distributed. However, since word distributions are not similar at any rate (e.g. compare the huge difference between the frequencies of "the" and "dactyloscopy"), this distance measure is not appropriate to testing vectors of dozens of most frequent words. Or, to be precise, it could be used to assess less frequent (content) words. According to Zipf's law, these words are distributed more or less similarly in a corpus since, by being less common than function words, they appear in the flattened sections of a Zipf curve. `"dist.euclidean"`

$$\delta_{(AB)} = \sqrt{\sum_{i=1}^{n} |(A_i)^2 - (B_i)^2|}$$

where:
n = the number of MFWs (most frequent words),
A, B = texts being compared,
A_i = the frequency of a given word i in the text A,
B_i = the frequency of a given word i in the text B.

- Manhattan Distance: obvious and well documented. It shares the pros and cons of Euclidean Distance. `"dist.manhattan"`

$$\delta_{(AB)} = \sum_{i=1}^{n} |A_i - B_i|$$

- Classic Delta as introduced by Burrows (2002). Since this measure relies on z-scores – i.e. normalized word frequencies – it is dependent on the number of texts analyzed and on a balance between these texts: if a corpus contains, say, a large nuber of plays by Lope de Vega and only one play by Calderón de la Barca, the final results might by biased. `"dist.delta"`

$$\Delta_{(AB)} = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{A_i - \mu_i}{\sigma_i} - \frac{B_i - \mu_i}{\sigma_i} \right|$$

where:
n = the number of MFWs (most frequent words or other features),
A, B = texts being compared,
A_i = the frequency of a given feature i in the text A,
B_i = the frequency of a given feature i in the text B,
μ_i = mean frequency of a given feature in the corpus,
σ_i = standard deviation of frequencies of a given feature.

Argamon (2008) showed that the above formula can be simplified algebraically:

$$\Delta_{(AB)} = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{A_i - B_i}{\sigma_i} \right|$$

- Argamon's Linear Delta, or Euclidean distance applied to normalized (z-scored) word frequencies (Argamon, 2008). The distance is sensitive to the number of texts in a corpus. `"dist.argamon"`

$$\Delta_{(AB)} = \frac{1}{n} \sum_{i=1}^{n} \sqrt{\left| \frac{(A_i)^2 - (B_i)^2}{\sigma_i} \right|}$$

- Eder's Delta: it is a modification of standard Burrows's distance; it slightly increases the weights of frequent words and rescales less frequent ones in order to suppress discriminative strength of some random unfrequent words. The distance was meant to be used with highly inflected languages. It is sensitive to the number of texts in a corpus. `"dist.eder"`

$$\Delta_{(AB)} = \frac{1}{n}\sum_{i=1}^{n}\left(\left|\frac{A_i - B_i}{\sigma_i}\right| \times \frac{n - n_i + 1}{n}\right)$$

where:
n_i = the position of a given feature on a frequency list (i.e. its rank).

- Eder's Simple: a type of normalization as simple as can be (independent on the size of the corpus), intended to convert the implications of Zipf's law. The normalization used in this distance is so obvious and so widely-spread in exact sciences that naming it "Eder's Simple Distance" is an abuse, so to speak. `"dist.simple"`

$$\delta_{(AB)} = \sum_{i=1}^{n}\left|\sqrt{A_i} - \sqrt{B_i}\right|$$

- Canberra Distance: sometimes amazingly good. It is very sensitive to differences in rare vocabulary usage among authors. On the other hand, this can be a disadvantage, since sensitiveness to minute differences in word occurrences also means significant sensitiveness to noise. Last but not least, Canberra Distance is very sensitive to the number of words (features) analyzed. `"dist.canberra"`

$$\delta_{(AB)} = \sum_{i=1}^{n}\frac{|A_i - B_i|}{|A_i| + |B_i|}$$

- Cosine Distance: a classical measure, introduced to this package in the version 6.3.: `"dist.cosine"`

$$\delta_{(AB)} = 1 - \frac{A \cdot B}{\|A\|\|B\|} = 1 - \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} A_i^2}}$$

- It is also possible to use any custom distance measure. This option is discussed below, in the section 9.4.

4.3.8 Sampling

When the default setting of "No sampling" is checked, each of the texts in its entirety is treated as a single sample. The second option, that of "Normal sampling", performs the analysis on equal-sized consecutive sections of each text, and the size is determined by the setting immediately below. Eder (2015) suggests that even better attribution results can be achieved with "Random sampling", where samples are made up of words each randomly selected from anywhere in the text ("bag of words"); here, too, the sample size must be set below. `sampling=` `"no.sampling"` `"normal.sampling"` `"random.sampling"` `sample.size=` `<integer>`

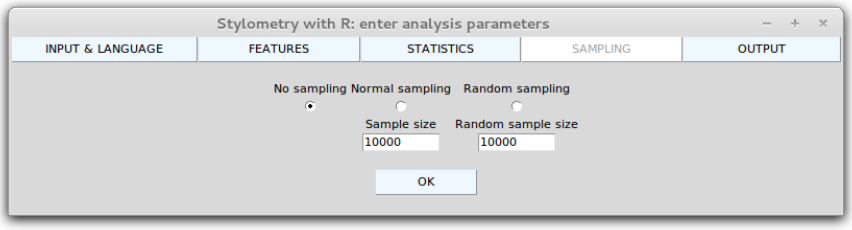

Figure 6: `stylo()` GUI, the fourth tab: Sampling

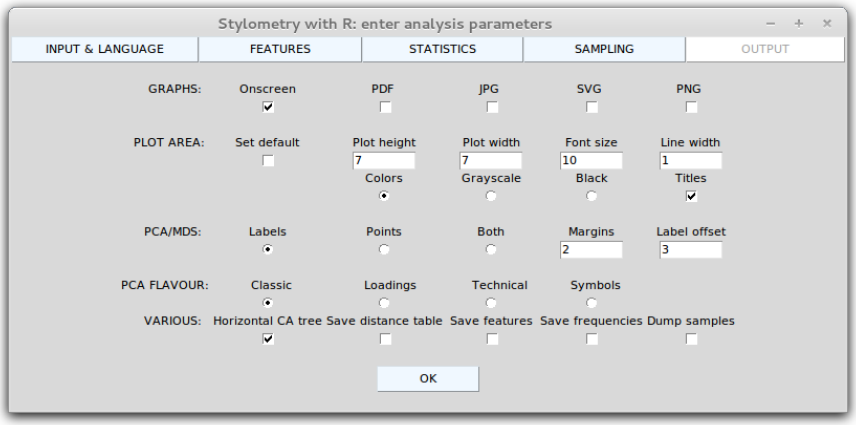

Figure 7: `stylo()` GUI, the fourth tab: Output

4.3.9 Graphs

Do you want to display the graph on the screen? Do you want to write the graph directly to a graphics file? Which format? If you've been thinking about any plots, these are the options to fiddle with. You can display the graph on the screen *and* write to a file (the latter will be done with much better quality). The produced files are saved in your working directory. As has already been mentioned, the name of your working directory is used both as the title on top of the graph (see 4.3.10 below) and as the name of the graph files; some important parameter settings (for MFW list size, culling, pronoun deletion, statistical method) are also placed on the graph and in the name of the file. However, remember that if you perform another analysis with the same parameters on a slightly modified corpus, this will overwrite the earlier graph.

- Onscreen: check this if you want to display the graph on the screen. `display.on.screen= TRUE|FALSE`
- PDF: check this to obtain a PDF file with your graph. `write.pdf.file= TRUE|FALSE`
- JPG: check this to obtain your graph in JPEG format. `write.jpg.file= TRUE|FALSE`
- SVG: check this to produce a SVG vector file (XML-based scalable image format). Useful if you want to embed your plot in HTML code, or to edit it further with, say, Inkscape. `write.svg.file= TRUE|FALSE`
- PNG: check this to obtain your graph in PNG format (probably the safest option if you want publication-ready resolution). `write.png.file= TRUE|FALSE`

4.3.10 Plot area

Further graphic options. Here you can specify the dimensions of the plot area (expressed in inches, yes, even though the makers of the package are all honest Continental Europeans and usually think in centimeters), font size, thickness of the lines used to plot the graphs, etc. Since it is usually hard to remember all the values, an additional option is provided to reset the picture options – if this is checked, the remaining options will be overwritten and the defaults restored.

- Set default: this is the above-mentioned "amnesia button". If you fiddle too much with different graphic settings and forget, say, the default size of scatterplots, this option is exactly what you need. (Remember, however, that some computers hate the "amnesia buttons", HAL from *2001: A Space Oddyssey* being the most convincing caveat). `plot.options.reset= TRUE|FALSE`
- Plot height: self-evident. The valid units are inches. `plot.custom.height=`
- Plot width: self-evident. The valid units are inches. `plot.custom.width=`
- Font size: self-evident. The unit is "points" (a detailed explanation what type of "points" it is and how they are related to "typographic points", "Didot points", "Postscript points" etc. can be skipped here; the only thing worth noting is that they are the same "points" as in popular office programs, like MS Office). `plot.font.size= <integer>`
- Line width: if you plan to re-scale your plot, you might also want to increase the thickness of the lines in your graph. This value is expressed in R generic units (1 is default). `plot.line.thickness= <integer>`
- Colors: when this option is checked, the script will automatically assign the same colors to texts with the same first segment of their file names (the first string ending in "_"). `colors.on.graphs= "colors"`
- Greyscale: select this option to have automatic color coding (in greyscale). The file names convention: see above. While the graphs become less pretty than when colored, this might be your method of choice if the journal you're planning to publish in makes you pay for color illustrations their weight (or, rather, resolution) in gold. `"greyscale"`
- Black: select to have a black & white graph. No file naming convention is required with this option. `"black"`
- Titles: if this is checked, the graphs will contain a main (top) title (based on the name of your folder and your choice of the statistics option) and a subtitle (bottom) showing the distance metric which you selected, as well as the MFW, Culling, Feature and Consensus settings. If you don't want to have the title decided automatically, you need to pass an argument `custom.graph.title`. `titles.on.graphs= TRUE|FALSE` `custom.graph.title= "Any text"`

4.3.11 PCA/MDS

More graphic options. The following settings, however, apply to scatterplots only, i.e. to the plots produced by principal component analysis and multidimensional scaling.

- Labels: to identify samples on MDS/PCA plots using their labels, or names, as specified in filenames (see 4.1 above). `text.id.on.graphs= "labels"`

- Points: instead of labels, use points to show the positions of the particular samples. This option might be helpful if you want to represent a large number of samples on one plot. `"points"`
- Both: in some cases, you might be interested to get *precise* positions of samples on a graph *and* to keep the samples' labels on. This option pinpoints the exact locations of the samples using points, and slightly offsets the labels (see the option 'Label offset' immediately below). `"both"`
- Margins: if you use particularly long samples' names (labels) and/or simply want to add some blank space on your plot, set custom margin size (in percentage of plot area). You can find out your best setting by trial and error. `add.to.margins=<integer>`
- Label offset: set custom offset between label and point (in percentage of plot area). `label.offset=<integer>`

4.3.12 PCA flavour

Even more graphic options. Should you want to use PCA, you can choose one of the four visualization flavors. `pca.visual.flavour=`

- Classic: original PCA visualization using (colored, if applicable) sample names or points. `"classic"`
- Loadings: display PCA feature (word etc.) loadings. This type of visualization is aimed to show the discriminative strength of particular features (e.g. words) across two first principal components. Some visual similarity to popular "word clouds" makes this approach attractive and comprehensive. `"loadings"`
- Technical: technical greyscale PCA visualization, showing feature loadings as well as a PC barplot. Potentially useful for greyscale printing in traditional publications. `"technical"`
- Symbols: select to display the samples in your PCA with a group symbol (instead of their entire name). Potentially useful when dealing with lots of samples. `"symbols"`

4.3.13 Various

- Horizontal CA tree: select to have your cluster analysis graph oriented horizontally. Probably the better option for clarity, especially if there are a lot of samples to fit on one dendrogram. `dendrogram.layout.horizontal=TRUE|FALSE`
- Save distance table: save final distance table(s) in separate text file(s). In most cases, you will not need to use this option. `save.distance.tables=TRUE|FALSE`
- Save features: save final feature (word, n-gram) list(s), e.g. the words actually used in the analysis. Use this option to have more control over the experiment: if you feel that your results are suspicious or too good to be true, you can open the generated file and check the features used. `save.analyzed.features=TRUE|FALSE`
- Save frequencies: this option gives you even more control: you can inspect frequencies of each word across the whole corpus. Switching this option saves frequency table(s) in separate text file(s). `save.analyzed.freqs=TRUE|FALSE`

- Dump samples: if you still feel that your experiment in not supervised enough, you migth be interested in a "post mortem" inspection of all the samples used: this option dumps the original samples (either whole texts, or chunks specified using the "Sample size" option above) to an external file. Be aware, however, that a corpus containig dozens of full-sized novels might produce a huge dump file.

`dump.samples=TRUE|FALSE`

5 classify()

This function performs a number of machine-learning algorighms of classification: Delta (Burrows, 2002), k-nearest neighbors, support vectors machines, naive Bayes, and nearest shrunken centroids (Jockers and Witten, 2010). Most of the options are derived from the above-mentioned `stylo()` function.

Unlike explanatory methods as supported by `stylo()`, this approach involves two stages of the analysis. In the first step, the traceable differences between samples produce a set of rules, or a classifier, for discriminating authorial "uniqueness" in style. The second step is of predictive nature – using the trained classifier, the machine assigns other text samples to the authorial classes established by the classifier; any disputed or anonymous samples will be assigned to one of the classes as well, provided that such a classification is usually based on probabilistic grounds.

The procedure described above relies on an organized corpus of texts. Namely, the clue is to divide all the available texts into two groups: primary (training) set and secondary (test) set. The first set, being a collection of texts written by known authors ("candidates"), serves as a sub-corpus for finding the best classifier, or discrimination rules, while the second set is a pool of texts of known authors, anonymous texts, disputed ones, and so on. The better the classifier, the more samples from the test set are attributed ("guessed") correctly and the more reliable the attribution of the disputed samples.

The function writes the resulting authorship (or similarity) candidates to a logfile (`results.txt`) in the current working directory.

5.1 Corpus preparation

Since machine-learning methods involve two sets of texts instead of one, you need to create two subdirectories within your working directory. You don't really need to name this directory in any special way – no graphs will be generated and thus no titles on graph will be used. However, the names of both subfolders are very important: the one containing training samples should be named `primary_set`, and the test set should be titled `secondary_set` (all file names are case sensitive!). In a usual authorship attibution study, the training set will contain at least one text by each candidate author, preferably one "representative" for his/her work, whatever that is (thus, for Goethe, something else than *Farbenlehre*; for Dickens, probably not *The Pickwick Papers*). In the test set, you usually put the anonymous or disputed samples you want to analyze, but in most cases you also include a number of known texts to test the robustness of a particular method. To keep the things simple: if you collect a number of texts written by women and men in the training set, you should also put some other text written by both groups (to provide "unseen" or "fresh" data) to the test set to see if they are correctly recognized. A sample corpus of English writers might be split into two subsets as follows:

```
ABronte_Agnes.xml
Austen_Emma.xml
```

```
Conrad_Nostromo.xml
...
ABronte_Tenant.xml
Austen_Pride.xml
Austen_Northanger.xml
Conrad_Lord.xml
Dickens_Pickwick.xml
Anonymous_Strange-novel.xml
...
```

The number of samples to be kept in these subfolders depends on the method you are going to use. For Delta, support vector machines, and k-NN, the minimal number of texts per class is 1 (as in the above example). Both naive Bayes and nearest shrunken centroids require at least two samples per class to be put into the training set. (Certainly, if you are short of texts, you can cheat: by putting a given sample twice into the training set under two different names, e.g. `Swift_Tub-1.txt`, `Swift_Tub-2.txt`). However, it is a commonly accepted fact that the more training data the better – whenever you have enough texts available, put a good portion of them into the training set.

Another aspect of the above question is the nature of your stylometric test. If you want to assess authorship, then a couple of texts per "candidate" should be fine, but if you want to find a rule of gender differentiation, then you probably should collect quite a lot of textual data written by men and women. And if you believe it is possible to separate left-handed and right-handed writers, you need to take tons of training texts, and even then your experiment will lack some methodological rigour (it is sometimes called "the risk of modeling a noise").

5.2 Calling the function

The function is evoked by the command `classify()`.

5.3 Options

Most of the options are derived from the `stylo()` function. Refer to the section 4.3 for further details.

5.3.1 Options inherited from `stylo()`

Input (4.3.1), Language (4.3.2), Features (4.3.3), MFW settings (4.3.4), Culling (4.3.5), Delta Distances (4.3.7), Sampling (4.3.8), Output: various (4.3.13).

5.3.2 Statistics

- Delta: Burrows's Delta. `classification.method="delta"`
- k-NN: k-nearest neighbor classification. `"knn"`
- SVM: support vector machines. `"svm"`
- NaiveBayes: naive Bayes classification. To use this method, you should have at least two texts of each class (author, genre, etc.) in the primary (training) set. `"naivebayes"`
- NSC: nearest shrunken centroid classification. To use this method, you should have at least two texts of each class (author, genre, etc.) in the primary (training) set. `"nsc"`

5.3.3 General

- ALL culling: the culling procedure (cf. 4.3.5) is based on the percentage of samples containing a given word. To compute this ratio, one might want to use the texts from the first set only, or from both sets. `culling.of.all.samples= "TRUE|FALSE"`
- ALL wordlists: the both tables of frequencies are build using the pre-prepared word list of the whole primary set (default). Alternatively, one might want to prepare this list of both sets by activating this option. `reference.wordlist.of.all.samples= "TRUE|FALSE"`
- ALL z-scores:: how the z-scores should be calculated. If the variable is set to FALSE, the z-scores are relying on the primary set only (this should be better in most cases; after all, this is the classical solution used by Burrows and Hoover). Otherwise, the scaling is based on all the values in the primary and the secondary sets. (This option is applicable to Delta only). `z.scores.of.all.samples= "TRUE|FALSE"`

5.3.4 SVM options

Support vector machines classification settings: refer to `help(svm)` from `library(e1071)` for details.

- Linear: linear kernel of SVM; probably the best choice in stylometry, since the number of variables (e.g. MFWs) is many times bigger than the number of classes. `svm.kernel= "linear"`
- Polynomial: polynomial kernel of SVM. `"polymonial"`
- Radial: radial kernel of SVM. `"radial"`
- Degree: parameter needed for kernel of type "polynomial" (default: 3). `svm.degree= <integer>`
- Coef0: parameter needed for kernel of type "polynomial" (default: 0). `svm.coef0= <integer>`
- Cost: cost of constraints violation (default: 1); it is the C-constant of the regularization term in the Lagrange formulation. `svm.cost= <integer>`

5.3.5 k-NN options

- k value: the k value in k-Nearest Neighbour algorithm, or number of neighbours to be considered. Certainly, the bigger the number the better the performance, but, on the other hand, to set this value to 3, you need to have at least three samples per class in the training set. If you keep just one text per class in the training set, the performance is *ex definitione* suboptimal. `k.value= <integer>`
- l value: minimum vote for definite decision, otherwise "doubt". (More precisely, less than $k - l$ dissenting votes are allowed, even if k is increased by ties). `l.value= <integer>`

5.3.6 Output: general

- Misclassifications: here you can specify whether you want to list the misclassified samples into the log file. Certainly, in most cases you will want to have them listed. However, if you plan to perform a multi-iterated large-scale experiment to test performance of a given method, you will probably prefer to switch off all that verbosity. `final.ranking.of.candidates= TRUE|FALSE`

- Count good guesses: report the number of correct guesses for each iteration (written to the log file). To say the truth, this option is a bit obsolete, since using `classify()` you are almost always interested in the number of correctly classified samples.

`how.many.correct.attributions= TRUE|FALSE`

- No. of candidates: final ranking of candidates is directed to a file. You may specify the number of final ranking candidates to be displayed (at least 1). This option works for Delta only.

`number.of.candidates= <integer>`

6 rolling.delta(), rolling.classify()

The procedure "Rolling Delta", supported by the function `rolling.delta()` is reminiscent of a number of earlier applications of the metric (e.g. van Dalen-Oskam and van Zundert, 2007; Kestemont, 2010; Burrows, 2010). The general goal is to use the Delta metric to reliably visualize stylistic shifts in texts, for instance in order to study the stylistic evolution in texts, to detect plagiarism or to pinpoint authorial takeovers in the case of collaborative authorship.

The first step involves a "windowing" procedure in which each reference text is segmented into consecutive, equal-sized samples or "windows" (`window.size` parameter). The samples are allowed to partially overlap (`step.size` parameter). If we specify a `window.size` of 5,000 and a `step.size` of 100, for example, the first sample of a text contains words 1–5,000, the second 101–5,100, the third sample 201–5,200, and so forth (see 6.3.2 below). Like Delta, our procedure uses the relative frequencies of a (preferably small) set of n words which were most frequent in the entire collection of reference texts. Subsequently, we compute a representative centroid for each reference text that consists of the mean relative frequency for each of the n words in the windows extracted from the text, as well as the standard deviation.

We then proceed to the analysis of the test text. We divide it into windows too and iteratively compute the difference in style (the Delta) between each test window and each reference centroid. In order to calculate the Delta with the n most frequent words we employ the following formula. Let C be an author's centroid and let W be the window we wish to compare it to. For each of the n words, we calculate the absolute difference between its average frequency in C and its frequency in W. Next, we weigh this difference using each word's standard deviation in the centroid. (Words whose frequencies display significant fluctuation in a reference text's windows are thus assigned a lower weight.) The final Delta is the summation of these n weighted differences.

$$\Delta(C, W) = \sum_{i=1}^{n} \frac{1}{\sigma_i C} |\mu_i(C) - f_i(W)|$$

After "rolling" through the test text we can plot the resulting series of Deltas for each reference text in a graph. The relatively lower the Deltas for a given reference text, the relatively more similar the style in the test windows – and vice versa (cf. Hoover, 2004b: 471). If the curve for a text would show a sudden drop at a given position, this could be indicative of a stylistic change in the text (which might, for instance, be caused by one author taking over from another. One can use vertical lines in the plot to mark the position of certain events in the test text as an aid in interpreting the graph (e.g. chapter beginnings).

6.1 Corpus preparation

You will need two subfolders in your working directory: `primary_set` should contain the test texts: the individual writings by each of the authors who collaborated on the test text – the latter goes into the `secondary_set` subfolder (once again, the names are case-sensitive). In the pilot study for this method (Rybicki et al., 2014), the primary set was composed of individual writings by Joseph Conrad and individual writings by Ford Madox Ford, such as, respectively, *The Heart of Darkness* and *The Good Soldier*; the secondary set only contained a single text at a time: one of the texts written in collaboration by the two writers, such as *The Inheritors*. To study another Conrad/Madox collaboration, *The Inheritors* had to be removed from the secondary set and replaced by, say, *Romance*.

6.2 Calling the function

The function is evoked by the command `rolling.delta()`.

6.3 Options

While many of the options are derived from the main `stylo()` function – especially in the "Input & Language", "Statistics", and "Output" tabs, some differences must be emphasized here.

6.3.1 Features

Contrarily to this section in `stylo()`, MFW settings only use a single value (Maximum) since only one analysis is performed, and the same is true of the Culling value.

6.3.2 Sampling

The "Slice length" parameter sets the size of the text "window" or of consecutive samples cut out one by one from the test text. "Stepsize" controls the size of the overlap between two consecutive windows. For the default settings, a slice length of 5,000 and a stepsize of 1,000 takes the first 5,000 words from the beginning of the test text as the first sample, the section from the 1001th word in the text to its 6,000th word, and so forth. Beware of very small stepsize values: we have not yet seen a computer that would not hang R at a setting of 1!

6.3.3 Colors

An additional tab has been added to control the colors of the curves for each training set text. Two courses of action are advisable here. If you only want to differentiate the texts by author, you might want to set a single color from the pull-down fields for that author's texts, and another for the texts by the second writer. "Color 1" sets the color for the text that comes first alphabetically in the ordinary listing of filenames, "Color 2" colors the second text, etc. Thus, to use the above examples, all texts by Conrad would precede those by Ford, and `conrad_heart.txt` (for *The Heart of Darkness*) would precede `conrad_jim.txt` (for *Lord Jim*), etc. The other alternative is to use different shades of the same basic color for each writer so that the similarity between the particular sets can be more visible. The colours in the pull-down fields can be replaced with other text names of colors in R's pallette; you can get a listing of these by invoking R's `colors()` command.

7 oppose()

It performs a contrastive analysis between two given sets of texts, using Burrows's Zeta (2007) in its different flavors, including Craig's extensions (Craig and Kinney, 2009). Also, the Whitney-Wilcoxon procedure as introduced by Kilgariff (2001) is available. The function generates a vector of words significantly *preferred* by a tested author, and another vector containing the words significantly *avoided*.

7.1 Corpus preparation

Suppose you want to find out the characteristic words of men and women, and then to see which of the anonymous books in a corpus might have been written by men, and which by women. In order to do this, you put all the women into the `primary_set` subfolder of your working directory, and all the men into the `secondary_set` folder, and then you place the anonymous texts in the `test_set` subfolder (the `test_set` folder is optional; the script *will* run if it is not there or if you are not interested in the anonymous texts).

7.2 Calling the function

The function is evoked by the command `oppose()`.

7.3 Options

- Text slice length (in words), the default is 10,000 words. This parameter refers to the size of the samples into which each text is "sliced" in order to perform zeta.
- Text slice overlap (in words). The default, 0, means that the first sample will contain words 1 to 10,000 in the text, the second sample 10,001 to 20,000, etc. If you set it to 2000, the first sample will contain words 1 to 10,000 in the text, the second sample 8001 to 18,000, etc. Beware of low values: setting it to 1 will result in a huge number of samples and R might eventually crash.
- Rare occurrences threshold: the default, 2 prevents hapax legomena and dislegomena from appearing in the resulting zeta wordlist file; 1 gets rid of just hapax legomena, 10 makes sure only words that appear at least 11 times in the corpus are included in the list.
- Filter threshold (default 0.1) gets rid of word of weak discrimination strength (it's like p, the degree of statistical significance, in various standard statistical texts). The higher the number, the less words appear in the final wordlist. It does not normally exceed 0.5. To quote Maciej Eder: "if the 'craig.zeta' method is selected, you might probably want to filter out some words of weak discrimination strength. Provided that 2 means the strongest positive difference and 0 the strongest negative difference (Hoover, 2009), the values either just above or just below 1 are not significant and thus can be (or rather should be) omitted. If chisquare method was chosen, all the differences of p value below 0.05 were filtered out, in pure Zeta, there is no a priori solution. Threshold 0.5 would filter out a vast majority of words, threshold set to 1 would filter all the words in a corpus."
- Method: we have 3 zeta flavors so far: Craig's (as described by him and Hoover); Eder's (not documented yet, but basically derived from Canbera distance measure); chi-square (not documented yet).

- Output: self-evident, except that "Identify points" only works (if it *does* work) with output set to "Onscreen".

The script outputs two files: a list of `words_preferred.txt`, which are words significantly *preferred* by the primary author(s); and a list of `words_avoided.txt`, which are words significantly *avoided* by the primary author(s).

The graph plots the frequencies of both word categories, preferred in avoided, for each sample into which the texts have been sliced; normally, primary set samples (marked as circles; colors correspond to the authors of the texts from which they were taken) should appear in the top left corner of the plot (high words preferred frequencies, low words avoided frequencies), while the secondary set samples (marked as triangles) should gather in the bottom right (low words preferred frequencies, high words avoided frequencies). Samples from texts by authors in the test set are marked as crosses; if they overlap with either the primary or the secondary set samples, this shows the stylistic similarity. Also, a polygon marking the outside limit of the primary set samples, and another one for the secondary set, are drawn on the graph. Of course, you can now combine the words preferred and avoided files into a single `wordlist.txt` file and use the `stylo()` function for better discrimination between two groups of texts.

8 Options unavailable on GUI

8.1 Cluster analysis linkage

- linkage: algorithm for establishing clusters in a dendrogram; choose one of the following linkage methods: `"nj"`, `"ward"` (default), `"single"`, `"complete"`, `"average"`, `"mcquitty"`, `"median"`, `"centroid"`. `linkage=`

8.2 Network analysis support

The package "stylo" does not produce any networks *per se*, however, it does generate tables of edges/nodes (or, edges alone), using two Eder's algorightms of establishing connections between the nodes (Eder, forthcoming). The table can be loaded into Gephi (https://gephi.org). To get such a table, invoke the function `stylo()` with an argument `network=TRUE`, and optionally with some other arguments, as listed below. E.g.:

```
stylo(network=TRUE, network.type="undirected")
```

- network: an output file (or files) will be generated when this option is set TRUE, if this is set FALSE, the options immediately below are immaterial. (Default: FALSE). `network=` `TRUE|FALSE`

- network.tables: one of two flavors of output: either one table (edges), or two (edges and nodes). Choose "edges" (default), or "both", respectively. Using both tables instead of one allows you to edit the table of nodes in, say, Excel, in order to set some node attributes. When you use two tables, however, make sure you import edges to Gephi first; also, make sure you uncheck (in Gephi) the option for creating missing nodes. `network.tables=` `"edges"` `"both"`

- network.type: when "undirected" type of network is chosen (default), then the connections *from* and *to* are counted together (summed into one stronger connection). When `"directed"` network is chosen, then the incoming connections and the outcoming ones are counted separately. `network.type=` `"undirected"` `"directed"`

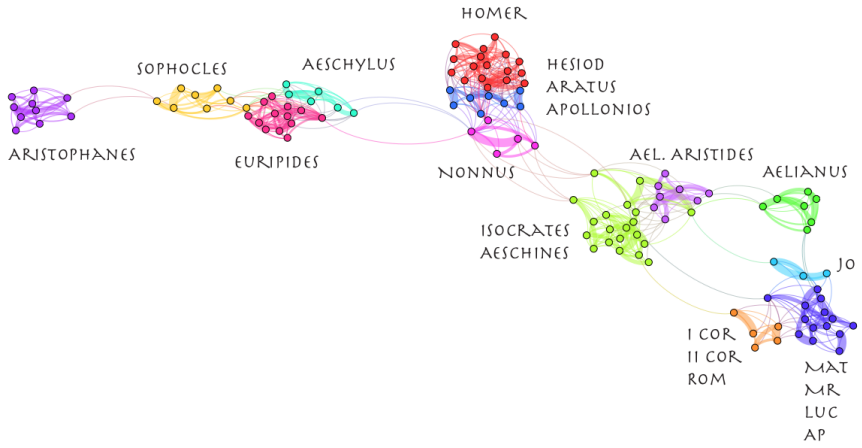

Figure 8: 124 Greek texts represented as connected nodes of a network

- linked.neighbors: if this value is set to 1, then a link between a given sample and its nearest neighbor is established; when it is set to 2, two neighbors are connected (the nearest neighbor and the firs runner-up), etc. Default value is 3, which means that the nearest neighbor and two runners-up are taken into consideration.

`linked.neighbors= <integer>`

- edge.weights: the connections' weights are always differentiated: the nearest neighbor has the strongest link, then comes the first runner-up, and so forth. The assigned weights might be `"linear"` $= 1, 2, 3, \ldots, n$; `"quadratic"` $= 1, 4, 9, ..., n^2$; or `"log"` (logarithmic) $= \log(1 + (1, 2, 3, ..., n))$.

`edge.weights= "linear" "quadratic" "log"`

Network analysis plots might be useful for visualizing textual relations in large datasets. Particular texts can be represented as nodes of a network, and their explicit relations as links between these nodes (Fig. 8). The procedure of linking is twofold (details: Eder, forthcoming). One of the involved algorithms computes the distances between analyzed texts, and establishes, for every single node, a strong connection to its nearest neighbor (i.e. the most similar text), and weaker connections to the 1st and the 2nd runner-up (i.e. two texts that get ranked immediately after the nearest neighbor). The second algorithm performs a large number of tests for similarity with different number of features to be analyzed (e.g. 100, 200, 300, ..., 1,000 MFWs). Finally, all the connections produced in particular "snapshots" are added, resulting in a consensus network.

8.3 Undocumented arguments

- `features`
- `frequencies`
- `training.frequencies`
- `test.frequencies`

- `parsed.corpus`
- `training.corpus`
- `test.corpus`
- `cv`
- `cv.folds`
- `encoding`
- `relative.frequencies`

9 Advanced topics

9.1 Batch mode

As mentioned somewhere in this document, it is possible to pass input data into `stylo()` and `classify()` from inside R, without relying on any external files. Also, the final results – apart from being plotted on screen and saved to the hard-drive – are accessible as R objects. One can imagine a very complex stylometric experiment computed remotely on a high-performance server without any file reading/writing involved. The next sections provide an outline of such a pipeline design.

The datasets that can be piped into `stylo()` and `classify()` from other R functions include: (1) pre-processed corpus, in terms of a list containing vectors of words (or other countable units); see `help(load.corpus.and.parse)` for further details, see also an example discussed in `help(stylo)`, (2) table of word frequencies: an R matrix or data frame with variables (words) formatted vertically as colums, and samples (texts) ordered horizontally as rows, (3) words (MFWs) or other features to be analyzed: an R vector containing the features as elements. In the case of `classify()`, two corpora and/or two frequency tables are piped instead. The following executable toy example, quoted after `help(classify)`, shows how the training and the test corpus should be passed in a pipelne:

```
# preparing a training corpus
txt1 = c("now", "i", "am", "alone", "o", "what", "a", "slave", "am", "i")
txt2 = c("what", "do", "you", "read", "my", "lord")
corpusTRAIN = list(txt1, txt2)
names(corpusTRAIN) = c("hamlet_sample1", "polonius_sample1")

# preparing a test corpus
txt4 = c("to", "be", "or", "not", "to", "be")
txt5 = c("though", "this", "be", "madness", "yet", "there", "is", "method")
txt6 = c("the", "rest", "is", "silence")
corpusTEST = list(txt4, txt5, txt6)
names(corpusTEST) = c("hamlet_sample2", "polonius_sample2",
         "uncertain_1")

# launching the classification
classify(training.corpus = setTRAIN, test.corpus = setTEST)
```

One can pass the features to be analyzed (e.g. MFWs) into `stylo()` or `classify` in a similar way. Consider the following example:

```
my.selection.of.function.words = c("the", "and", "of", "in", "if",
          "into", "within", "on", "upon", "since")
stylo(features = my.selection.of.function.words)
```

In any attempts to use existing tables of frequencies, one needs to remember that R in general, and the package `stylo` in particular, accepts tabular datasets with variables stored as *columns*. Certainly, this might be slightly confusing, since in other statistical programs the variables are usually stored in *rows*. If your dataset is formatted as follows:

	ABronte	Austen	CBronte	Conrad	Dickens	
	Agnes	*Emma*	*Jane*	*Lord*	*Bleak*	...
"the"	4.57	4.24	4.25	4.19	4.47	...
"to"	3.11	3.29	3.43	3.14	3.71	...
"and"	3.19	3	3.08	2.85	2.81	...
"of"	2.6	3	2.63	2.43	2.86	...
"I"	2.17	2.2	2.13	2.42	2.22	...
"a"	2.24	1.92	1.92	2.21	1.92	...
⋮	⋮	⋮	⋮	⋮	⋮	⋱

then you should do a tiny tweak before piping it further:

```
my.rotated.dataset = t(my.dataset)
stylo(frequencies = my.rotated.dataset)
```

For the sake of compatibility, the tables produced by the package `stylo` are always saved in the transposed format – do not be confused, then.

When it comes to the final results stored as R objects, you should remember that each function returns its value after evaluation. The main functions of the package `stylo` do not clutter your screen with thousands of values, but the results are there anyway. They are made invisible. To have an access to these values, pipe the function, say, `classify` to a variable:

```
my.results = classify()
```

The variable `my.results` is a list containing a number of elements, including tables of frequencies, classification results, and so forth. E.g. type `my.results$frequencies.training.set` to get one of the tables.

Now, since the functions accept R objects as input data, and at the same time their output is stored in R's memory as a list of R objects, it is possible to connect two functions "on the fly". Consider an experiment involving the function `oppose()`, aimed at extracting the most characteristic words for subcorpora A and B, which will be later used to perform a cluster analysis on some other samples. The solution is straightforward:

```
# launching 'oppose' to extract significant words
zeta.test.results = oppose()

# combining two lists of words into one vector
```

```
set1 = zeta.test.results$words.preferred
set2 = zeta.test.results$words.avoided
words.from.zeta = c(set1, set2)

# using the above vector as features for 'stylo'
stylo(features = words.from.zeta)
```

9.2 Custom splitting rule

When calling stylo, users can define their tokenization rules. The argument `splitting.rule` takes the form of a regular expression which will be used to split input character strings into discrete units, usually words. In obsolete versions of the package 'stylo', the default splitting sequence of chars was `"[^[:alpha:]]+"` on Mac/Linux, and `"\\W+_"` on Windows. Two different splitting rules were used, because regular expressions are not entirely platform-independent. In the version 0.5.6, then, assumed letter characters have been indicated explicitly. Type `help(txt.to.words)` to see the actual chararacter codes that were used as default rule.

If you are sure that your corpus contains clean spaces as word delimiters (i.e. there is no punctuation), you can use a very simple splitting rule, such as the one listed below. Even better, the rule might allow spaces, tab characters and newlines as word delimiters (the second variant):

```
classify(splitting.rule="[:space:]+")
classify(splitting.rule="[ \t\n]+")
```

Some other regular expressions may include:

```
"[^\u0041-\u005A\u0061-\u007A]+"                # (standard Latin-1)
"[^\u00C0-\u00D6\u00D8-\u00F6\u00F8-\u00FF]+"  # (Western European)
"[^\u0100-\u017F]+"                             # (Central European)
```

The custom splitting rule option can be also used if your input "texts" contain sequences of POS-tags rather than words, e.g.:

```
N-voc ADJ N-gen N-gen N-nom N-gen ADJ-NUM ABBR N-nom ADJ ...
```

Then your splitting rule might look as follows:

```
"[^[A-Za-z]-]+"
```

9.3 Splitting rule and batch mode combined

In many cases, the default functionalities provided by the library `stylo` can turn out to be insufficient for your needs. Then, you can prepare your own corpus using some of the functions provided by the library, then pipe it into other R functions and/or packages, and take it back to `stylo` after relevant modifications. Let's consider the following example in which the aforementioned custom regular expression solution was used. Suppose there is a collection of texts stored in a subdirectory "corpus". However, these texts are tagged:

```
All_NNP true_JJ histories_NNS contain_VBP instruction_NN ;_: though_RB
,_, in_IN some_DT ,_, the_DT treasure_NN may_MD be_VB hard_JJ to_TO
find_VB ,_, and_CC when_WRB found_VBN ,_, ...
```

Suppose one wants do drop the lemmas and to get tags only: NNP JJ NNS VBP NN RB IN DT DT NN MD VB JJ TO VB CC WRB VBN RB... It is possible to extract the grammatical annotation via the function `parse.pos.tags()`, using the following code:

```
# loading all input texts from the directory 'corpus':
my.raw.data = load.corpus(files = dir(), corpus.dir = "corpus")

# we invoke the function 'parse.pos.tags'
my.cool.data = parse.pos.tags(my.raw.data, tagger = "stanford",
      feature = "pos")

# now, we launch stylo() with an argument:
stylo(parsed.corpus = my.cool.data)
```

The above function is not supported by the older versions of `stylo` (ver. ¡0.6.2), but it can be easily worked around:

```
# loading all input texts from the directory 'corpus':
my.raw.data = load.corpus(files = dir(), corpus.dir = "corpus")

# now, it's time for some substitutions and regular expressions:
my.slightly.better.data = lapply(my.raw.data, gsub,
          pattern = "[[:alpha:],.;'-]+_", replacement="")

# to get rid of punctuation marks, another regexp might be helpful:
my.acceptable.data = lapply(my.slightly.better.data, gsub,
          pattern = "[[:punct:]]+", replacement = "")

# next, using the function 'txt.to.words' (from the "stylo" package)
# one can tokenize the whole corpus:
my.cool.data = lapply(my.acceptable.data, txt.to.words)

# the last stage is to launch the stylo() function with an argument:
stylo(parsed.corpus = my.cool.data)
```

Similarly, custom tables of frequencies can be build separately, and used in a form of external R objects piped into `stylo()`:

```
# external frequencies:
my.table = make.table.of.frequencies(my.cool.data, words = c("nn",
      "jj","dt","prp") )
stylo(frequencies = my.table)
```

9.4 Custom similarity measures

The package `stylo` in ver. ¿0.6.0 provides a socket for defining and plugging in custom distance measures. Suppose you want to test the Cosine Delta (or, Würzburg Delta) distance discussed by Jannidis, Schöch and Pielstrom (2015). Their measures is basically a regular Cosine Distance applied to z-scored data. To use it with `stylo`, one has to prepare a custom function that will compute the distance out of table of frequencies. The following function does the job, even if it could be slighlty optimized:

```
wurzburg.cosine = function(x)
        # z-scoring the input matrix of frequencies
        x = scale(x)
        # computing cosine dissimilarity
        y = as.dist( x %*% t(x) / (sqrt(rowSums(x^2) %*% t(rowSums(x^2)))) )
        # then, turning it into cosine similarity
        z = 1 - y
        # getting the results
        return(z)
```

Now, the code has to be typed (or copy-pasted) to the R console so that it is visible for other functions. We are ready to use the usual functions, supplemented with an additional argument:

```
stylo(distance.measure = "wurzburg.cosine")

classify(distance.measure = "wurzburg.cosine")

rolling.classify(distance.measure = "wurzburg.cosine")
```

Other possible applications include e.g. testing if the Entropy Distance outperforms other similarity measures. The code for the similarity function is straightforward:

```
dist.entropy = function(x)
        A = t(t(x + 1) / colSums(x + 1))
        B = t(t(log(x + 2)) / -(colSums(A * log(A))))
        y = dist(B, method="manhattan")
        return(y)


stylo(distance.measure = "dist.entropy")
```

Etc. etc. The are plethora of possible distance measures. The users are encouraged to examine them all!

9.5 Large-scale stylometric tests

Suppose that one wants to conduct a large experiment: the goal is to perform multiple runs with different lengths of the wordlist, increasing gradually the 'start.at' variable. This option is not implemented in the package `stylo`. However, using the batch mode it is just a few steps to success. Consider the following tailored script:

```
### the script begins ###
library(stylo)

# assume we want to perform a series of tests using 50 words
# and gradually moving the starting point on the wordlist
# e.g., from 100 to 1000 by 50 (i.e. for 100, 150, 200, 250, ... 1000)
# this is a vector of the start points we want to test:
where.to.start = seq(100,1000,50)
```

```
for(current.start.point in where.to.start) {

    #############
    # CORE CODE:
    # in each iteration, 'stylo' will be launched in batch mode
    # the option "start.at" will be incremented
    stylo(gui = FALSE,
          display.on.screen = FALSE,
          use.existing.freq.tables = TRUE,
          corpus.lang = "English.all",
          mfw.min = 50, mfw.max = 50,
          start.at = current.start.point)
    #############

  # now, we want to get the table of distances
  current.results = results.stylo$distance.table

  # what about saving this table?
  # first, we have to create a unique file name to prevent overwriting
  # the same file in each iteration:
  current.filename = paste("distances_starting_at_",
                       current.start.point, ".txt", sep="")

  # now, it's time to save the results in their files
  write.table(file = current.filename, current.results)
}

# a short message on screen, followed by a newline char:
cat("what about another stylometric test?\n")
### the script is done ###
```

10 Error messages and troubleshooting

[TBD]

References

Alexis, A., Craig, H., and Elliot, J. (2014). Language chunking, data sparseness, and the value of a long marker list: explorations with word n-grams and authorial attribution. *Literary and Linguistic Computing*, **29**(2): 147–63.

Argamon, S. (2008). Interpreting Burrows's delta: geometric and probabilistic foundations. *Literary and Linguistic Computing*, **23**(2): 131–47.

Burrows, J. (2002). Delta: a measure of stylistic difference and a guide to likely authorship. *Literary and Linguistic Computing*, **17**(3): 267–87.

Burrows, J. F. (2007). All the way through: testing for authorship in different frequency strata. *Literary and Linguistic Computing*, **22**(1): 27–48.

Burrows, J. (2010). Never say always again: reflections on the numbers game. In McCarty, W. (ed), *Text and Genre in Reconstruction. Effects of Digitalization on Ideas,*

behaviors, Products and Institutions. Cambridge: Open Book Publishers, pp. 13–36.

Craig, H. and Kinney, A. F., eds. (2009). *Shakespeare, Computers, and the Mystery of Authorship*. Cambridge: Cambridge University Press.

van Dalen-Oskam, K. and van Zundert, J. (2007). Delta for Middle Dutch: author and copyist distinction in Walewein. *Literary and Linguistic Computing*, **22**: 345–62.

Eder, M. (2011). Style-markers in authorship attribution: a cross-language study of the authorial fingerprint. *Studies in Polish Linguistics*, **6**: 99–114. `http://www.wuj.pl/page,art,artid,1923.html`.

Eder, M. (2013). Mind your corpus: systematic errors in authorship attribution. *Literary and Linguistic Computing*, **28**(4): 603–14.

Eder, M. (2015). Does size matter? Authorship attribution, short samples, big problem. *Digital Scholarship in the Humanities*, **30**(2): 167–82.

Eder, M. (forthcoming). Visualization in stylometry: cluster analysis using networks. *Digital Scholarship in the Humanities*, **30**, doi: 10.1093/llc/fqv061, in press.

Eder, M. and Rybicki, J. (2011). Stylometry with R. In *Digital Humanities 2011: Conference abstracts*. Stanford University, CA, pp. 308–11.

Eder, M., Kestemont, M. and Rybicki, J. (2013). Stylometry with R: a suite of tools. In *Digital Humanities 2013: Conference Abstracts*. Lincoln: University of Nebraska-Lincoln, pp. 487–89.

Hoover, D. L. (2002). Frequent word sequences and statistical stylistics. *Literary and Linguistic Computing*, **17**: 157–80.

Hoover, D. L. (2003). Frequent collocations and authorial style. *Literary and Linguistic Computing*, **18**: 261–86.

Hoover, D. (2004a) Testing Burrows's Delta. *Literary and Linguistic Computing*, **19**(4): 453–75.

Hoover, D. (2004b). Delta prime. *Literary and Linguistic Computing*, **19**(4): 477–95.

Hoover, D. (2010). Teasing out authorship and style with t-tests and Zeta. In *Digital Humanities 2010: Conference Abstracts*. King's College London, pp. 168–70.

Hoover, D. (2011). The Tutor's Story: a case study of mixed authorship. In *Digital Humanities 2011: Conference Abstracts*. Stanford University, Stanford, CA, pp. 149–51.

Hoover, D. L. (2012). The rarer they are, the more they are, the less they matter. In *Digital Humanities 2012: Conference Abstracts*. Hamburg University, Hamburg, pp. 218–21.

Jannidis, F., Schöch, C. and Pielstrom, S. (2015). Improving Burrows' Delta – An empirical evaluation of text distance measures. In *Digital Humanities 2015: Conference Abstracts*, `http://dh2015.org/abstracts`.

Juola, P., Noecker, J., Ryan, M., and Zhao, M. (2008). JGAAP3.0 – authorship attribution for the rest of us. In *Digital Humanities 2008: Book of Abstracts*. University of Oulu, pp. 250–51.

Kestemont, M. (2010). Velthem et al. A stylometric analysis of the rhyme words in the account of the Battle of the Golden Spurs in the fifth part of the Spiegel historiael. *Queeste*, **17**: 1–34.

Kilgariff A. (2001). Comparing Corpora. *International Journal of Corpus Linguistics* **6**(1): 1–37.

Koppel, M., Schler, J. and Argamon, S. (2009). Computational methods in authorship attribution. *Journal of the American Society for Information Science and Technology*, **60**(1): 9–26.

Rybicki, J., Kestemont, M. and Hoover D. (2014). Collaborative authorship: Conrad, Ford and rolling delta. *Literary and Linguistic Computing*, **29**(3): 422–31.

Stamatatos, E. (2009). A survey of modern authorship attribution methods. *Journal of the American Society for Information Science and Technology*, **60**(3): 538–56.