# Contents

1 Introduction ................................................. 5
   1.1 Text Reuse ........................................... 5
   1.2 TRACER ................................................. 5
      1.2.1 Current status .................................... 5
      1.2.2 Under the hood .................................... 6

2 System prerequisites ................................. 7
   2.1 Install Java ............................................ 7
   2.2 Install Apache Ant ................................. 7
      2.2.1 Downloading Apache Ant ....................... 8
      2.2.2 Unpacking Apache Ant ......................... 8
      2.2.3 Adding the environment variable ANT_HOME .... 8
      2.2.4 Updating the PATH ....................... 9
   2.3 Install a Text Editor .......................... 10
   2.4 Install Unzip Software ...................... 10

3 TRACER installation and configuration file .... 11
   3.1 Download TRACER .............................. 11
   3.2 Download TRACER with Git .................... 11
      3.2.1 Working with Git ............................. 11
   3.3 Compiling TRACER .............................. 12
   3.4 TRACER's configuration file ............... 13

4 Formatting a text to meet TRACER requirements .... 15
   4.1 How to prepare your texts ..................... 15
      4.1.1 Segmentation .................................. 15
      4.1.2 Column layout ................................ 15
   4.2 Where to store your texts .................. 19
   4.3 Other input files .............................. 20
      4.3.1 TRACER support for morphological analysers .. 21

5 Preprocessing ............................................ 22
   5.1 First run of TRACER ......................... 22
   5.2 Results .............................................. 23
      5.2.1 Results storage ......................... 23
   5.3 Types of preprocessing ...................... 24
      5.3.1 Letter-level preprocessing ............. 24
      5.3.2 Word-level preprocessing ............. 25
      5.3.3 Results .................................. 25
      5.3.4 Customising parameters or properties ... 26
   5.4 Understanding preprocessing ............. 26
This user manual describes the implementation of the TRACER machine, a powerful and flexible suite of some 700 algorithms for the automatic detection of (historical) text reuse. TRACER was developed by Marco Büchler and is written in Java. It is the most comprehensive tool yet and it is continuously improved thanks to the feedback gathered by the numerous tutorials and workshops given by the eTRAP (Electronic Text Reuse Acquisition Project) team at international conferences and events. For more information about the eTRAP Research Group, please visit: http://www.etrap.eu

About TRACER

TRACER Developer: Marco Büchler
TRACER homepage: http://www.etrap.eu/research/tracer/
TRACER repository: http://vcs.etrap.eu/tracer-framework/tracer.git
TRACER javadoc: http://www.etrap.eu/tracer/doc/javadoc/
Medusa repository: http://vcs.etrap.eu/tracer-framework/medusa.git
Medusa javadoc: http://www.etrap.eu/medusa/doc/javadoc/Medusa-2.0/
TRACER bug reports: http://www.etrap.eu/redmine/projects/tracer/
TRACER presentation slides: http://www.etrap.eu/tutorials/

Access to the .git repositories can be requested for free by writing to contact@etrap.eu

Copyright

TRACER is released under an Academic Free License 3.0 (AFL).
This manual is published under a Creative Commons Attribution-ShareAlike 4.0 International License.

Version history

Version 1.2 (2017-05-23): Updated version following the AIUCD (Associazione per l’Informatica Umancistica e la Cultura Digitale) 2017 tutorial.
Version 1.1 (2016-09-09): Updated version following the AIUCD (Associazione per l’Informatica Umancistica e la Cultura Digitale) 2016 tutorial.

Feedback

We greatly appreciate your feedback on this handbook. Your suggestions and comments will help us improve our documentation and adapt TRACER to your needs. We welcome both general feedback about TRACER and manual-specific suggestions. To get in touch, please email us at contact@etrap.eu
Introduction

1.1 Text Reuse

At its most basic level, text reuse is a form of written text repetition or borrowing. Text reuse can take the form of an allusion, a paraphrase or even a verbatim quotation, and occurs when one author borrows or reuses text from an earlier or contemporary author. The borrower, or quoting author, may wish to reproduce the text of the quoted author word-for-word or reformulate it completely. We call this form of borrowing “intentional” text reuse. “Unintentional” text reuse can be understood as an idiom or a winged word, whose origin is unknown and that has become part of common usage. Text reuse detection on historical data is particularly challenging due to the fragmentary nature of the texts, the language evolution, as well as linguistic variants and copying errors.

1.2 TRACER

TRACER is a suite of 700 algorithms, whose features can be combined to create the optimal model for detecting those words, sentences and ideas that have been reused across texts. Created by Marco Büchler during the eTRACES project, TRACER is designed to facilitate research in Text Reuse detection and many have made use of it to identify plagiarism in a text, as well as verbatim and near verbatim quotations, paraphrasing and even allusions. The thousands of feature combinations that TRACER supports allow to investigate not only contemporary texts, but also complex historical texts where reuse is harder to spot (see section 14.2 in the Appendix for a more detailed description of TRACER’s inner workings). TRACER is a command line engine. The reason it does not come with a user-interface is to boost the computing speed and performance. TRACER uses large and remotely accessible servers, which also facilitate the computation of large data-sets. The reuse results, however, can be visualised in a more readable format via TRAViz (see section 10).

1.2.1 Current status

TRACER is language independent and has been successfully tested on Ancient Greek, Arabic, Coptic, English, German, Hebrew, Latin and Tibetan. The External Reports section of this manual lists reports written by TRACER users. We continuously seek new languages to work with. TRACER has been used on both historical and modern texts. A list of bug reports and current developments is available here.
1.2.2 Under the hood

This manual provides a user-friendly guide to text reuse detection for both novice and expert users. For this reason, it does not give an accurate description of the roughly 700 algorithms constituting TRACER but only the necessary knowledge in order to perform simple text reuse detection tasks. Those who wish to study TRACER's algorithms in detail should consult TRACER's Javadoc.
To use TRACER you must first ensure that you have Java (version 8 or later), a text editor and unzip software running on your computer.

### 2.1 Install Java

The pre-compiled version of TRACER requires version 8 or higher of Java. To check your Java version, open the terminal or command line and type:

```java
java -version
```

If the resulting message displays a version number below 8, a new Java package needs to be installed. Download the package for your operating system, install, restart the computer, open the terminal and re-type the command above. Your version should have changed to 8.

If TRACER is downloaded from its GitLab repository (see section 3.2) and compiled `locally` the earlier Java 6 version is sufficient. Nevertheless, the use of the latest compiler, such as JDK 8 or later, is recommended.

### 2.2 Install Apache Ant

Apache Ant is a build system used to create an executable program. Among its many benefits, Ant allows to execute sequences of tasks from an XML file, typically called `build.xml`. Apache Ant is used to compile TRACER’s source code into an executable program (see 3.3).

If you install TRACER on a Linux system, you can use package managers such as `dnf` or `apt-get` to install Apache Ant. The benefit of using your system’s package manager is that new releases of Ant will be automatically installed when performing an update.

To install Apache Ant with `apt-get` type the following command:

```bash
sudo apt-get install ant
```

If your Linux distribution uses `dnf` then type the following command:

```bash
sudo dnf install ant
```

If you’re a Linux user, the installation instructions above should suffice in helping you run Apache Ant and you can therefore skip the rest of this section. If, however, you would prefer to install Ant manually, then please read on.

---

1 These can be accessed [here](#).
2.2.1 Downloading Apache Ant

Apache Ant\(^3\) is a regularly updates software. At the time of writing these guidelines (May 2017), the latest version of Ant is 1.10.1. The following instructions are based on this version. Please ensure your Ant version is always up-to-date.

Navigate to the Apache Ant download page. There, the Current Release of Ant section lists different download options in archive formats. Select the .zip format. For the Ant 1.10.1 version, select apache-ant-1.10.1-bin.zip. The bin substring in the file name indicates that a pre-compiled binary and executable java program is included in this release of Ant.

Please store this file in a dedicated folder on your machine, such as /Users/johnsmith/Tools. If you want to make Apache Ant available for more than one user on your machine (e.g. make it accessible from a server), you should not install Ant in a personal home folder (e.g. /Users/johnsmith but in an openly accessible folder, such as /opt/apache-ant. The instructions below are based on the installation of Ant in a personal folder.

2.2.2 Unpacking Apache Ant

The download file can be extracted by navigating to the download folder, right clicking the apache-ant-1.10.1-bin.zip file and selecting 'Extract'. Alternatively, the extraction can be processed via the command line. To do so, open a terminal and use the cd command to change to your download folder. For example:

```
cd /Users/johnsmith/Tools
```

To unzip the file, type:

```
unzip apache-ant-1.10.1-bin.zip
```

After unzipping the file, a new folder apache-ant-1.10.1 is created. You can verify its existence by typing:

```
lsc -l
```

For Windows users the command is not ls -l but:

```
dir
```

Under Mac OS X and Linux distributions, it is also necessary to set the access rights so that the program can be also executed. You can set the necessary rights by typing:

```
chmod -R 755 *
```

2.2.3 Adding the environment variable ANT_HOME

An environment variable is a key-value-pair to which all system-wide programs have access. The Apache Ant software looks for the environment variable ANT_HOME along with the folder in which Ant is installed. To reach the Ant home folder, in the command line type:

```
cd apache-ant-1.10.1
```

To visualise the path of Ant's home folder, type:

```
pwd
```

This command stands for **parent working directory** and will show you the folder's location on your computer (e.g. /Users/johnsmith/Tools/apache-ant-1.10.1). You should now copy the path as you will need it in the next step.

In order to make the value or path available for the environment variable ANT_HOME type:

```
cd -
```

---

\(^3\)Link available at: [http://ant.apache.org/bindownload.cgi](http://ant.apache.org/bindownload.cgi)
which will take you to your personal home folder (i.e. /Users/johnsmith/). In this folder, you should see a file called .bash_profile; if not, create it by typing the following in the command line:

```
vim ~/.bash_profile
```

`vim` is a command line text editor and works equally on local and remote machines. To edit the file .bash_profile press the i button on your keyboard. When in i (i.e. insert) mode, type:

```
export ANT_HOME=
```

and paste the path you copied earlier after the equal sign, like this:

```
export ANT_HOME=/Users/johnsmith/Tools/apache-ant-1.10.1
```

To exit the `vim` insert mode, press the ESC button on your keyboard. To save the changes and close `vim`, type:

```
:wq
```

where w stands for 'write' and q stands for 'quit'.

**On Windows machines**, setting the environment variable works a little differently. First, right click My Computer, click Properties and go to Advanced system settings. Click now on the Advanced tab. Click on Environment Variables and then on New .... Insert ANT_HOME as variable and /Users/johnsmith/Tools/apache-ant-1.10.1 as its value.

**Note 1**: You can download and use different versions of Apache Ant at the same time by storing the unpacked files separately (e.g. in a folder called 'Tools', such as /Users/johnsmith/Tools/). You can then define the version you want to use by typing the version number in the ANT_HOME environment variable.

**Note 2**: The same principle applies to Java. You can store multiple versions of Java on your machine and define which version you wish to run in the JAVA_HOME variable.

If you wish to double-check the value of a variable, type:

```
export
```

This command will list all environment variables with their respective values. To select only the ANT_HOME variable, type:

```
export | grep ANT_HOME
```

### 2.2.4 Updating the PATH

The last step is to update the environment variable PATH, which lists all directories of the file system that the computer should search for an executable program. The procedure is exactly the same as that described in section 2.2.3.

Linux and Mac OS X users should open the .bash_profile file again and in the last line type:

```
export PATH=$PATH:$ANT_HOME/bin
```

This command makes the scripts in the bin folder of ANT_HOME available to the operating system to look for the ant command.

Windows users should follow the same procedure as that described in section 2.2.3 with the difference that the PATH variable already exists and need only be edited. Another distinction is that in Mac OS X and Linux systems, different directories are separated by a colon, while in Windows systems by a the semicolon. Furthermore, under Windows, the value of an environment variable is called by %ANT_HOME%, whereas under Linux and Mac OS X it is $ANT_HOME. The easiest way to add ANT_HOME under Windows to type:
at the end of the PATH variable's current value.
In the command line, now type:

```bash
ant -v
```
where `-v` stands for 'version'. This commands shows you the version of Ant you're using and the build date of the program. The entry in the `.bash_profile` file is important as it is now permanently available on your computer.

### 2.3 Install a Text Editor

Depending on your operating system, install a standalone text editor such as `Notepad++` or any other text editor of your choice. The best editor in terms of readability and performance is `Sublime Text`.

### 2.4 Install Unzip Software

Make sure you also have unzip software, such as `WinZip` or `7-Zip`, installed on your computer. The TRACER package you download has to be unzipped to be used.
3.1 Download TRACER

Download and unzip TRACER from http://www.etrap.eu/tracer/ to a folder on your computer. TRACER zipped is some 107MB in size, unzipped it’s roughly 500MB. Additionally, in order to run TRACER needs between 5GB and 10GB of space. It is therefore important that you locate a folder on your computer with 10GB of space in which TRACER can be installed and computed. Do not store and run TRACER on a USB drive as this will result in very slow computing times.

3.2 Download TRACER with Git

Alternatively, you can clone the most recent releases of TRACER from our git repository. However, please be aware that the most recent version might be unstable. Once you’ve obtained an account you can download the latest version from here or by using git.

![TRACER's GitLab repository.](image)

### 3.2.1 Working with Git

Git is a so-called version control system that tracks file changes. It is heavily used by software developers and teams who simultaneously work on the same project.

If Git is not already installed on your machine, you can get a copy for your operating system from the official homepage. After installation, and to run Git, you must prefix commands in your terminal with git. An example is git status, which shows pending changes and suggests potential next steps. To get a copy of TRACER’s repository you first need to clone it. This is done by typing the following into your terminal:

```bash
git clone http://vcs.etrap.eu/tracer-framework/tracer.git
```

You’ll be prompted for your username and password, and a new folder will appear if the repository is successfully cloned. Before you can start working with TRACER you first have to build it from the source. Change into the tracer directory and execute the command ant. If you get a BUILD FAILED message some of the requirements haven’t been met.

---

4You can request an account at contact@etrap.eu
The recommended Git workflow is to first create a new branch by:

```
$ git checkout -b <your branchname>
```

You're now on a new local branch and all changes done to the tracked files will only remain in this local branch. You can get an overview of all branches by typing:

```
$ git branch -a
```

And you can switch branches in the same way you created one. The original branch is always called master. If you switch branches the tracked files will change according to your branch, while untracked files will remain in all branches. If you already built TRACER, you can type `git status` and see that some untracked files were created by the build process. If you want to obtain the newest version of TRACER just switch to the master branch and type:

```
$ git pull
```

This will download the latest changes into your master branch. Remember not to work on the master branch unless you're familiar with Git.

### Contributing

Before contributing please read the git commit conventions. With your account you also received the right to create your own branches and merge requests in GitLab. Please create a merge request if you want to contribute to TRACER.

### Reporting

With your account you also have the ability to create new issues. If you encounter a bug or are in need of a feature please create a new issue. If you find a bug, please always provide your build number.

### 3.3 Compiling TRACER

To compile TRACER, please navigate to the `tracer` folder containing the `build.xml` file, which includes all instructions on how to create the executable TRACER program, as well as the TRACER Javadocs.

To compile the software, in the command line type:

```
$ ant
```

which starts the build process. Depending on the disc type and the file system, this process can take between five and twenty seconds. Once the build process is complete, the `tracer.jar` file is created. The `.jar` files is TRACER's executable program and currently measures 33MB in size.

TRACER's build file `build.xml` contains a number of targets\(^5\) that aren't executed in order to speed up the build process. However, you can visualise all targets by typing:

```
$ ant -p verbose
```

which lists, among others, `javadoc`. If you now type:

```
$ ant javadoc
```

you'll create TRACER's Javadoc in the folder `doc/web/javadoc/Tracer-1.0/`. There, click on the `index.html` file to view the Javadoc in a web browser.

---

\(^5\) A 'target' is a procedure or pipeline. Compiling code is, for example, one target. A target can consist of one or more tasks.
3.4 TRACER's configuration file

TRACER's configuration file is its control tower. This is where you define the parameters of your detection tasks. To open TRACER's configuration file, `tracer_config.xml`, open a terminal window, change directory to your TRACER folder (with `cd`) and type:

```
vim conf/tracer_config.xml
```

![Image of terminal window with commands]

Figure 3.2: Open the terminal and navigate to the TRACER folder on your machine using the `cd` command. Then open TRACER's configuration file to define properties. The concept of 'defining properties' will become clear later in the manual (see section 5 onwards).

Press `ENTER` and you should now see the following screen:

![Image of configuration file]

Figure 3.3: After pressing `ENTER`, you should see this screen. This is the configuration file of TRACER, listing all of the properties with which you'll be working.
Properties are made up of a namespace and a class. For example:

```xml
<property name="PREPROCESSING_IMPL" value="eu.etrap.tracer.preprocessing.WordLevelPreprocessingImpl"/>
```

Figure 3.4: Structure of properties present in TRACER’s tracer_config.xml file.

To point TRACER to your corpus or texts, locate the eighth property, SENTENCE_FILE_NAME:

```xml
<property name="SENTENCE_FILE_NAME" value="data/corpus/Bible/KJV.txt"/>
```

Figure 3.5: The highlighted property on line 17 provides the TRACER directory containing the texts to be investigated.

As you can see, the default corpus of TRACER is the KJV.txt Bible file (King James Version). If you want to work with a different text, you must first format it in accordance with TRACER requirements and store it the corpora sub-folder of TRACER (see section 4.2).
4 Formatting a text to meet TRACER requirements

TRACER works with plain text files (.txt). This means that if you have marked-up texts in XML, you must remove all of the XML tags before you can use TRACER.

4.1 How to prepare your texts

To optimise the performance of TRACER, place your texts (two or more, depending on how many you wish to analyse) in the same .txt file, one under the other. Placing all texts in a single file is beneficial as TRACER will consider every line as a reuse unit to compare. To distinguish the texts, you will use different IDs (keep reading to find out how to do this).

4.1.1 Segmentation

The first thing you need to do is to segmentise your texts by verse, paragraph, sentence, or whatever unit you believe best suits your reuse analysis. Every unit must appear on a separate line in the .txt file. You can use the free NLTK service to quickly segmentise your text, but this tool will use full-stops as the only cue to identify the end of a sentence. Copy and paste your unsegmentised .txt into NLTK's text box. Click on the segmentation button, copy the segmentised output and paste it into a new .txt file. Save this new file and give it a clear name, such as mytext-segmentised.txt. Please note that NLTK only takes a certain number of lines, so if you have a very long text you will have to divide it into smaller chunks. Always double-check the output because sometimes formatting errors can sneak in unnoticed.

4.1.2 Column layout

Next, your .txt must contain four columns separated by TABs:

1. The first column contains unique sentence IDs (running numbers are recommended, as shown in Table 4.1).
2. The second column contains the sentence.
3. The third column can contain the date of the file creation in the YYYY-MM-DD format or the word NULL. If NULL, make sure it is written in upper-case.
4. The fourth column contains the book or section from which the sentence is taken. This information is crucial for the visualisation shown in Figure 10.2. The top drop-down menu you see there will list the information you provide in this fourth column.

You can achieve this column structure by using Microsoft Excel or Libre Office. The instructions below are based on Libre Office Version 5.1.5.2.
First, you need to import the segmentised `.txt` file into Excel or Libre. To do so, simply right click on the `.txt` file, click on Open with and select either Excel or Libre. Alternatively, open Microsoft or Libre, click on File > Open and select the `.txt` file. Whatever you choose, it’s **essential** that you specify **TAB** as the field separator when importing the file. If you don’t, TRACER won’t be able to read your file. The import window looks something like this:

![Figure 4.1: Select TAB as the field separator.](image)

Click **OK** and you should now see something like this:

![Figure 4.2: How a segmentised text first appears in Libre Office.](image)
We now need to remove all blank rows. If you’re using **Libre**, select the entire text column, then click on the **AUTOFILTER** button in the toolbar. This generates a clickable arrow symbol on the first cell. Click it and then select **EMPTY** (see Figure 4.3). This will bring all blank rows to the top of the document (if you look at the row IDs now you’ll notice that the column only lists even numbers, i.e. alternating blank rows). Delete all of these by pressing the **SHIFT** button on your keyboard and selecting all of the rows. Then, deselect the **AUTOFILTER** button in the toolbar and save the changes.

![Figure 4.3: The filter icon and arrow as they appear in Libre Office.](image)

If you’re using **Microsoft Excel**, select the first column, click on the **FILTER** button in the toolbar. Again, click on the little arrow that is now available in the first cell. In the resulting pop-up window you need to do two things: first, click on the **SELECT ALL** option to deselect all options; second, scroll down to the bottom of the options pane and select **BLANKS**. This will bring all the empty rows to the top of the document (if you look at the row IDs now you’ll notice that the column only lists even numbers in blue). Delete all the empty rows by pressing the **SHIFT** button and selecting all the blue-numbered rows. Then, deselect the **FILTER** button in the toolbar and save the changes.

![Figure 4.4: Microsoft Excel’s filter buttons and pane. After deselecting all options within the pane, reselect Blanks in order to delete all blank rows.](image)

Next, add a column to the left of your sentence column. This new column should contain sentence IDs. To automate the creation of IDs, type in the first three IDs (one per sentence), then select the three ID cells, hover over the bottom right corner of the third cell and finally drag the selection all the way down to the last sentence. Remember to save the changes!

![Figure 4.5: The first three IDs are typed in manually. The rest are automatically generated by dragging the selection to the last sentence in the document. To drag, grab the bottom-right corner of the third cell in the ID column.](image)

The sentence IDs should be sequential and unique. The default set-up of TRACER requires IDs to be 7-digits in length and numbers below 2.000.000. If you’re analysing two texts, make sure to restart the ID sequence for your second text. So, for example:
Table 4.1: Required segment (sentence, verse, etc.) ID formatting. Texts to be analysed are told apart by the first two digits in the ID.

As you can see from Table 4.1, all IDs are 7-digits long and the two different texts to be analysed, A and B, are distinguished via the first two digits of the ID. TRACER’s default ID settings can be changed in the configuration file in the Selection section, as shown below.

Figure 4.6: The value of the `intWorkNumbering` property is set by default to 1000000. This can be changed to accommodate different ID schemes.

However, doing so is only recommended in consultation with the TRACER team as it may affect the detection process and your results.

Moreover, ensure that there are no new blank lines at the end of the document and that there’s no white-space between text A and text B. Any blank lines will bring up errors.

Next, add two columns to the right of your sentence column. In the third column you can either put a date of file-creation (in the YYYY-MM-DD format only) or the case-sensitive word NULL. To populate the entire column with the date or NULL, repeat the drag action used above but from the first cell only, not the third. By dragging from the first the cell values will not increase or change as you move downwards.

Figure 4.7: For the third column, in the first cell type either NULL or a date in the YYYY-MM-DD format and apply the changes to the entire column by dragging the contents of the first cell all the way down to the last data cell in the document.

The fourth column should list the source of the text, whether it’s a book, a chapter or the title of your text. This information is necessary for the text reuse visualisation to work later on. To populate this column repeat the actions described in the previous paragraph.

Save your four-column file as a .csv document. Finally, change the extension of the file from .csv to .txt.
That's it, your text is now TRACER-compatible!

Figure 4.9 below provides an example of the King James Bible Version text formatted for TRACER.\(^6\)

4.2 Where to store your texts

Place your texts in the \texttt{corpora} folder of TRACER's \texttt{data} folder (see Figure 4.10).

\begin{figure}
\centering
\includegraphics[width=\textwidth]{Figures/Fig4_10.png}
\caption{Structure of the TRACER folder (Mac `Finder' view). Deposit your .\texttt{txt} file in a new folder under \texttt{data} > \texttt{corpora}.}
\end{figure}

\textbf{Important}: never use white-spaces in file names.

Next, make sure TRACER's configuration file points to your .\texttt{txt} file. For example, if your text file is called \texttt{KJV.txt}, locate the \texttt{SENTENCE\_FILE\_NAME} property in the configuration file and add the path of your file to the \texttt{value} attribute:

\begin{verbatim}
6This is a different example from the one we've been working on but it should give you an idea of the final document layout.
\end{verbatim}
4.3 Other input files

In order to detect text reuse, TRACER also requires files containing linguistic information pertaining to the language you’re examining. These are:

- A *lemma* file listing all word-forms present in the text, their base-form and word-class or part-of-speech tag. This information can be automatically extracted using a morphological analyser that is able to read your language. The output file produced by your morphological analyser of choice needs to be converted to TRACER’s input format. TRACER takes three columns, separated by TABs: a column listing the *word-forms* as they appear in the text; a column providing the *base-form* or dictionary entry of every word-form; and a third column with the relative *word-class* (verb, noun, etc.):

![Figure 4.12: This TAB-separated three-column .txt file contains information TRACER needs in order to detect text reuse in Latin texts.](image)

- A *synonyms* file listing all dictionary entries for your particular language along with their synonyms. In other words, a thesaurus. You can extract these lists from WordNets, if available for the language you are working with. While WordNets typically display this information as *lemma synonym1, synonyms2, synonym3, ...* (one line per lemma), TRACER requires a bidirectional two-column list, such as:

![Figure 4.13: This TAB-separated two-column .txt file contains information TRACER needs in order to detect looser forms of text reuse in Latin texts (e.g. paraphrase). The left column contains the base-form or lemma and the right column one of its synonyms.](image)

Figure 4.13 displays one synonym per lemma. For lemmas with multiple synonyms the file should be structured as follows:

| lemma 1 | synonym1  
|--------|-----------
| lemma 2 | synonym2  
| lemma 3 | synonym3  

And should be bidirectional:

| lemma 1 | synonym1  
|--------|-----------
| synonym1 | lemma 1  

Figure 4.11: The path to your .txt file must be specified in the `tracer_config.xml` file in the `SENTENCE_FILE_NAME` property.
If you wish to detect non-verbatim text reuse (e.g. paraphrase) you need both the lemma and the synonyms files. If you’re only interested in word-for-word text reuse, you don’t need the synonyms file. The .txt, .lemma and, optionally, the .synonyms files must be deposited in TRACER’s corpora directory and declared in the tracer_config.xml file, as shown below.

Figure 4.14: The path of the input files must be specified in the tracer_config.xml file.

If you don’t use a .syns file, you needn't delete the SYNONYMS_FILE_NAME property but must declare this in the Preprocessing section of the configuration file (see section 5.3.2).

Can the formatting be automated?

For the .syns file, TRACER currently doesn't provide any means of automating the column-formatting. For the .lemma file, the latest release of TRACER provides automatic column-formatting for files analysed with TreeTagger and Stanford CoreNLP morphological analysers (see section 4.3.1). Future work on TRACER includes the provision of automatic formatting options for .syns files, as well as .lemma files generated with other morphological analysers.

4.3.1 TRACER support for morphological analysers

One of the most recent developments of TRACER includes two preprocessing options to convert the output of the TreeTagger\(^7\) and Stanford CoreNLP\(^8\) tools to TRACER’s required input format, thus saving the user considerable preprocessing work. Figure 4.15 shows the corresponding preprocessing sections in TRACER’s configuration file.

Figure 4.15: TRACER preprocessing configuration options to import and convert TreeTagger and Stanford CoreNLP output to the required text reuse detection input format.

As you can see, each category in Figure 4.15 has five properties. If you wish to import TreeTagger and Stanford CoreNLP files into TRACER you need to change the first two properties: the first property defines the file path, while the second defines the input file’s extension. The third property, which must not be changed, contains the mapping instructions TRACER needs to convert input into output.

\(^7\)At: http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/

\(^8\)At: http://stanfordnlp.github.io/CoreNLP/
With TRACER you can perform two types of preprocessing: letter-level and word-level processing. For historical texts, letter-level preprocessing can be useful to, for example, read *scriptura continua* or, in other words, to read a sequence of letters without white spaces. Word-level preprocessing is the default preprocessing technique and is used to read sequences of words. Within Word-level preprocessing, synonym processing yields the best results. Other options include string similarity, which can be used to detect similar words or strings to reveal OCR errors, morphological variation or even spelling errors.

### 5.1 First run of TRACER

Open the terminal and navigate to the TRACER folder (using the command `cd`). To start TRACER, type:

```
java -Xmx600m -Dfile.encoding=UTF-8 -Deu.etrap.medusa.config.ClassConfig=conf/tracer_config.xml -jar tracer.jar
```

What does this command do?

- `java` opens a Java programme.
- `-Xmx600m` defines the amount of memory to be used for the detection. In this case, the computation is set to use up to 600 MB memory. If you work on large data-sets and your computer supports it, you might have to raise this parameter to 1g (=1GB), 2g (=2GB) or more.
- `OPTIONAL -Dfile.encoding` sets the encoding of your input file. This is important because operating systems use different default encoding. To set the character encoding to a **Unicode standard**, add the relevant code as follows: `-Dfile.encoding=UTF-8`.
- `-Deu.etrap.medusa.config.ClassConfig=conf/tracer_config.xml` calls the configuration file.
- `D` is a system property.

Press ENTER and TRACER will perform a first text reuse run of the texts. The speed of the first run will depend on the memory of your computer but it generally takes a few minutes. When TRACER is done, you'll see the screen below:
Figure 5.1: The first run of TRACER is complete. The words DONE!! and END OF PROCESS LEVEL 5 (SCORING) indicate that TRACER has successfully gone through the Preprocessing, Featuring, Selection, Linking and Scoring steps.

5.2 Results

The results of the first run and, indeed, the results of every run of TRACER are automatically stored in a newly generated folder called TRACER_DATA under data > corpora > Bible:

If you’re working with your own texts—not TRACER’s default Bible data—the TRACER_DATA folder would be created in the relevant directory (specified in section 3.4).

5.2.1 Results storage

TRACER organises the results to reflect any changes the user makes to the configuration parameters (more about configuration customisations in section 5.3). It creates a deep folder structure with long but self-explanatory folder names. Figure 5.3 below is the extension of Figure 5.2.
Figure 5.3: The folder structure within TRACER_DATA. Long folder names are used to reflect the property settings in the TRACER tracer_config.xml file. This system allows users to better locate their results, especially when running TRACER multiple times with modified parameters.

As you can see, the first run of TRACER produced a folder with a very long name:

```
01-02-WLP-lem_true_syn_true_ssim_false_redwo_false-ngram_5-LLR_true_toLC_false_rDia_false_w2wl_false-wlt_5
```

Here’s how you read it (see also section 5.3.1 and section 5.3.2):

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preprocessing step</td>
<td>(01)-word-level</td>
</tr>
<tr>
<td></td>
<td>(02)-Word Level Processing</td>
</tr>
<tr>
<td>Lemmatisation</td>
<td>ENABLED</td>
</tr>
<tr>
<td>Synonym replacement</td>
<td>ENABLED</td>
</tr>
<tr>
<td>Replace String</td>
<td>DISABLED</td>
</tr>
<tr>
<td>similar words</td>
<td></td>
</tr>
<tr>
<td>Reduced words option</td>
<td>DISABLED</td>
</tr>
<tr>
<td>to the most</td>
<td></td>
</tr>
<tr>
<td>significant 5 letter</td>
<td></td>
</tr>
<tr>
<td>n-gram</td>
<td></td>
</tr>
<tr>
<td>significance is measured</td>
<td>by the Log-Likelihood Ratio</td>
</tr>
<tr>
<td>ENABLED</td>
<td></td>
</tr>
<tr>
<td>Everything to Lower Case</td>
<td>DISABLED</td>
</tr>
<tr>
<td>remove Diacritics</td>
<td>DISABLED</td>
</tr>
<tr>
<td>Replace word by word</td>
<td>DISABLED</td>
</tr>
<tr>
<td>length</td>
<td></td>
</tr>
<tr>
<td>words of 5 letters and</td>
<td></td>
</tr>
<tr>
<td>more</td>
<td></td>
</tr>
</tbody>
</table>

5.3 Types of preprocessing

After the first run, you can decide whether to keep TRACER’s default preprocessing properties or change them. Any changes you make entirely depend on your texts but TRACER can perform two levels of preprocessing: letter-level and word-level processing.

5.3.1 Letter-level preprocessing

This can be useful for historical texts to, for example, address *scriptura continua* where the words are not separated by punctuation.

```
<category name="eu.etrop.tracer.preprocessing.LetterLevelPreprocessingImpl">
  <property name="boolReplaceWhitespaces" value="false"/>
  <property name="intNgramSize" value="5"/>
  <property name="boolRemoveDiacritics" value="false"/>
  <property name="boolMakeAllLowerCase" value="false"/>
</category>
```

Figure 5.4: The letter-level properties of TRACER as listed in the tracer_config.xml file.

More information on this in the next manual release.
5.3.2 Word-level preprocessing

This processing can be used to, for example, reduce inflected words to their base form (linguistic approach). Synonym processing is the default/most popular word-level processing but cohyponym processing yields better results. What is a cohyponym? Observe Figure 5.5 below.

![Figure 5.5: Linguistic tree illustrating relationships between terms describing colour. Source: Wikimedia Commons.](image)

Other types of word-level processing are word-length replacement, which helps with calculation speed but also to better visualise the total number of features. And there is string similarity, which detects similar words or strings and thus helps to reveal OCR errors, morphological variation, typing or spelling errors, etc.

![Figure 5.6: The word-level properties of TRACER as listed in the tracer_config.xml file.](image)

5.3.3 Results

Looking back at Figure 5.3, you'll notice that the WLP folder (the one with the very long name!) contains a number of files with the .prep suffix (which stands for preprocessing). Let's look at these one by one.

**KJV.prep**

This is the result file of the preprocessing step. If you look carefully, you'll notice that sentence 4000001 contains the words get and make, which are the preprocessed versions of the original beginning and created (see Figure 4.9). The lemmatisation settings have erroneously replaced beginning with get and the synonym replacement has changed created to make. These settings can be changed in order to correct any mistakes.

Similarly, in sentence 4000006, the archaic term midst has not been replaced with the modern equivalent middle but it could if we wanted! For this reason, it's important that you thoroughly check the KJV.prep file before moving onto the next step.
KJV.prep.inv
inv stands for inverted list. It shows you that a specific word (first number) appears in a specific verse (second number) in a specific position (third number).

KJV.prep.meta
This file provides overview information about the preprocessing tasks, the settings and the results. For example, it tells us that 103,673 words out of the entire corpus were lemmatised.

The WLP folder also contains further nested folders, described in this manual as we move along.

5.3.4 Customising parameters or properties
To change preprocessing properties simply enable/disable them in the tracer_config.xml as you see fit. For example, to switch off lemmatisation, replace its corresponding value true with the value false (as illustrated in Figure 5.6) and save the changes. When all the changes have been made, rerun TRACER and a new folder within TRACER_DATA will be produced with the updated results. Lemmatisation helps to, for example, discriminate nouns from verbs (e.g. the word 'power', which can be both a verb and a noun). Lemmatisation is especially important when we wish to analyse paraphrases and allusions. If we wanted to find direct quotations (verbatim or near verbatim), lemmatisation is not going to be useful so we would switch its value in the tracer_config.xml file back to false.

5.4 Understanding preprocessing
The processing techniques described so far are common practice in Natural Language Processing (NLP). One of the laws of NLP is known as Zipf's law. According to Zipf, 90% of words in a text are rare,
occurring 10 times or less; 50% of all words occur only once; 16% of all words occur only twice and 8% only three times, and so on. This means that the frequency of any word is inversely proportional to its statistical rank. For example, according to wordcount.org, the most popular word in the English language is ‘the’. As such, its rank is 1 (see Figure 5.7).

**Figure 5.7: ‘The’ is the most popular word in the English language.**

The word ‘cat’ ranks 2532 and does not appear as often as the word ‘the’ (Figure 5.8).

**Figure 5.8: Rank of the English word ‘cat’.

So, again, the higher the rank, the less frequent the word and vice versa. We can visualise this proportion as a log-log graph, which reveals a ‘straight line’ relation between word frequency and ranking (Figure 5.9). Amazingly, Zipf’s law applies to all languages.
Figure 5.9: Log-log graph of word frequency and ranking. The higher the rank of a word, the lower its frequency and vice versa.
Now that we've preprocessed the text, we can proceed to breaking it down into units or features that can be compared (e.g. words, bigrams, trigrams). For this reason, this step is known as Featurizing or Training. Figure 6.1 below provides an overview of Featurizing. TRACER can perform two types of featurizing, Syntactic and Semantic. Syntactic dependency parsing is not always accurate because parsers (and the TRACER parser) are not yet able to infer dependencies. But if TRACER takes manually annotated data from treebanks, for example, then the parser would be able to produce usable results.

More information about semantic dependency parsing in the next manual release.

As you can see, there exists also a third type of Featurizing implementation, Non-statistic Approaches. These aren’t part of TRACER. Among these, Verba dicendi featuring could be used for fragmentary texts but it’s not a stable feature; Surface Features, for example quotation marks, are also an unstable feature as they don’t always occur in historical texts; for Canonical References, please consult the research of Dr Matteo Romanello; for Signal Processing see [SC08].

6.1 Features

TRACER supports 10 different features or featurizing units, from words as features to 10-gram features. Generally, features of 3-4 grams are sufficient for text reuse analyses. The n in ‘n-gram’ could be a character or a word.
• **Word-based featuring**: good for paraphrase detection. With this type of featuring one should also lemmatise and activate synonym replacement.

• **Bigram featuring**: good to detect verbatim and near-verbatim text reuse. Lemmatisation and synonym replacement with this N-gram approach might not work or be necessary.

• **Trigram featuring**: good to detect verbatim and near-verbatim text reuse. Lemmatisation and synonym replacement with this N-gram approach might not work or be necessary.

### 6.1.1 Feature density

The *Selection* step in TRACER asks users to define a *Feature density* for every detection task. For more information about *Feature density*, see section 7.

### 6.2 Types of featuring

#### 6.2.1 Overlapping

An overlapping type of featuring is **shingling**, a process typically used to detect near-verbatim reuse. In NLP or text mining, a shingle is an *n*-gram and the process of shingling creates a sequence of overlapping tokens in a document. Here are some examples of shingling:

**Example sentence:** *I have a very big house*

<table>
<thead>
<tr>
<th><strong>Bigram shingling</strong> (bigram = an <em>n</em>-gram of size 2)</th>
<th><strong>Four-gram shingling</strong> (four-gram = an <em>n</em>-gram of size 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(I have), (have a), (a very), (very big), (big house)</td>
<td>(I have a very), (have a very big), (a very big house)</td>
</tr>
<tr>
<td>Feature 1 = I have</td>
<td>Feature 1 = I have a very</td>
</tr>
<tr>
<td>Feature 2 = have a</td>
<td>Feature 2 = have a very big</td>
</tr>
<tr>
<td>Feature 3 = a very</td>
<td>Feature 3 = a very big house</td>
</tr>
<tr>
<td>Feature 4 = very big</td>
<td>Feature 4 = very big house</td>
</tr>
<tr>
<td>Feature 5 = big house</td>
<td></td>
</tr>
</tbody>
</table>

**Trigram shingling** (trigram = an *n*-gram of size 3)

(I have a), (have a very), (a very big, (very big house)

| Feature 1 = I have a                                |
| Feature 2 = have a very                             |
| Feature 3 = a very big                              |
| Feature 4 = very big house                          |

Shingling creates more features, so your detector will need more time to compute similarity.

#### 6.2.2 Non-overlapping

A non-overlapping type of featuring is **hash-breaking**, a process typically used to detect duplicates or exact copies. Hash-breaking creates features with no overlap. Here are some examples of hash-breaking:

**Example sentence:** *I have a very big house*
Bigram hash-breaking
(bigram = an $n$-gram of size 2)
(I have), (a very), (big house)

Feature 1 = I have
Feature 2 = a very
Feature 3 = big house

Trigram hash-breaking
(trigram = an $n$-gram of size 3)
(I have a), (very big house)

Feature 1 = I have a
Feature 2 = very big house

6.2.3 Distance-based
The Distance-based Bigram can be used to improve retrieval performance. Distance-based bigrams are good for detecting paraphrases or reuse that is not 100% literal. The Distance-based bigram is a word pair whose distance between the two components is $\geq 1$.

Example sentence: *I have a big house*

Bigram

<table>
<thead>
<tr>
<th>Feature</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>I have 1</td>
<td>1</td>
</tr>
<tr>
<td>have a 2</td>
<td>2</td>
</tr>
<tr>
<td>a big 3</td>
<td>3</td>
</tr>
<tr>
<td>big house 4</td>
<td>4</td>
</tr>
</tbody>
</table>

where 1 describes the distance between ‘I’ and ‘have’ (one word)
where 2 describes the distance between ‘I’ and ‘a’
where 3 describes the distance between ‘I’ and ‘big’
where 4 describes the distance between ‘I’ and ‘house’

6.3 Shingling or hash-breaking?
Let us assume you’re attempting to find the following string:

```
ABCDE
```

in the following reuse units:

1. GHABCDELM
2. NABCDEXYZ

What would be the best algorithm to do so, *shingling* or *hash-breaking*? Let’s compare these two methods.

6.3.1 Hash-breaking
If we use hash-breaks of 2 features the similarity detection in string 2 won't work because we break the reuse units the wrong way:

```
AB CD ES
NA BC DE XY Z$  
```

As you can see, the AB feature in string 2 is split between NA and BC. For string 1, on the other hand, hash-breaking works:

```
AB CD ES
GH AB CD DE LM
```

In this case the breaks allow us to find the reuse.
6.3.2 Shingling

Both bigram and trigram shingling work to detect the example reuse above in strings 1 and 2. It doesn’t matter what method we use. What changes, however, is performance.

6.4 Customising parameters or properties

In TRACER, the featuring settings can be changed in the `tracer_config.xml` (see Figure 6.2).

![Figure 6.2: The value of the highlighted property can be changed to perform shingling or hash-breaking tasks.](image)

Let’s assume we want to change the above property in order to run trigram shingling on the text. The changed property will look like this:

![Figure 6.3: The value of the highlighted property has been changed from BiGramShinglingTrainingImpl to TriGramShinglingTrainingImpl.](image)
We save the document and rerun TRACER. This reuse detection task produces a number of files, including:

**KJV.meta**
This file provides a human-readable overview of the trigram shingling training.

```
<table>
<thead>
<tr>
<th>FEATURE ID</th>
<th>REUSE UNIT ID</th>
<th>POSITION IN REUSE UNIT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**KJV.train**
This file is organised as follows:

```
FEATURE ID - REUSE UNIT ID - POSITION IN REUSE UNIT
```

**KJV.fmap**
This file provides a map or breakdown of each feature:

```
FEATURE ID - WORD ID - WORD ID - WORD ID
```

**KJV.feats**
This document is the word index, where words are sorted by frequency:

```
WORD ID - WORD - FREQUENCY
```

<table>
<thead>
<tr>
<th>WORD ID</th>
<th>WORD</th>
<th>FREQUENCY</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>the</td>
<td>59385</td>
</tr>
<tr>
<td>102</td>
<td>and</td>
<td>37310</td>
</tr>
<tr>
<td>103</td>
<td>of</td>
<td>33041</td>
</tr>
<tr>
<td>104</td>
<td>to</td>
<td>12899</td>
</tr>
</tbody>
</table>
If you look again at the *Training* property in TRACER's `tracer_config.xml` (Figure 6.3), you'll notice that the trigram shingling training we've done is *syntactic*. TRACER also contains a *semantic* featuring property, immediately preceding the syntactic property (Figure 6.4). This property is commented out by default but can be enabled by removing the comment syntax (`<!-- ... -->`).

![Figure 6.4: TRACER's training or featuring can also be semantic.](image)

For *semantic* training, TRACER looks at *words* and *co-occurrences*.
There are roughly six different selection strategies in TRACER, which can also be combined. Through Selection we look at the idiosyncrasies or minutiae of the features we’ve computed, thus focusing on fewer features while being faster and more precise. This is the point where we create our ‘fingerprint’ with the core reuse elements we’re interested in. Algorithmically-speaking, through Selection some entries in the KJV.train file will be kicked out. With Selection we’re narrowing down the search by filtering out all irrelevant results. The challenge in this step is to find out what our core elements or minutiae are.

### 7.1 Selection strategies

There are many different selection strategies, including:

- **Pruning**: maximum or minimum.
  - *Maximum* pruning cuts out high frequency words (e.g. function words); removing high frequency words reduces computational complexity. For example, if you’re looking at a feature that occurs only 10 times you are comparing one feature with 9 others - this means that you are making 90 comparisons. This is reasonably fast. If you, however, do not prune and have a frequency of 100, you would have to make 900 comparisons. That’s slow.
  - *Minimum* pruning removes low frequency words (e.g. words with a frequency of 1). Min pruning is used to clear up more space on the desk by reducing the size of the index.

- **Term weighting**: we give features a weight. Rarer words have a higher weight.

- **Frequency classes**: you can take the frequency of the most frequent word and divided it by the frequency of the word you want to compare it to.

- **Feature dependencies**: it makes a comparison between paired features and sees if there are features that tend to co-occur.

- **Random Selection**: random selection of features. Very good if you don’t have a clue of what to expect from your data.
• **Winnowing**: with this strategy we can pick a window. For example window size \( w = 2 \) means that the selection will work for every two features. If one word is a feature:

<table>
<thead>
<tr>
<th>word 1</th>
<th>word 2</th>
<th>word 3</th>
<th>word 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>The</td>
<td>house</td>
<td>is</td>
<td>blue</td>
</tr>
</tbody>
</table>

The selection is made based on the rarest parts of the winnowing window - i.e. the most infrequent features. But in this way function words are not entirely removed, since sometimes they can still be useful. With the winnowing algorithm it’s possible to select features all over the reuse unit and avoid clusters. For example, “To be or not to be that is the question”. It would appear that the only interesting word here is ‘question’ but we don’t want to be eliminating all of the rest. Winnowing, with its windows, allows us to pick the lowest ranked feature (in frequency) for every window - so there is a selected feature for every window and features are distributed evenly.

To change the selection strategy in TRACER, locate the **Selection strategy** in the `tracer_config.xml` file, as shown in Figure 7.1 below.

![Figure 7.1: The value of the highlighted Selection property in the tracer_config.xml file can be changed according to the preferred strategy.](image)

Different selection strategies require different parameters, and it’s often difficult to compare and decide which strategy works best for a given case. For this reason, we use the **Feature Density** or, in other words, the comparison between the overall number of features and the number of features selected. The Feature density parameter accepts value ranges between 0 and 1; for, for example, if it is set to 0.8, TRACER will keep 80% of the features and ignore the remaining 20%.

Here’s how TRACER computes feature density:

\[
F = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} s_{ij}}{\sum_{i=1}^{n} \sum_{j=1}^{m} f_{ij}}
\]

Brief explanation of this formula in the next manual release.

Simply put:

- **Global knowledge**: Information derived from the entire corpus; global knowledge is, for example, the computed feature frequency in a corpus.

- **Local knowledge**: Information derived from the reuse unit (e.g. a sentence); local knowledge is the context of the reuse unit or, for instance, the length of its words.

- **Global usage**: Selection is applied to, for example, the entire text or corpus.
Table 7.1: Selection Knowledge vs. Selection Usage. The matrix compares Pros and Cons between the respective categories of Selection processes. Global Selection Knowledge with Local Selection Usage offers the best compromise between the mentioned advantages and disadvantages.

- **Local usage**: Selection is applied to the reuse unit.

These can be combined in the `tracer_config.xml` file in the following ways:

- `localglobal`: Local knowledge in a global context. This is the default setting of TRACER and the most used.

- `globallocal`: Global knowledge in local context.

- `globalglobal`: Global knowledge in a global context. TRACER treats every reuse unit in the same way but it can easily create empty reuse units.

- `locallocal`: Local knowledge in a local context. For example, given a certain word-length, TRACER removes from the reuse unit all words that are shorter than the specified length.
During the Linking stage, TRACER searches for the occurrences of the features defined in previous steps. Unlike other TRACER steps, which are all linear, Linking is of squared complexity depending on the feature frequency. Through Linking TRACER will deliver results based purely on the parameters set in previous steps. It is up to us to interpret those results and filter out what is good and what is bad.

### 8.1 Types of Linking

TRACER can perform two types of Linking:

- **Intra-linking**: looking for matches within the same text or work;
- **Inter-linking**: looking for matches across multiple texts or works.

Let’s assume we have two different editions of the Bible, *Edition A* and *Edition B*. The difference between Intra-linking and Inter-linking can be visualised as follows:

*Figure 8.1: Left: Intra-linking identifies reuse within Edition A, for example between Mark and Luke. Right: Inter-linking identifies reuse between Edition A and Edition B.*
Remember that Edition A and Edition B are actually stored in the same .txt file, one under the other, as described in section 4. The Inter/Intra parameter can be changed in the TRACER tracer_config.xml:

![XML Configuration Example]

Figure 8.2: The value of the highlighted property in the TRACER tracer_config.xml can be changed to InterCorpusLinkingImpl, if needed.

TRACER outputs Linking results in a .link file in a three column structure:

```
REUSE ID 1 - REUSE ID 2 - OVERLAP
```

Here's how this structure looks like in the corresponding file:

![Three-column Structure Example]

Figure 8.3: The three-column structure in the Linking output file of the King James Version Bible text: REUSE ID 1 - REUSE ID 2 - ABSOLUTE OVERLAP

What does TRACER mean by absolute overlap? The absolute overlap is the minimal number of common features shared by the first two columns (REUSE ID 1 and REUSE ID 2), which TRACER sets as default to 5. This overlap number can be changed and is used to cut the long tail of reuses which would likely not be relevant matches or reuses at all. The .meta Linking file will provide you with an overview of the features linked:

![Feature Overlap Example]

Figure 8.4: Overview of Linking results provided by the Linking .meta file.

Linking is the most time-consuming step and the step that can be best parallelised, as shown in Figure 8.5 below.

---

10 The overlap is computed based on the preprocessing and selected features.
8.1.1 Moving Window Linking Implementation for short reuse

The two types of Linking implementation described above work well when linking text reuse units (e.g. sentences) of a similar size and for detecting close reuse or rewording (i.e. where the reuse covers roughly more than 50% of the unit). One example of where we could use these implementations is the Bible: we could, for instance, compare Mark with Luke. The reuse units, the verses, are similar in length and our knowledge of the Bible tells us that Luke copied from Mark. If, however, you’re looking for paraphrase or allusions, or if the reuse covers less than 50% of the unit, these two implementations will not perform well. That is not to say that you can’t use them. But if you do, you must lower the similarity thresholds in the `tracer_config.xml` file, and by doing so TRACER will contaminate your results with many false positives.

For these cases, TRACER provides another Linking implementation, the Moving Window. This approach is well-suited, if not necessary, for the detection of very small reuse; for example, to detect a four-word overlap in two sentences (as reuse units) of 20 and 25 words each. If we set a Moving Window of, for example, 10 words, TRACER will read the reuse unit 10 words at a time with a one-word overlap, as shown in the Latin example below:

![Figure 8.6: The reuse unit 100000801 is divided into 10-word chunks. The first chunk (on the first line) begins with the words 'patris tueretur, ab offenso', the second chunk (on the second line) skips 'patris' and begins with 'tueretur', the third chunk (on the third line) skips 'patris tueretur' and begins with 'ab', and so on and so forth.](image)

If you don’t know your data well and would like TRACER to give you a rough idea of the degree of similarity between two or multiple texts, the recommendation is to first run a detection task without the Moving Window implementation in order to detect the closest matches. Use the Moving Window only if the result-set or recall from this first analysis is too low and/or if you’d like to find smaller reuse.

**How to use the Moving Window**

To use the Moving Window implementation, you first need to segmentise your reuse units into windows. We recommend you try with windows of 10 or 15 words as these should detect most reuses. But you may, of course, run more than one detection with bigger or smaller window sizes and compare the results!

First, ensure your corpus `.txt` file is in TRACER’s `corpora` folder. Then, in the terminal, navigate to TRACER’s main folder and type this command (one line, which for formatting reasons doesn’t fit here):
java -cp tracer.jar eu.etrap.tracer.preprocessing.MovingAverageSegmentizerMain data/corpora/YourCorpus/yourcorpus.txt 15

Where the data/...yourcorpus.txt path points to the location of your text in TRACER’s data folder and 15 tells TRACER to split reuse units into windows of 15 words. Press ENTER and wait a few seconds for TRACER to compute the result. This command creates a version of the yourcorpus.txt where reuse units are formatted into 15-word windows. This file version is automatically named yourcorpus-W15.txt and saved in the TRACER corpora folder. If you want to break the text into windows of 10 words, change the final 15 to 10 in the java command.

Next, you need to make some changes in TRACER’s configuration file. First, make sure that the SENTENCE_FILE_NAME property points to your new yourcorpus-W15.txt file (as described in section 4.2). Next, locate the Linking property in the tracer_config.xml file and ensure you have the right class, MovingWindowInterCorpusLinking or MovingWindowIntraCorpusLinking (depending on whether you want to run Inter- or Intracorpus linking):

```
<category name="general">
    <property name="FRAMEWORK_IMPL" value="eu.etrap.tracer.DefaultFrameworkImpl"/>
    <property name="PREPROCESSING_IMPL" value="eu.etrap.tracer.preprocessing.WordLevelPreprocessingImpl"/>
    <property name="TRAINING_IMPL" value="eu.etrap.tracer.feature.RULEBasedTrainImpl"/>
    <property name="SELECTION_IMPL" value="eu.etrap.tracer.selection.GlobalLocalGlobalWordClassSelectorImpl"/>
    <property name="LINKING_IMPL" value="eu.etrap.tracer.linking.MovingWindowInterCorpusLinking"/>
    <property name="SCORING_IMPL" value="eu.etrap.tracer.scoring.feature.selected.SymmetricSelectedFeatureReusabilitySimilarityImpl"/>
</category>
```

Figure 8.7: The class of the LINKING_IMPL property in TRACER’s configuration file should read MovingWindowInterCorpusLinking or MovingWindowIntraCorpusLinking, depending on the type of Linking you’re interested in.

Next, define the length of your Window in the Linking category of the tracer_config.xml file, in the intWindowSize property:

```
<category name="eu.etrap.tracer.linking.MovingWindowInterCorpusLinking">
    <property name="feature2RUDDImplementation" value="eu.etrap.tracer.linking.connector.RUDDFeatureRAMConnectorImpl"/>
    <property name="intWindowOverlap" value="7"/>
    <property name="intWindowSize" value="15"/>
</category>
```

Figure 8.8: Define the size of your Moving Window in the value attribute of the intWindowSize property of the Linking category. In this case, we define a Window of 15 words.

The value of the intWindowSize property must match the window size you defined in the previous java command.

Save the changes and run TRACER!
The *Scoring* process of TRACER simply adds a fourth column to the previous *KJV-KJV.link* file (see Figure 8.3) with the *weighted overlap*, as opposed to the *absolute overlap* in the third column. The *weighted overlap* is based on a scoring metric and is used to balance out the similarities computed by TRACER. To calculate the *Resemblance* score, TRACER uses the formula:

\[
\theta_{O}^R(s_i, s_j) = \theta_{O}^L(s_i, s_j) \cdot \theta_{O}^L(s_i, s_j) = \frac{|S_i \cap S_j|}{|S_i \cup S_j|} \tag{9.1}
\]

**Brief explanation of the formula in the next manual release.**

And the resulting file will look something like:

![Figure 9.1: The fourth column in the KJV-KJV.scoring file provides the weighted overlap.](image)

If you look at the *SCORING* section of the *tracer_config.xml* file, you’ll notice three different classes you can choose from:

```xml
<category name="eu.eтратracer.scoring.word.full.symmetric.FullWordResemblanceSimilarityImpl">
  <property name="dbfScoringThreshold" value="6.0" />
</category>
<category name="eu.eтратracer.scoring.feature.selected.symmetric.SelectedFeatureWordClass2WeightedOverlapImpl">
  <property name="dbfScoringThreshold" value="20001.0" />
</category>
<category name="eu.eтратracer.scoring.feature.selected.symmetric.SelectedFeatureResemblanceSimilarityImpl">
  <property name="dbfScoringThreshold" value="4.9" />
</category>
```

*Figure 9.2: The three Scoring implementations in TRACER.*
Let's look at them in more detail:

**FullWordWeightedResemblanceSimilarityImpl**

This class computes similarities including *all words* in the sentence and applies the same weight to every word. This implementation was designed to deliver the results in a human-readable way (i.e. for GUIs). *Full-weighted* means that the similarity score is multiplied by the feature overlap in order to convey some form of confidence; the value of this property here is no longer a scale between 0 and 1 but the weight depends on the length of the segmented unit you chose.

**SelectedFeatureWordClass2WeightedOverlapImpl**

With this implementation selected features are transformed into word classes, and the word classes are weighed. More weight is given to content classes (verbs, nouns, etc.) than to, for instance, articles.

**SelectedFeatureResemblanceSimilarityImpl**

Given selected features from a reuse pair (max pruning without function words) this scoring implementation compares the selected features and gives all words the same weight. The value of this property can range between 0 and 1. If you set a high value, such as 0.9, TRACER will return only very close matches.
Postprocessing: Visualising the reuse results

TRACER integrates a variant graph visualisation tool called TRAViz\(^\text{11}\) in order to visualise the reuse results in a more legible format. TRAViz provides different types of visualisations for both close and distant reading, including dot-plots and text-alignments.

To view the computed results, navigate to the TRACER_DATA folder and locate the relevant results sub-folder therein. Clicking through the various sub-folders should eventually lead you to a TRAViz folder in which you’ll find an index.html file. Double-click on index.html and your browser should now show this:

![Text Re-use Visualizations](image)

*Figure 10.1: The index.html file in the TextReuseVisualization folder is the access point to the visualisations computed by TRACER/TRAVis.*

Here, you can explore three different visualisations of your results. The middle visualisation, the Text Reuse Browser, compares two works or texts only; with the third visualisation, the Text Reuse Alignment, you can compare multiple texts. Try opening the Text Re-use Browser.

\(^{11}\)TRAViz was developed and is maintained by Stefan Jänicke at the University of Leipzig.
Select your texts from the drop-down menus and click on Set texts. You should now see two visualisations. In both visualisations, the greener the colour the more similar the matches. The first visualisation is a dot-plot or distant reading view, typically used in plagiarism detection; on the Y axis we have the gospel of Mark and on the X axis Luke. The length of the axes correspond to the length of the work, and what we can immediately observe is that Mark gospel's shorter than Luke's. The numbers you see on the axes correspond to the IDs of the first and last sentences in each gospel. Each axes is made up of points, each corresponding to a sentence. The dots in the white area pinpoint similarities between the two texts and their colour the degree of similarity. The bottom plot offers a close reading exploration of the reuse alignments. The left pane gives you a macro-view, where the two grey rectangles represent Mark and Luke's gospels. The connecting lines join similar sentences and give you their exact location within the work. The right pane offers a micro-view of the data, allowing you to compare alignments.

Play with the visualisation and try to interpret the results!

10.1 Understanding TRACER's automatically-computed files

Every run of TRACER automatically produces a number of files, which can be viewed in the corpora directory. For example, running TRACER on the KJV.txt file will automatically produce the following files (also depending on the parameters set in the tracer_config.xml file, as already described in section 6.4):

KJV.txt.inv

inv stands for inverted list. This file works like a word index and is the heart of any retrieval system. It
Figure 10.3: Files contained in the Bible data folder of TRACER.

shows you that a specific word appears in a specific verse in a specific position.

<table>
<thead>
<tr>
<th>Word number</th>
<th>Verse ID</th>
<th>Position of word in verse</th>
</tr>
</thead>
<tbody>
<tr>
<td>114</td>
<td>4003870</td>
<td>2</td>
</tr>
</tbody>
</table>

\textit{KJV.txt.wnc}

\textit{wnc} stands for \textit{words number complete}. This file gives more information about the word types in the corpus including frequency, rank and word length. IDs are frequency-sorted.

\textit{KJV.txt.meta}

This file provides an overview or statistics, the \textit{metadata} as it were, of the corpus segmentation:

- **Sentences**: lines.
- **Word Types**: number of unique words as dictionary entries.
- **Word Tokens**: occurrences of a word; every word in a text, no matter how many times it occurs.
- **Sources**: for example, a book.
- **SSIM Threshold**: degree of similarity required to consider two words as similarly written.
- **SSIM Edges**: number of links or word pairs satisfying the similarity requirement stated in SSIM Threshold.
- **BOW Word Tokens**: tokens that appear in a line, but counting them as one even if they appear multiple times in the line e.g. ‘the’ appears 3 times in a line, but is counted as 1.
- **SSIM**: information is needed for the preprocessing step of TRACER. It’s basically the preprocessing of the preprocessing.

\textit{KJV.txt.tok}

This file is a \textit{tokenized} version of the source text. This means that all punctuation has been removed or separated from the word (depending on the settings). The default setting in TRACER is delete.
KJV.txt.tok.dist.char
This file displays the distribution of the characters across the corpus. This file is useful to clean up overlooked dirt in the text(s) under analysis. If the file contains an unexpected character, thanks to the .char information the user can more easily identify and remove it from the text(s).

KJV.txt.tok.dist.letter.02gram
This file displays the distribution of letter bigrams across the corpus.

KJV.txt.ssim
Everything comes together in this file. The first and second columns represent the two words; the third column is the overlap of letter bigrams; the fourth column is the weighted overlap $w$ between

\[ w \in [0, 1] \] (10.1)

by Broder’s Resemblance measure.

Brief explanation of this in the next manual release.
CHAPTER 11

Limitations of TRACER

TRACER’s detection results depend on the quality of the input .txt, .lemma and .syn files. The higher the performance of the morphological analysers and the higher the accuracy of the WordNets, the better the TRACER results.

The second limitation of TRACER concerns the visualisation of the results. While adequate for small data, TRAViz is not well-suited for visualising a large number of results (e.g. comparing 10 texts against one another). This is because the underlying code of TRAViz is not able to process and load large amounts of information. The current workaround to this issue is to put TRACER’s scores into gnuplots\(^\text{12}\) but that means you will lose the possibility of reading your texts as variant graphs.

\(^{12}\) For more information, see: http://www.gnuplot.info/
Q. How do I cite TRACER?
The first TRACER article in English is still under publication. Depending on your referencing style, you can adapt the following:


Q. Can I reuse TRACER's code in my work?
Yes. As per TRACER’s Academic Free License (see Copyright), you’re allowed to repurpose the source code as long as you credit the source, share your code in a similar fashion and don’t financially profit from it.

Q. Where can I find information about future TRACER tutorials?
The www.etrap.eu website always lists upcoming tutorials a few weeks in advance. These are typically run during conferences or summer schools.
Post-processing out-of-memory error
If TRACER generates an out-of-memory error in the post-processing step, the problem is likely caused by the scoring threshold set in the configuration file. If the scoring threshold is too low, such as 0.1, TRAViz needs more time and memory to compute the results, as a low scoring threshold leads to the production of more results. To solve the issue, increase the memory value contained in the TRACER execution command (see section 5.1) from Xmx600m to, for example, Xmx1g.

Empty .score and/or .link files (0KB)
It might happen that TRACER runs a detection successfully but produces empty .link and .score files. There are to potential reasons for this:

1. You might have the wrong value in the Linking implementation property: INTER instead of INTRA corpus linking (see text enclosed in asterisks):
   
   ```
   <property name="LINKING_IMPL" value="eu.etrap.tracer/linking.INtraCorpusLinkingImpl"/>
   ```

2. Your ID numbering might be incorrect. Again, in the Linking section of the configuration file, you'll notice the following property:
   
   ```
   <property name="intWorkNumbering" value="1000000"/>
   ```

   The value is set to a 7-digit ID. This means that your IDs should be 7-digits long. If you would rather work with shorter digits, you must change this value in the configuration file accordingly. If possible, please avoid using IDs above 2.000.000, as these require extra steps in order to be processed.
14.1 Recommendation: Keeping a TRACER log-book

Now that you’ve experimented with TRACER, you’ll realise how complex automatic text reuse detection can be. It takes more than one detection task to tease out reuses from your text; and each detection task can take different parameters. To help keep track of these experiments, we recommend you create a log-book to record each and every task with the respective result-set. That way, you’ll avoid duplicating tasks and can more easily identify the optimal settings for a particular detection task. For example, you might want to create a table like this:

<table>
<thead>
<tr>
<th>Task N</th>
<th>Preprocessing</th>
<th>Selection</th>
<th>Scoring</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>syn. replace</td>
<td>local max feat. freq.</td>
<td>sim. thresh. = 0.5</td>
<td>5 reuses</td>
</tr>
<tr>
<td></td>
<td>lemmatiz.</td>
<td>feat. dens. = 0.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Word-based
### Bigram
### Trigram

14.2 TRACER and MEDUSA

**Medusa** is TRACER’s *Preprocessing* (tokenisation) and *Featuring* component. Medusa is an NLP (Natural Language Processing) framework for high performance computing of co-occurrences and n-grams of lengths between 2 and 10 [Büc08]. Furthermore, Medusa tokenises texts and created inverted lists both for words and co-occurrences. Medusa’s speed is the result of a custom-built hash-system that brings together perfect and open hashing. *Perfect hashing* in terms of an Array hash is used for frequent co-occurrences or bigrams, while the open hashing is used for less frequent data by way of a bucket hashing. The RAM hash is split into separate buckets each containing individual data-stores. Once a buckets is completely filled the entire RAM hash is stored on disc and emptied in memory. After a complete run through the text the Array hash and all temporary hashes, written on disc, are merged into one persistent hash. With 1GB of memory Medusa reaches an average of about two accesses to the data structure to find the right location for the data-set. When increasing the memory this value can be further reduced.
15.1 Links

- English language statistics: Wordcount.org;
- Historical Text Reuse Detection Google Group;
- META-SHARE linguistic resources;
- NLTK Sentence Segmentation;
- Part of Speech taggers for various languages: TreeTagger;
- Historical Text Reuse Zotero Group;

15.2 Publications

More about TRACER and its applications:


15.3 External Reports

Reports written by TRACER users about TRACER:


