Orange Textable v1.3a4 documentation

Release 1.3a4

December 07, 2013

Contents

Textable is an add-on for Orange data mining software package. It enables users to build data tables on the basis of text data, by means of a flexible and intuitive interface. It offers in particular the following features:

- import text data from various sources
- apply systematic recoding operations
- apply analytical processes such as segmentation and annotation
- manually, automatically or randomly select unit subsets
- build concordances and collocation lists
- compute quantitative indices such as frequency and complexity measures

Textable was designed and implemented by LangTech Sarl on behalf of the department of language and information sciences (SLI) at the University of Lausanne.

CHAPTER 1

Installation

To install Orange Textable add-on for Orange from PyPi run:

```
pip install Orange-Textable
```

To install it from source code run:

```
python setup.py install
```

Orange Textable can also be installed directly from within Orange Canvas, using the Add-ons manager (menu **Options > Add-ons**).

CHAPTER 2

Getting started

This part of the documentation introduces the basic usage patterns of Orange Textable. It is meant to be read in the indicated order.

2.1 Segmentations

Segmentations are at the heart of Orange Textable. Start learning about them.

2.1.1 Strings, segments, and segmentations

The main purpose of Orange Textable is to build tables based on text strings. As we will see, there are several methods for importing text strings, the simplest of which is keyboard input using widget *Text Field* (see also *Keyboard input and segmentation display*). Whenever a new string is imported, it is assigned a unique identification number (called *string index*) and stays in memory as long as the widget that imported it.

Consider the following string of 16 characters (note that whitespace counts as a character too), and let us suppose that its string index is 1:

Character	*a*		*s*	*i*	*m*	*p*	*l*	*e*		*e*	*x*	*a*	*m*	*p*	*l*	*e*
Position	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

In this context, a *segment* is basically a substring of characters. Every segment has an *address* consisting of three elements:

1. string index
2. initial position within the string
3. final position

In the case of *a simple example*, address (1, 3, 8) refers to substring *simple*, (1, 12, 12) to character *a*, and (1, 1, 16) to the entire string. The substring corresponding to a given address is called the segment's *content*.

A *segmentation* is an ordered list of segments. For instance, segmentation ((1, 1, 1), (1, 3, 8), (1, 10, 16)) contains 3 word segments, ((1, 1, 1), (1, 2, 2), ..., (1, 16, 16)) contains 16 character segments, and ((1, 1, 16)) contains a single segment covering the whole string.

As shown by the word segmentation example, every character in the string needs not be included in a segment. Moreover, a single character may belong to several segments simultaneously, as in ((1, 1, 1), (1, 1, 8), (1, 3, 8), (1, 3,

16), (1, 10, 16), (1, 3, 8)). This also shows that the order of segments in a segmentation can diverge from the order of the corresponding substrings in the string. **Exercise 1:** What is the content of each of the 6 segments in the previous example? (*solution*)

In the previous examples, all the segments of a given segmentation refer to the same string. However, a segmentation may contain segments belonging to several distinct strings. Thus, if string *another example* has string index 2, segmentation ((2, 1, 7), (1, 3, 16)) is perfectly valid. **Exercise 2:** What is the content of the segments in the previous example? (*solution*)

In order to store segmentations and transmit them between widgets, Orange Textable uses the *Segmentation* data type. Aside from the segment addresses, this data type associates a *label* with each segmentation, i.e. an arbitrary string used to identify the segmentation among others. [1] **Solution to exercise 1:** *a*, *a simple*, *simple*, *simple example*, *example*, *simple* (in this order). (*back to the exercise*) **Solution to exercise 2:** *another*, *simple example*. (*back to the exercise*)

2.1.2 Keyboard input and segmentation display

Typing text in a *Text Field* widget is the simplest way to import a string in Orange Textable. This widget has no input connexions, and emits in output a segmentation containing a single segment whose address points to the entire string that was typed. This segmentation is assigned the label specified in the **Output segmentation label** field (see *figure 1* below):

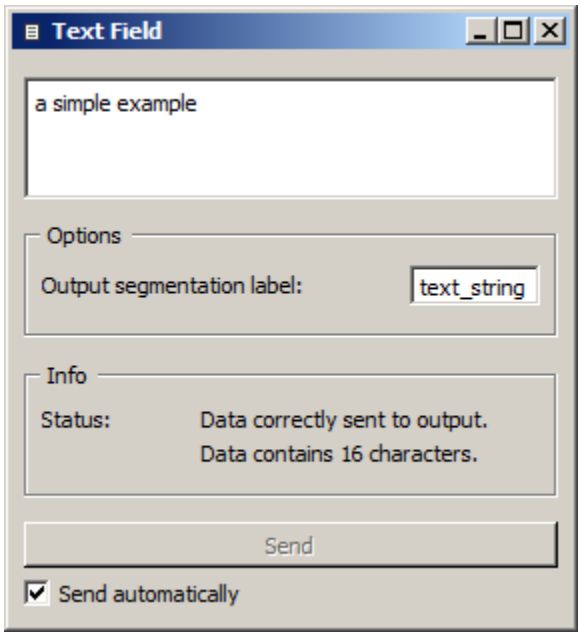


Figure 2.1: Figure 1: Typing *a simple example* in widget *Text Field*.

This widget's simplicity makes it most adequate for pedagogic purposes. Later, we will discover other, more powerful ways of importing strings.

The *Display* widget can be used to visualize the details of a segmentation. By default, it shows the segmentation's label followed by each successive segment's address and content. A segmentation sent by a *Text Field* instance will contain a single segment covering the whole string (see *figure 2* below).

By default, *Display* passes its input data without modification to its output connexions. It is very useful for viewing intermediate results in an Orange Textable scheme and making sure that other widgets process data as expected.

[1] As we will see *later*, the *Segmentation* data type can also store annotations associated with segments.

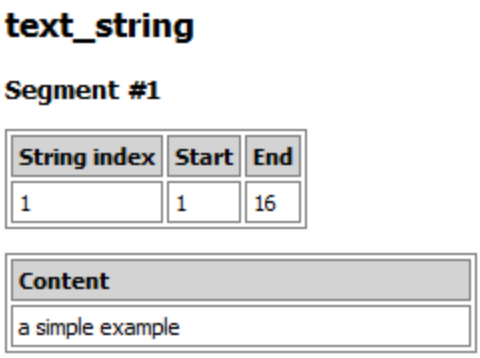


Figure 2.2: Figure 2: Viewing *a simple example* in widget *Display*.

2.1.3 Merging segmentations together

Computerized text analysis often implies consolidating various text sources into a single *corpus*. In the framework of Orange Textable, this amounts to grouping segmentations together, and it is the purpose of the *Merge* widget.

To try out this widget, create on the canvas two instances of *Text Field*, an instance of *Merge* and an instance of *Display* (see *figure 1* below). Type a different string in each *Text Field* instance (e.g. *a simple example* and *another example*) and assign it a distinct label (e.g. *text_string* and *text_string2*). Eventually, connect the instances as shown on *figure 1*.

The interface of widget *Merge* (see *figure 2* below) illustrates a feature shared by most Orange Textable widgets: the **Advanced settings** checkbox triggers the display of more complex controls offering more possibilities to the user. For now we will stick to the basic settings and leave the box unchecked.

Section **Ordering** of the widget's interface lets the user view the labels of incoming segmentations and control the order in which they will appear in the output segmentation (by selecting them and clicking on **Move Up / Down**). The **Output segmentation label** can be set in section **Options**. We will return *later* to the purpose of checkbox **Import labels with key**; leave it unchecked for now.

Figure 3 above shows the resulting merged segmentation, as displayed by widget *Display*. As can be seen, *Merge* makes it easy to concatenate several strings into a single segmentation. If the incoming segmentations contained several segments, each of them would appear in the output segmentation, in the order specified under **Ordering** (and, within each incoming segmentation, in the original order of segments). **Exercise:** Can you add a new instance of *Merge* to the scheme illustrated on *figure 1* above and modify the connections (but not the configuration of existing widgets) so that the segmentation given in *figure 4* below appears in the *Display* widget? (*solution*) **Solution:** (*back to the exercise*)

2.1.4 A note on regular expressions

Orange Textable widgets rely heavily on *regular expressions* (or *regexes*), which are essentially a body of conventions for describing a set of strings by means of a single string. These conventions are widely documented in books and on the Internet, so we will not give here yet another introduction to this topic. Nevertheless, a basic knowledge of regexes is required to perform any non-trivial task with Orange Textable, and more advanced knowledge to fully exploit the software's possibilities.

The syntax of regexes is partly standardized, but some variations remain. Orange Textable uses Python regexes, for which Python documentation is the best source of information. In particular, it features a good introduction to regexes. A first reading might be limited to the following sections:

- Simple Patterns
- More Metacharaters

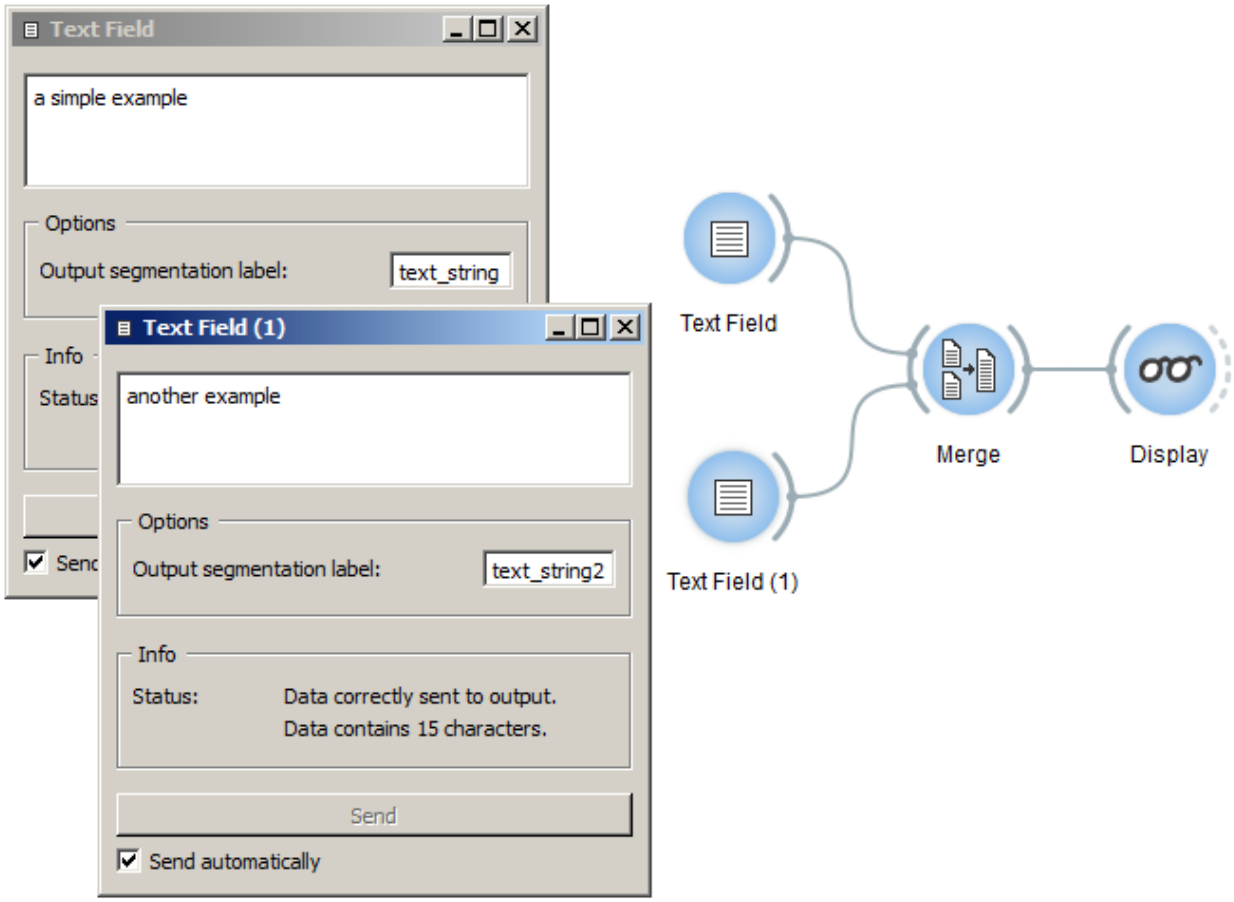


Figure 2.3: Figure 1: Grouping *a simple example* with *another example* using widget *Merge*.

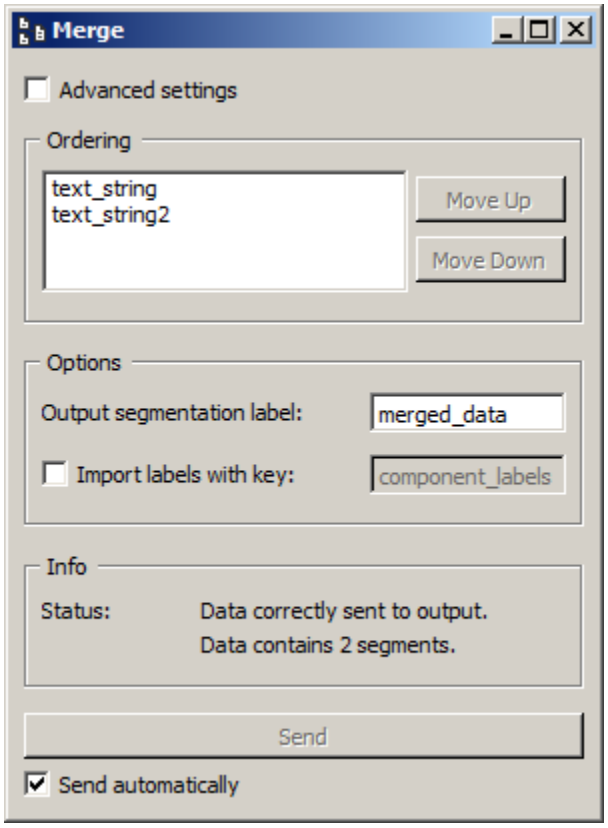


Figure 2.4: Figure 2: Interface of widget *Merge*.

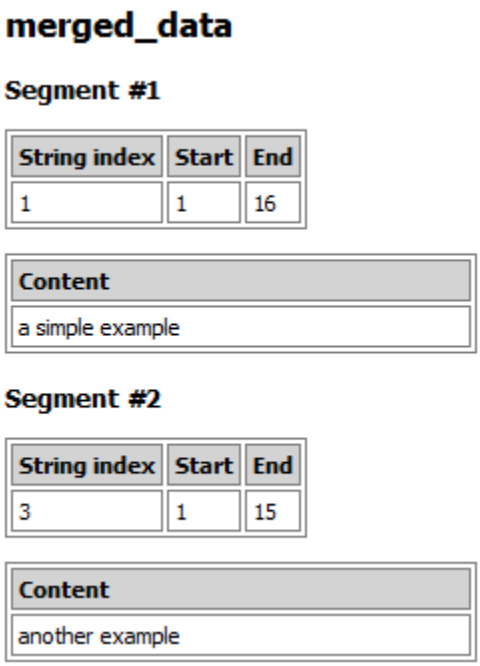
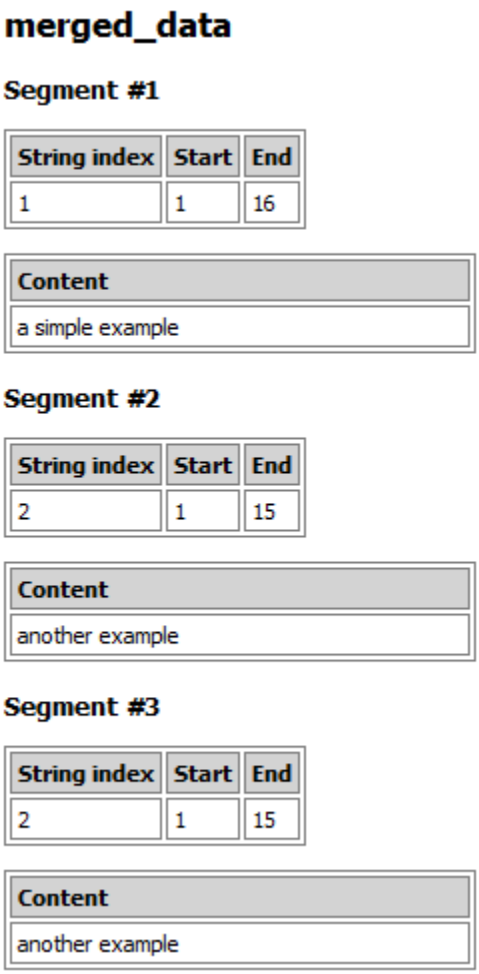


Figure 2.5: Figure 3: Merged segmentation.



merged_data

Segment #1

String index	Start	End
1	1	16

Content
a simple example

Segment #2

String index	Start	End
2	1	15

Content
another example

Segment #3

String index	Start	End
2	1	15

Content
another example

Figure 2.6: Figure 4: The segmentation requested in the *exercise*.

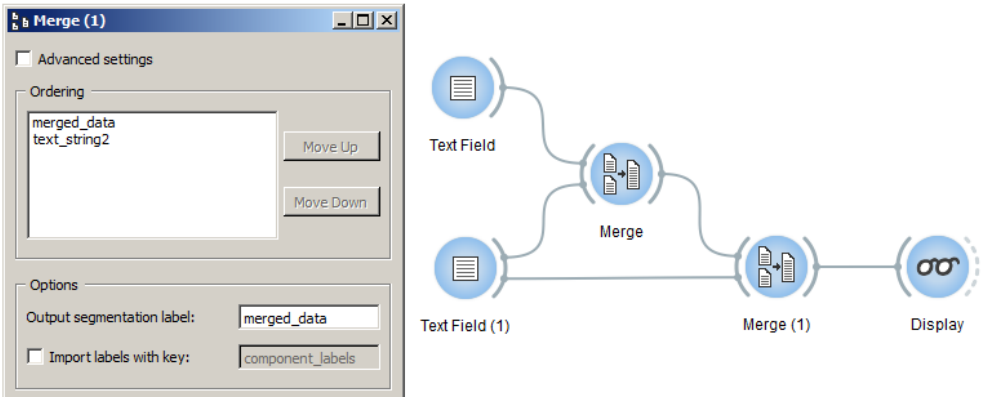


Figure 2.7: Figure 5: Solution to the *exercise*.

Also recommended are the following:

- Compilation Flags
- Lookahead Assertions
- Greedy vs. Non-Greedy

2.1.5 Segmenting data into smaller units

We have seen *previously* how to combine several segmentations into a single one. We will often be performing the inverse operation: create a segmentation whose segments are *parts* of another segmentation's segments. Typically, we will be segmenting strings into words, characters, or any kind of text units that will be later counted, measured, and so on. This is precisely the purpose of widget *Segment*.

To try it out, create a new scheme with an instance of *Text Field* connected to an instance of *Segment*, itself connected to an instance of *Display* (see *figure 1* below). In what follows, we will suppose that the string typed in *Text Field* is *a simple example*.

Figure 2.8: Figure 1: A scheme for testing the *Segment* widget

In its basic form (i.e. with **Advanced settings** unchecked, see *figure 2* below), *Segment* takes a single parameter (aside from the **Output segmentation label**), namely a regex. The widget then looks for all matches of the regex pattern in each successive input segment, and creates for every match a new segment in the output segmentation.

For instance, the regex *\w+* divides each incoming segment into sequences of alphanumeric character (and underscore)–which in our case amounts to segmenting *a simple example* into three words. To obtain a segmentation into letters (or to be precise, alphanumeric characters or underscores), simply use *\w*.

Of course, queries can be more specific. If the relevant unit is the word, regexes will often use the *\b anchor*, which represents a word boundary. For instance, the words that contain less than 4 characters can be retrieved with *\b\w{1,3}\b*, those ending in *-tion* with *\b\w+tion\b*, and the flexion of *retrieve* with *\bretriev(e|es|ed|ing)\b*.

2.1.6 Hierarchical segmentations and performance issues

When widget *Segment* is applied to real, much longer texts than *a simple example*, using such general regexes as *\w+* or *\w* may result in the creation of a huge number of segments. Creating and manipulating such segmentations can slow down excessively the execution of Orange Textable, or even lead to memory overflow.

However, it is sometimes necessary to segment large texts into words or letters, for instance in order to examine their frequency distribution. In that case, if hardware allows it, a lot of time can be saved at the expense of memory usage. Indeed, the cumulated time required to successively create several ever more fine-grained segmentations (for instance into lines, then words, then letters) is usually spectacularly shorter than the time required to produce the most fine-grained segmentation directly (see *figure 1* below).

The situation is different when word or letter segmentation are conceived as intermediate steps toward the creation of a segmentation containing only selected words or letters. In that case, it is much more efficient (in memory and execution time) to use a single instance of *Segment* with a regex identifying only the desired words, as seen *previously* with the example of *\bretriev(e|es|ed|ing)\b*.

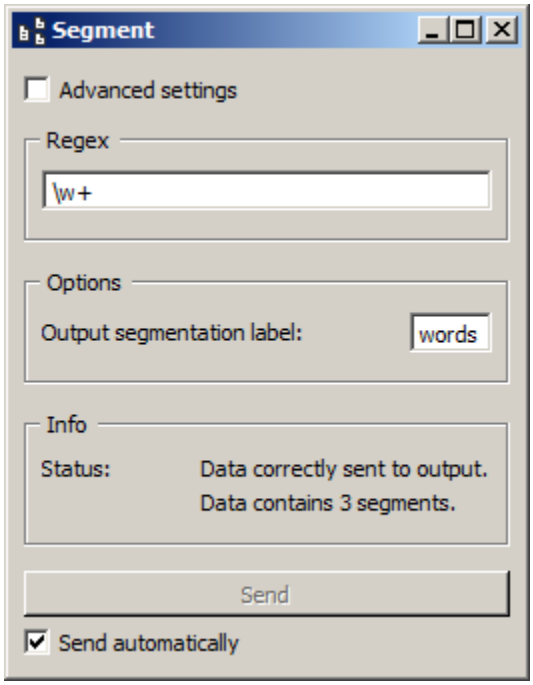


Figure 2.9: Figure 2: Interface of the *Segment* widget, configured for word segmentation

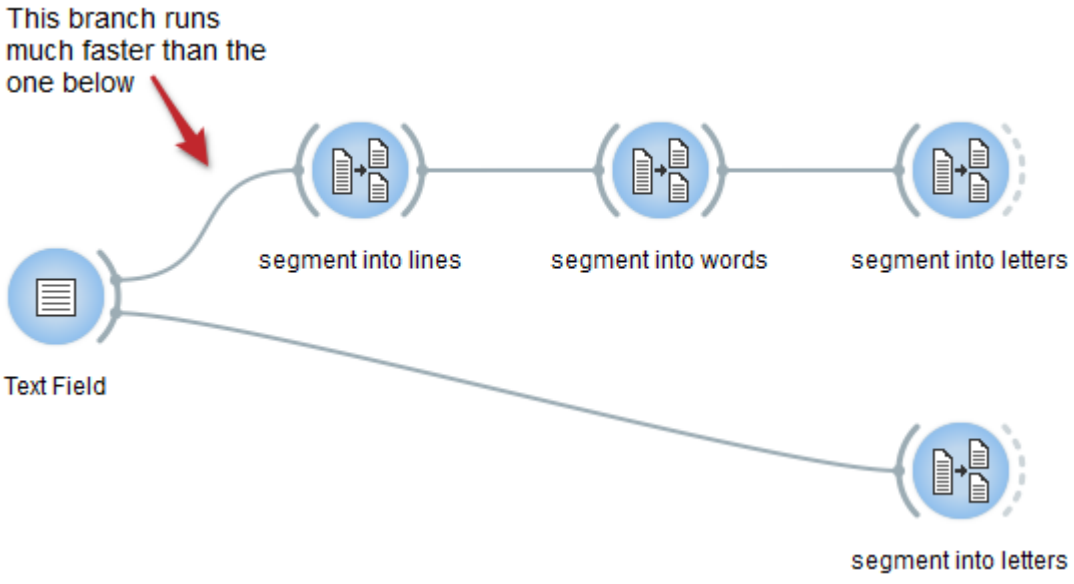


Figure 2.10: Figure 1: Chaining *Segment* instances to reduce execution time.

2.1.7 Partitioning segmentations

There are many situations where we might want so selectively in- or exclude segments from a segmentation. For instance, a user might be want to exclude from a word segmentation all those that are less than 4 letters long. The *Select* widget is tailored for such tasks.

The widget's interface (see *figure 1* below) offers a choice between two modes: *Include* and *Exclude*. Depending on this parameter, incoming segments that satisfy a given condition will be either included in or excluded from the output segmentation. By default (i.e. when the **Advanced settings** box is unchecked), the condition is specified by means of a regex, which will be applied to each incoming segment successively. (For now, the option **Annotation key: (none)** can be ignored.)

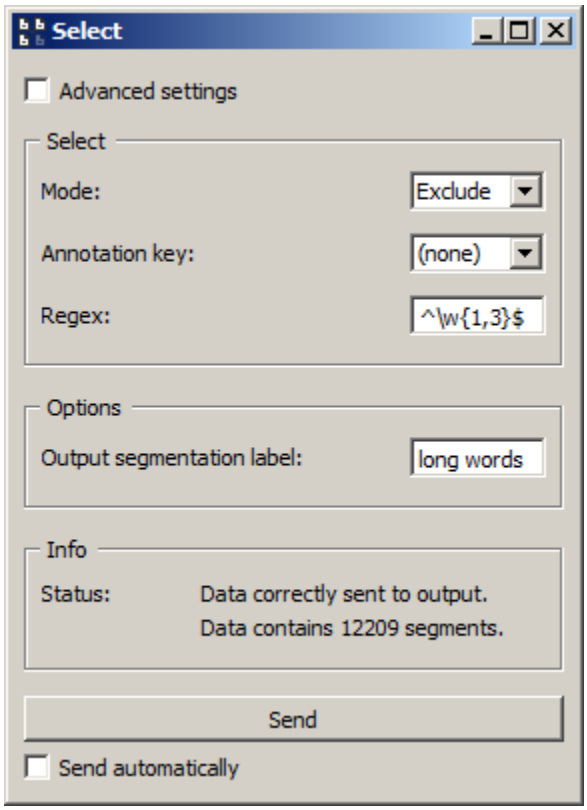


Figure 2.11: Figure 1: Excluding short words with widget *Select*.

In the example of *figure 1*, the widget is configured to exclude all incoming segments containing no more than 3 letters. Note that without the *beginning of segment* and *end of segment* anchors (^ and *$*), all words containing *at least* a sequence of 1 to 3 letters–i.e. all the words–would be excluded.

Note that *Select* automatically emits a second segmentation containing all the segments that have been discarded from the main output segmentation (in the case of *figure 1* above, that would be all words less than 4 letters long). This feature is useful when both the selected *and* the discarded segments are to be further processed on distinct branches. By default, when *Select* is connected to another widget, the main segmentation is being emitted. In order to send the segmentation of discarded segments instead, right-click on the outgoing connexion and select **Reset Signals** (see *figure 2* below).

This opens the dialog shown on *figure 3* below, where the user can "drag-and-drop" from the gray box next to **Discarded data** up to the box next to **Segmentation**, thus replacing the existing green connexion. Clicking **OK** validates the modification and sends the discarded data through the connexion.

Figure 2.12: Figure 2: Right-clicking on a connexion and requesting to **Reset Signals**.

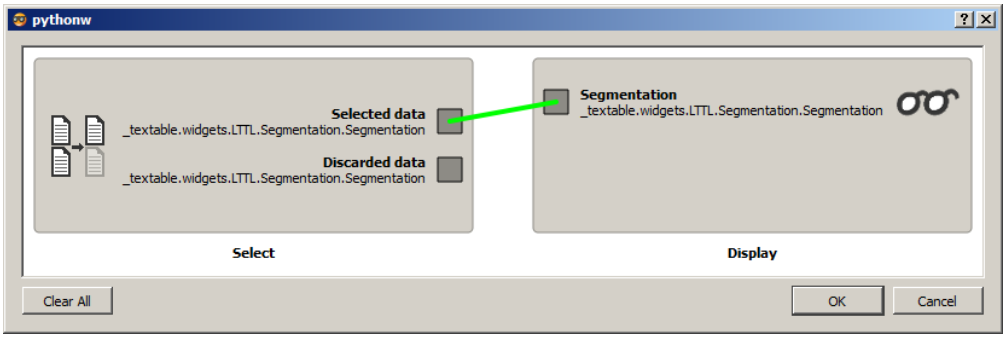


Figure 2.13: Figure 3: This dialog allows the user to select a non-default connexion between two widgets.

2.1.8 Using a segmentation to filter another

In some cases, the number of forms to be selectively included in or excluded from a segmentation is too large for using the *Select* widget. A typical example is the removal of "stopwords" from a text: in English for instance, although the list of such words is finite, it is too long to try to encode it by means of a regex (cf. an example of such a list).

The purpose of widget *Intersect* is precisely to solve that kind of problem. It takes two segmentations in input and lets the user include in or exclude from the first (*source*) segmentation those segments whose content is the same as that of a segment in the second (*filter*) segmentation. The widget's basic interface is shown on *figure 1* below).

Similarly to widget *Select*, user must choose between modes **Include** and **Exclude**. The next step is to specify which incoming segmentation plays the role of the **Source** segmentation and the **Filter** segmentation. (Here again, we will ignore the **Annotation key** option for the time being.)

In order to try out the widget, set up a scheme similar to the one shown on *figure 2* below). The first instance of *Text Field* contains the text to process (for instance the Universal Declaration of Human Rights), while the second instance, *Text Field (1)*, contains the list of English stopwords mentioned above. Both instances of *Segment* produce a word segmentation with regex \w+; the only difference in their configuration is the output segmentation label , i.e. *words* for *Segment* and *stopwords* for *Segment (1)*. Finally, the instance of *Intersect* is configured as shown on *figure 1* above.

The content of the first segments of the resulting segmentation is:

> *PREAMBLE Whereas recognition inherent dignity equal inalienable rights members human family foundation freedom justice peace world ...*

Exercise: Based on an instance of *Text Field*, produce a segmentation containing all words less than 4 letters long that appear at the beginning of each line, excluding *I, you, he, she, we*. (*solution*) **Solution:**

Figure 3 below shows a possible solution. The 4 instances in the lower part of the scheme (*Text Field (1)*, *Segment (1)*, *Intersect*, and *Display*) are configured as in *figure 2* above–with *Text Field (1)* containing the list of pronouns to exclude.

The difference lies in the addition of a *Segment* instance in the upper branch. In this branch, the first instance (*Segment*)

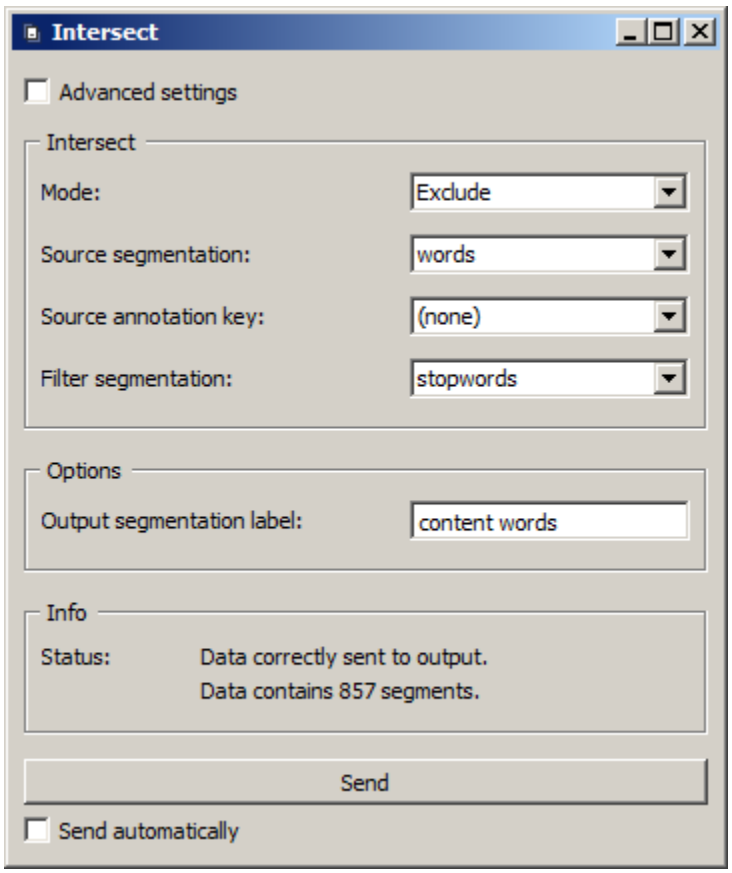


Figure 2.14: Figure 1: Interface of widget *Intersect* configured for stopword removal.

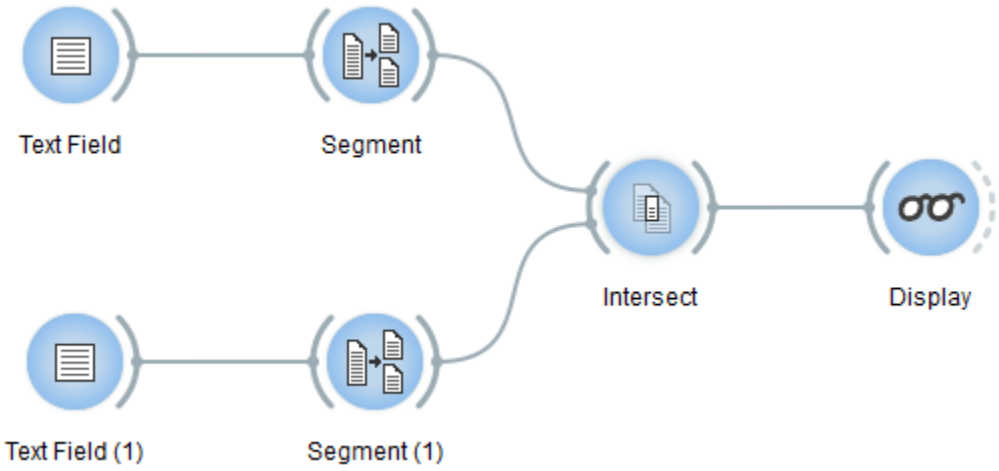


Figure 2.15: Figure 2: Example scheme for removing stopword using widget *Intersect* .

produces a segmentation into lines with regex .+ while *Segment (2)* extracts the first word of each line, provided it is shorter than 4 letters (regex ^\w{1,3}\b*). *Intersect* eventually takes care of excluding the pronouns listed above.

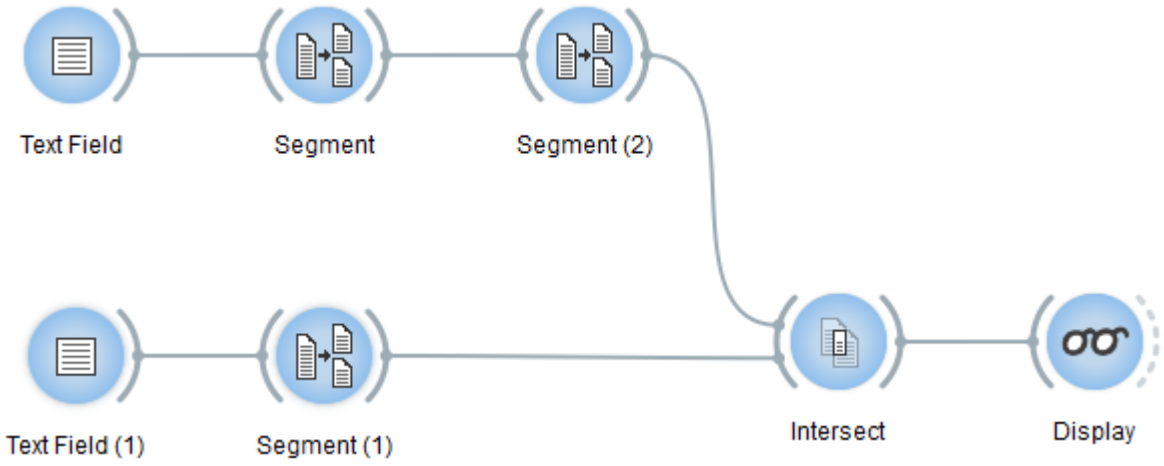


Figure 2.16: Figure 3: A possible solution.

(*back to the exercise*)

2.2 Tables

Segmentations are to tables what a means is to an end. In this section, you will learn how to go from the ones to the others.

2.2.1 From segmentations to tables

The main purpose of Orange Textable is to build tables based on texts. Central to this process are the segmentations we have learned to create and manipulate earlier. Indeed, Orange Textable provides a number of *widgets for table construction*, and they all operate on the basis of one or more segmentations.

For the time being, we will focus on the construction of frequency tables, which are very common in computerized text analysis and which will serve as introduction to other types of tables. For the sake of simplicity, consider first the segmentation of *a simple example* into letters. Counting the frequency of each letter type yields a table such as the following:

Table 2.1: Table 1: Frequency of letter types.

a	***s***	***i***	***m***	***p***	***l***	***e***	***x***
2	1	1	2	2	2	3	2

More often, we will be interested in comparing frequency across several *contexts*. For instance, if the word segmentation of *a simple example* is also available, it may be used together with the letter segmentation to produce a so-called *contingency table* (or *document–term matrix*):

Table 2.2: Table 2: Frequency of letters within words.

	a	***s***	***i***	***m***	***p***	***l***	***e***	***x***
a	1	0	0	0	0	0	0	0
simple	0	1	1	1	1	1	1	0
example	1	0	0	1	1	1	2	1

In a real application, rows could correspond to the writings of an author and columns to selected prepositions, for instance. The general idea is to determine the number of occurrences of various *units* in various *contexts*. Such data can then further analyzed by means of a statistical test (aiming at answering the question "does the distribution of units depend on contexts") or a graphical representation (making it possible to visualize the attraction or repulsion between specific units and contexts).

2.2.2 Converting between table formats

Orange Canvas has a "native" type for representing data tables, namely *ExampleTable*. However, this type does not support Unicode well, which is a serious limitation in the perspective of text processing. To overcome this issue (as much as possible), Orange Textable defines its own table representation format, simply called *Table*.

Every *table construction widget* in Orange Textable emits data in *Table* format. Instances of these widget must then be connected with an instance of *Convert*, which has mainly two purposes:

- It *converts* data in Orange Textable's *Table* format to the native *ExampleTable* format of Orange Canvas, which makes it possible to use the other widgets of Orange Canvas for visualizing, modifying, analyzing, etc. tables built with Orange Textable.
- It *exports* data in *Table* format to text files, in tab-delimited format, typically in order to import them later in a third party data analysis software package; at the time of writing, this scenario is the only way to correctly visualize a table containing data encoded in Unicode.

As shown on *figure 1* below, section *Conversion* of the widget's interface lets the user choose the encoding of the *ExampleTable* object produced in output (**Orange table encoding**); variants of Unicode should be avoided here since they are currently not well supported by other widgets in Orange Canvas.

The encoding for text file export can be selected in the *Export* section (**Output file encoding**); in this case there are no counter-indications to the use of Unicode. Checkbox **Include Orange headers** triggers inclusion of additional table headers in the case where the output file should be later re-imported in Orange Canvas. Export proper is performed by clicking the **Export** button and selecting the output file in the dialog that appears.

The take-home message here is this: when you create an instance of a *table construction widget*, you may systematically create a new instance of *Convert* and connect them together. Usually, moreover, you will want to connect the *Convert* instance to a *Data Table* instance (from the *Data* tab of Orange Canvas) in order to view the table just built–except in the case where it contains Unicode data that wouldn't display correctly in *Data table*.

2.2.3 Counting segment types

Widget *Count* takes in input one or more segmentations and produces frequency tables such as tables 1 and 2 *here*. To try it out, create a scheme such as illustrated on *figure 1* below. As usual, we will suppose that the *Text Field* instance contains *a simple example*. The *Segment* instance is configured for letter segmentation (**Regex:** *\w* and **Output segmentation label:** *letters*). The default configuration of the instances of *Convert* and *Data Table* (from the **Data** tab of Orange Canvas) needs not be modified for this example.

Basically, the purpose of widget *Count* is to determine the frequency of segment types in an input segmentation. The label of that segmentation must be indicated in the **Segmentation** menu of section **Units** in the widget's interface,

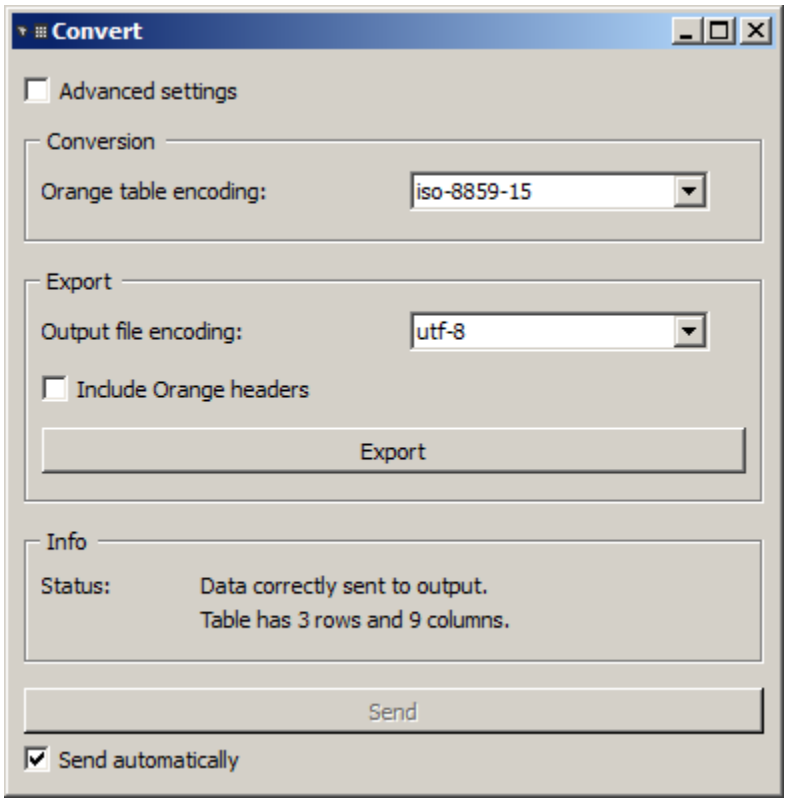


Figure 2.17: Figure 1: Basic interface of widget *Convert*.

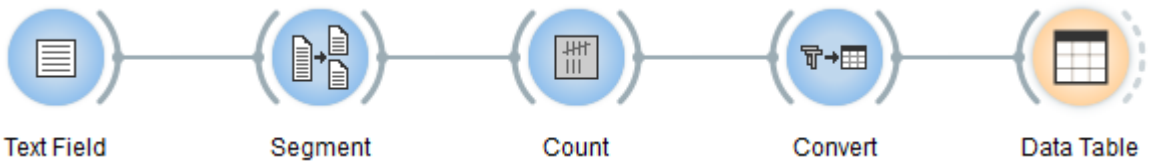


Figure 2.18: Figure 1: Scheme for testing the *Count* widget.

while other controls may be left in their default state for now (see *figure 2* below). Clicking **Compute** then double-clicking the *Data Table* instance should display essentially the same data as table 1 *here* (with possible variations in the order of columns).

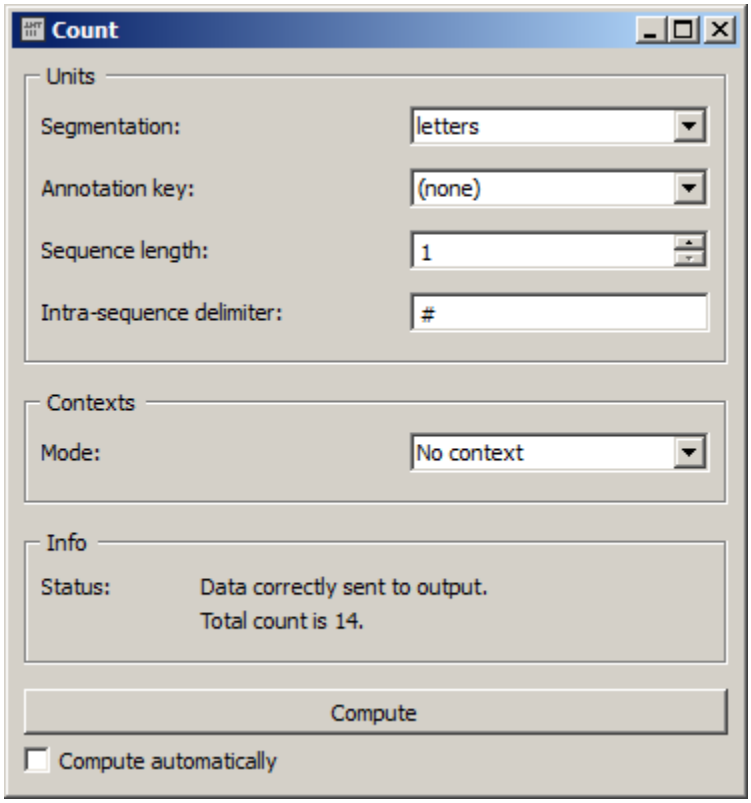


Figure 2.19: Figure 2: Counting the frequency of letter types with widget *Count*.

Note that checkbox *Compute automatically* is unchecked by default so that the user must click on **Compute** to trigger computations. The motivation for this default setting is that *table construction widgets* can be quite slow when operating on large segmentations, and it can be annoying to see computations starting again whenever an interface element is modified.

To obtain the frequency of letter *bigrams* (i.e. pairs of successive letters), simply set parameter **Sequence length** to 2 (see *table 1* below). If the value of this parameter is greated than 1, the string specified in field **Intra-sequence delimiter** is inserted between successive segments for the sake of readability–which is more useful when segments are longer than individual letters. Note that in this example, word boundaries are not taken into account–nor even known, in fact–which is why bigrams *as* and *ee* have a nonzero frequency.

Table 2.3: Table 1: Letter bigram frequency.

as	***si***	***im***	***mp***	***pl***	***le***	***ex***	***xa***	***am***
1	1	1	2	2	2	1	1	1

2.2.4 Counting in specific contexts

In preparation.

2.3 Annotations

Annotations let you go beyond what's in the text.

2.3.1 Annotations and their uses

In preparation.

2.3.2 Annotating by merging

In preparation.

2.3.3 Converting XML markup to annotations

In preparation.

2.3.4 Exploiting annotations

In preparation.

CHAPTER 3

Widget reference

This part of the documentation explains the effect of every control of each Orange Textable widget.

Orange Textable introduces mainly two new data types in Orange Canvas: *Segmentation* and *Table*. Widgets making up Orange Textable are grouped into 4 main categories based on their input and output data types:

3.1 Text import widgets

Widgets of this category take no input and emit *Segmentation* data. Their purpose is to import text data in Orange Canvas, either from the keyboard (*Text Field*), from files (*Text Files*), or from the Internet (*URLs*).

3.1.1 Text Field

In preparation.

3.1.2 Text Files

In preparation.

3.1.3 URLs

In preparation.

3.2 Segmentation processing widgets

Widgets of this category take *Segmentation* data in input and emit data of the same type. Some of them (*Preprocess* and *Recode*) generate modified text data. Others (*Merge*, *Segment*, *Select*, *Intersect* and *Extract XML*) do not generate new text data but only new *Segmentation* data. *Display*, finally, is mainly used to visualize the details of a given *Segmentation* object (content and address of segments, as well as their possible annotations).

3.2.1 Preprocess

In preparation.

3.2.2 Recode

In preparation.

3.2.3 Merge

In preparation.

3.2.4 Segment

In preparation.

3.2.5 Select

In preparation.

3.2.6 Intersect

In preparation.

3.2.7 Extract XML

In preparation.

3.2.8 Display

In preparation.

3.3 Table construction widgets

Widgets of this category take *Segmentation* data in input and emit *Table* data. They are thus ultimately responsible for converting text to tables, either by counting items (*Count*), by measuring their length (*Length*), by quantifying their diversity (*Variety*), or by exploiting the annotations associated with them (*Annotation*). Finally, widget *Context* makes it possible to build concordances and collocation lists.

3.3.1 Count

In preparation.

3.3.2 Length

In preparation.

3.3.3 Variety

In preparation.

3.3.4 Annotation

In preparation.

3.3.5 Context

In preparation.

3.4 Table conversion/export widget

The only widget in this category, *Convert*, takes *Table* data in input and emits *ExampleTable* data for further processing with Orange Canvas. It also makes it possible to apply various standard transforms to a table, such as sorting, normalizing, etc., as well as to export its contents to a file.

3.4.1 Convert

In preparation.

CHAPTER 4

Cookbook

In preparation.

CHAPTER 5

Case studies

In preparation.