Orange3-Textable Documentation

Release 3.1.10

LangTech Sarl

May 04, 2021

Contents

Welcome to the documentation of Orange Textable.

This documentation is divided in five main sections (see detailed *contents* below):

- The *Introduction* offers a brief overview of what Orange Textable can do, as well as how it should be installed and configured. This is what you should read first if you are unsure whether Orange Textable is the right piece of software for your needs or how to set it up.
- Section *Textable's basics* is a tutorial that introduces the basic concepts underlying Orange Textable and its main usage patterns. This should be your first reading once you've determined that Orange Textable can be useful to you and installed it.
- Section *Advanced topics* enables the advanced user to benefit from more complex text queries using regex and xml markups. This part implies a solid knowledge of the above Basics section.
- In the *Cookbook* section, you'll find a number of concise, illustrated recipes describing how to perform various basic tasks with Orange Textable. When starting a new project, you might want to skim through this section in case some elementary operation you need is listed there.
- Section *Case studies* presents several illustrations of the application of Orange Textable to more complex and interesting problems in text data analysis.
- The *Reference* is an exhaustive explanation of the role and effect of every component of Orange Textable's interface. The purpose of this part of the documentation is to help you find a specific piece of information about Orange Textable's operation when using it for your own projects.

CHAPTER 1

Contents

1.1 Introduction

Orange Textable is an open-source add-on bringing advanced text-analytical functionalities to the Orange Canvas data mining software package (itself open-source). It essentially enables users to build data tables on the basis of text data, by means of a flexible and intuitive interface. Look at the following *example* to see it in typical action.

Orange Textable was designed and implemented by LangTech Sàrl on behalf of the department of language and information sciences (SLI) at the University of Lausanne (see *Credits* and *How to cite Orange Textable*).

1.1.1 Features

Orange Textable offers the following features:

- text data import from keyboard, files, or urls
- support for various encodings, including Unicode
- standard preprocessing and custom recoding (based on regular expressions)
- segmentation and annotation of various text units (letters, words, etc.)
- ability to extract and exploit XML-encoded annotations
- automatic, random, or arbitrary selection of unit subsets
- unit context examination using concordance and collocation tables
- calculation of frequency and complexity measures
- recoded text data and table export

1.1.2 Illustration: mining Humanist

The following example is meant to show *what* Orange Textable typically does, without considering (for now) every detail of *how* it does it.

In a paper reflecting on terminology in the field of Digital Humanities[1], Patrik Svensson compares the evolution of the frequency of expressions *Humanities Computing* and *Digital Humanities* over 20 years of archives of the Humanist discussion group. He uses these figures to show that while the former denomination remains prevalent over these two decades, the latter has been quickly gaining ground since the 2000s.

The same experiment can be run with Orange Textable, by building a "visual program" like the one shown on *figure 1* below:

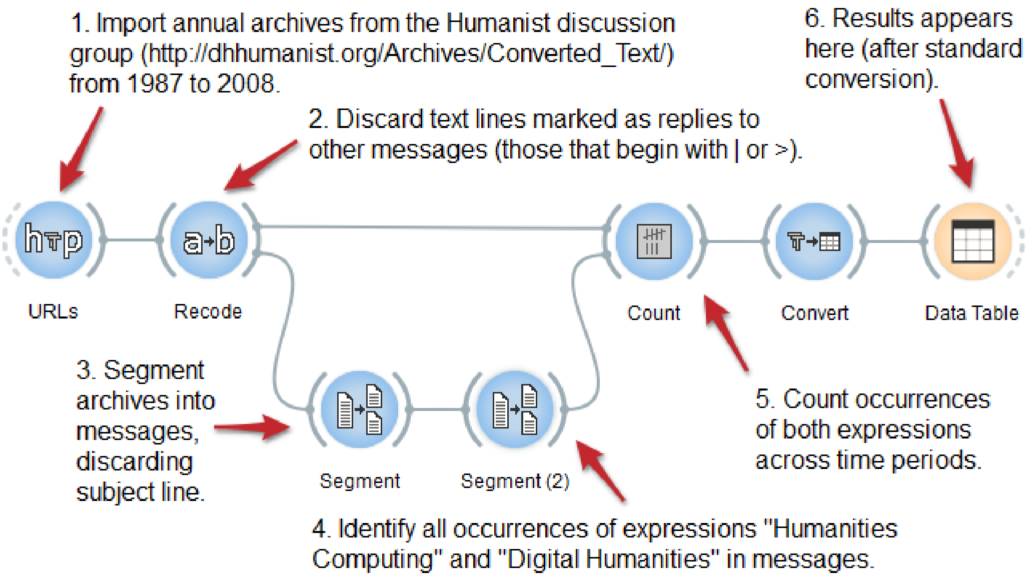

1. Import annual archives from the Humanist discussion group (http://dhhumanist.org/Archives/Converted_Text/) from 1987 to 2008.
2. Discard text lines marked as replies to other messages (those that begin with | or >).
6. Results appears here (after standard conversion).
URLs
Recode
Count
Convert
Data Table
3. Segment archives into messages, discarding subject line.
Segment
Segment (2)
5. Count occurrences of both expressions across time periods.
4. Identify all occurrences of expressions "Humanities Computing" and "Digital Humanities" in messages.

Fig. 1: Figure 1: Mining Humanist with an Orange Textable schema.

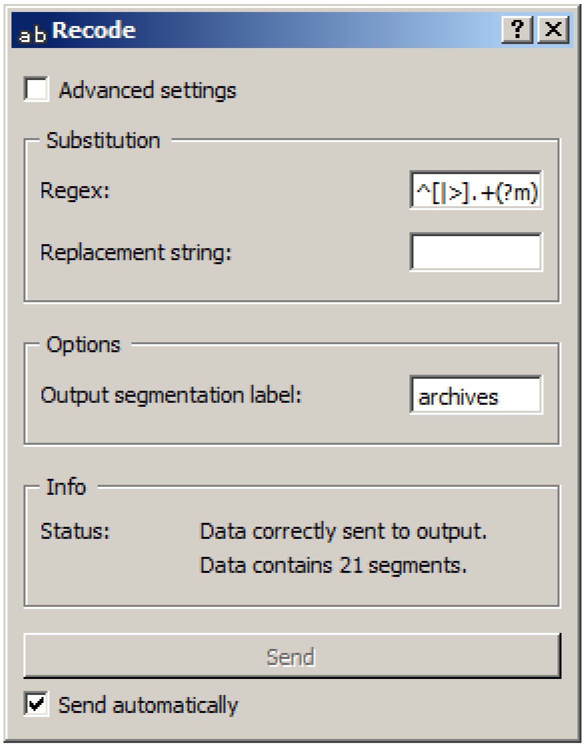
Such a program is called a *schema*. Its visible part consists of a network of interconnected units called *widget instances*. Each instance belongs to a type, e.g. *URLs*, *Recode*, *Segment*, and so on. Widgets are the basic blocks with which a variety of text analysis applications can be built. Each corresponds to a fundamental operation, such as "import data from an online source" (*URLs*) or "replace specific text patterns with others" (*Recode*) for example. Connections between instances determine the flow of data in the schema, and thus the order in which operations are carried on. Several parallel paths can be constructed, as demonstrated here by the *Recode* instance, which sends data to *Segment* as well as *Count*.

Widget instances can (and indeed must) be individually parameterized in order to "fine-tune" their operation. For example, double-clicking on the *Recode* instance of *figure 1* above displays the interface shown on *figure 2* below. What this particular configuration means is that every line beginning with symbol `|` or `>` (**Regex** field) should be replaced with an empty string (**Replacement string**): in other words, remove those lines that are marked as being part of a reply to another message. There is a fair amount of variation between widget interfaces, but regular expressions play an important role in many of them and Orange Textable's flexibility owes a lot to them.

After executing the schema of *figure 1* above, the resulting frequencies can be viewed by double-clicking on the **Data Table** instance, whose interface is shown on *figure 3* below. On the whole, these figures lend themselves to the same interpretation as that of Patrik Svensson, but they differ wildly from the frequencies he reports. This might be explained by the fact that, in the present illustration, we have used *preprocessed* data made available on the Humanist website, or it might be that we have not processed the data exactly like Svensson did. The user can always refer to the Orange Textable schema (including the parameters of each instance) to understand exactly the operations that it performs.[2] In

[1] Svensson, P. (2009). Humanities Computing as Digital Humanities. *Digital Humanities Quarterly 3(3)*. Available here.

[2] The schema can be downloaded from `here`. Note that two decades of Humanist archives weigh dozens of megabytes and that retrieving these data from the Internet can take a few minutes depending on bandwidth. Please be patient if Orange Textable appears to be stalled when the schema is being opened.



□ Advanced settings

Regex: ^[|>].+(?m)

Replacement string:

Output segmentation label: archives

Status: Data correctly sent to output.
Data contains 21 segments.

☑ Send automatically

Fig. 2: Figure 2: Interface of the *Recode* widget.

this sense, Orange Textable does not only attempt to make the construction of text analysis programs easier; it aims to make *communicating* and *understanding* such programs easier.

1.1.3 Installation

Python v2.7 and Orange Canvas v2.7 must imperatively be installed *before* Orange Textable. **Please note that Orange Textable is not compatible with Orange 3 at the time of writing.** After installation, Orange Textable appears in the form of an additional tab in Orange Canvas.

The installation procedure is slightly different on Windows and MacOS X.[1]

Windows installation

1. On the Orange 2.7 download page, download the software installer by following the *Orange 2.7 installer for Windows* link.
2. Execute the Orange Canvas installer and click **Ok** at each stage (including the stages of the installation of Python modules).
3. Start Orange Canvas then select menu **Options > Add-ons…** (see *figure 1*).
4. In the window which has opened (see *figure 2*), click on **Refresh list**, check the *Orange-Textable* box then the **Ok** button (twice).

If step 4 was carried out correctly, the Orange Textable tab appears in the list on the left of the window of Orange Canvas after having exited and restarted the program.

[1] Although several users have reported successful installation on Linux, it has not been specifically tested.

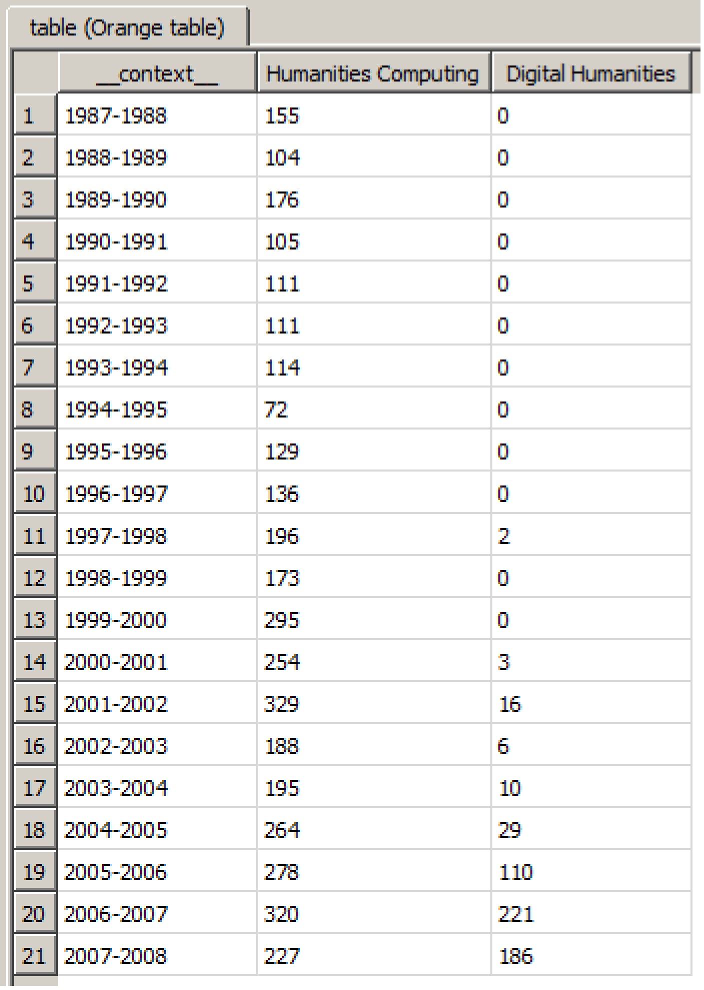


table (Orange table)

	__context__	Humanities Computing	Digital Humanities
1	1987-1988	155	0
2	1988-1989	104	0
3	1989-1990	176	0
4	1990-1991	105	0
5	1991-1992	111	0
6	1992-1993	111	0
7	1993-1994	114	0
8	1994-1995	72	0
9	1995-1996	129	0
10	1996-1997	136	0
11	1997-1998	196	2
12	1998-1999	173	0
13	1999-2000	295	0
14	2000-2001	254	3
15	2001-2002	329	16
16	2002-2003	188	6
17	2003-2004	195	10
18	2004-2005	264	29
19	2005-2006	278	110
20	2006-2007	320	221
21	2007-2008	227	186

Fig. 3: Figure 3: Monitoring the frequency of *Humanities Computing* vs. *Digital Humanities*.

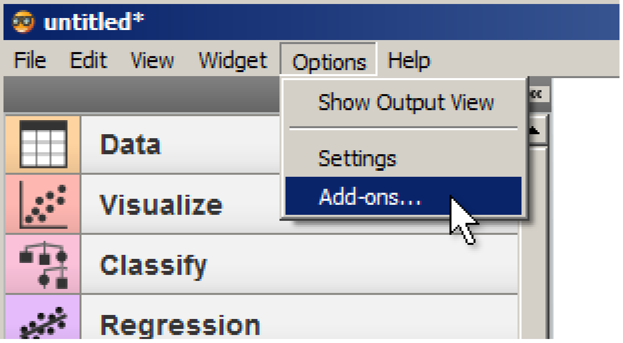
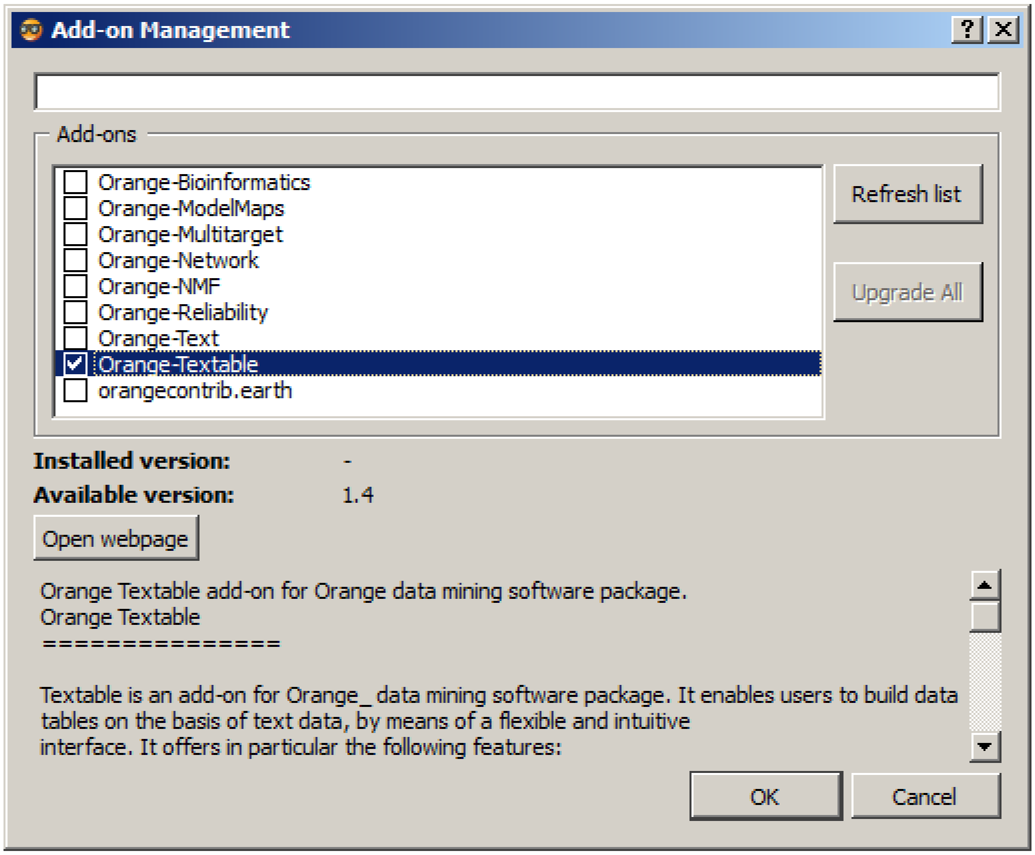


Fig. 4: Figure 1: Opening the Add-ons management dialog in Orange Canvas.



□ Orange-Bioinformatics
□ Orange-ModelMaps
□ Orange-Multitarget
□ Orange-Network
□ Orange-NMF
□ Orange-Reliability
□ Orange-Text
☑ Orange-Textable
□ orangecontrib.earth

Installed version: -
Available version: 1.4

Orange Textable add-on for Orange data mining software package.
Orange Textable
===============

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:

Fig. 5: Figure 2: Orange Textable marked for installation.

Only if step 4 was not correctly carried out:

5. Go to PyPI to download the Orange Textable Windows installer (*MS Windows installer*, *.exe* file).
6. Execute the Orange Textable installer and click on **Ok** for each stage.

If install was completed without issues but nothing happens when trying to launch the application:

Try to follow the steps described here.

MacOS X installation

1. On the Orange 2.7 download page, download the software installer by following the *Orange 2.7 bundle for OSX* link.
2. In the window that opens at the end of the download, drag the Orange Canvas icon and drop it over the *Applications* folder icon.
3. Start Orange Canvas then select menu **Options > Add-ons…** (see *figure 1*).

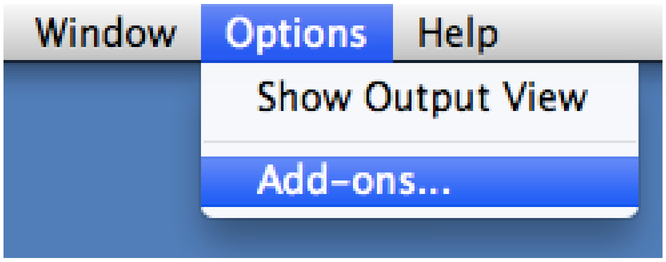


Fig. 6: Figure 1: Opening the Add-ons management dialog in Orange Canvas.

4. In the window which has opened (see *figure 2*), click on **Refresh list**, check the *Orange-Textable* box then the **Ok** button (twice).

If step 4 was carried out correctly, the Orange Textable tab appears in the list on the left of the window of Orange Canvas after having exited and restarted the program.

Only if step 4 was not correctly carried out:

5. Go to PyPI to download the source distribution of Orange Textable (*.tar.gz* file).
6. Decompress the archive then open a Terminal and navigate to the decompressed archive (see below for more details on this step). Then enter the following instruction:

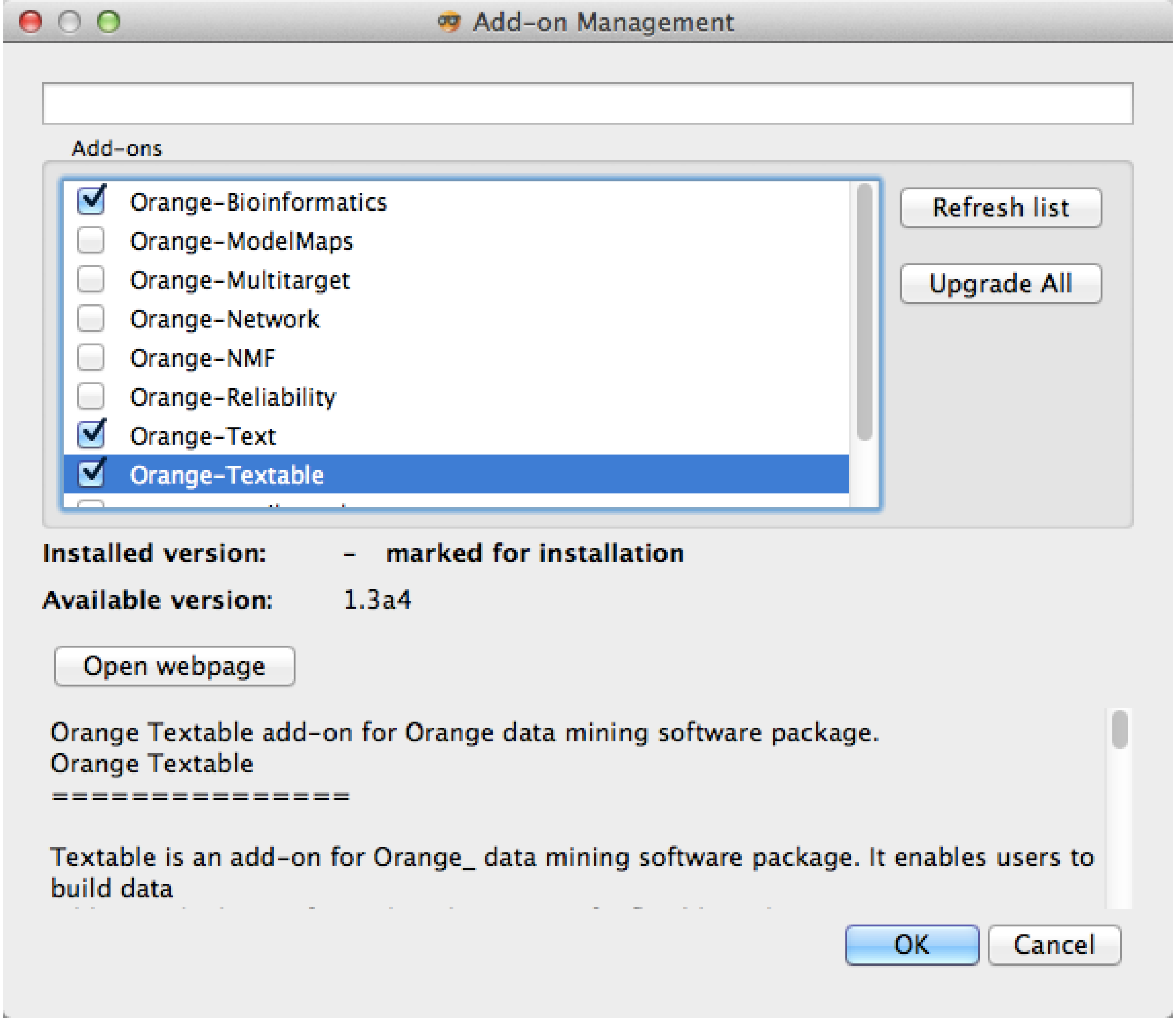
```
python setup.py install
```

NB: if this process fails, it is sometimes possible to resolve the problem by replacing the instruction with this one:

```
/Applications/Orange.app/Contents/MacOS/python setup.py install
```

In case of difficulty in "opening a Terminal and navigating to the decompressed archive…":

a. Drag and drop on the desktop the *Orange-Textable-X* file (where *X* is the version number, e.g. "1.5") which can be found in the downloaded archive.
b. In **Finder > Applications > Utilities**, double-click on *Terminal*.
c. In the Terminal, correctly enter the instruction:



Add-ons

- ☑ Orange-Bioinformatics
- □ Orange-ModelMaps
- □ Orange-Multitarget
- □ Orange-Network
- □ Orange-NMF
- □ Orange-Reliability
- ☑ Orange-Text
- ☑ Orange-Textable

Installed version: – **marked for installation**

Available version: 1.3a4

Orange Textable add-on for Orange data mining software package.
Orange Textable
===============

Textable is an add-on for Orange_ data mining software package. It enables users to build data

Fig. 7: Figure 2: Orange Textable marked for installation.

```
cd Desktop/Orange-Textable-X
```

(where *X* still is the version number).

d. Then enter the instruction:

```
python setup.py install
```

(or if necessary, the alternative instruction shown here above).

1.1.4 Configuration

Although this is by no means required for using Orange Textable, schemas created with Orange Canvas tend to be easier to read after deactivating the display of channel names on widget connections. This can be done using the **Settings** dialog of Orange Canvas, accessible on Windows via the menu entry **Options > Settings** (see *figure 1*), and on Mac OSX via the menu entry **Orange > Preferences** (see *figure 2*).

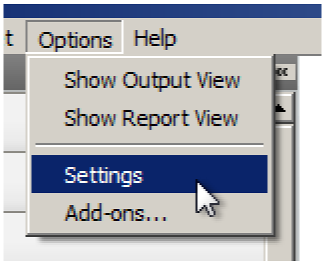


Fig. 8: Figure 1: Opening the Settings dialog on Windows.

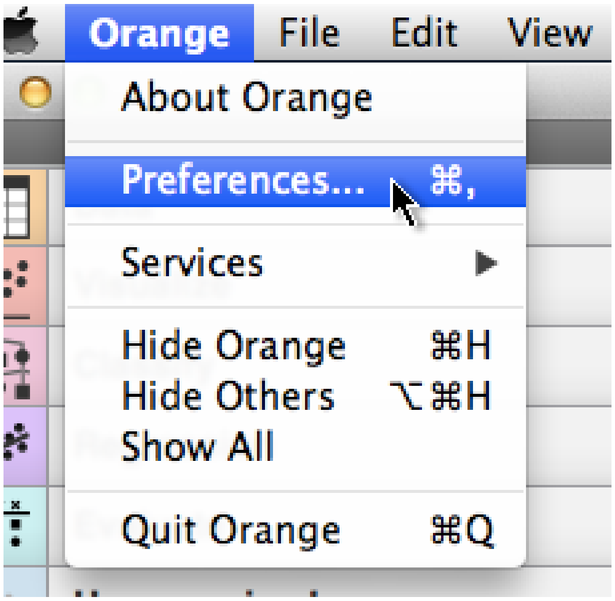
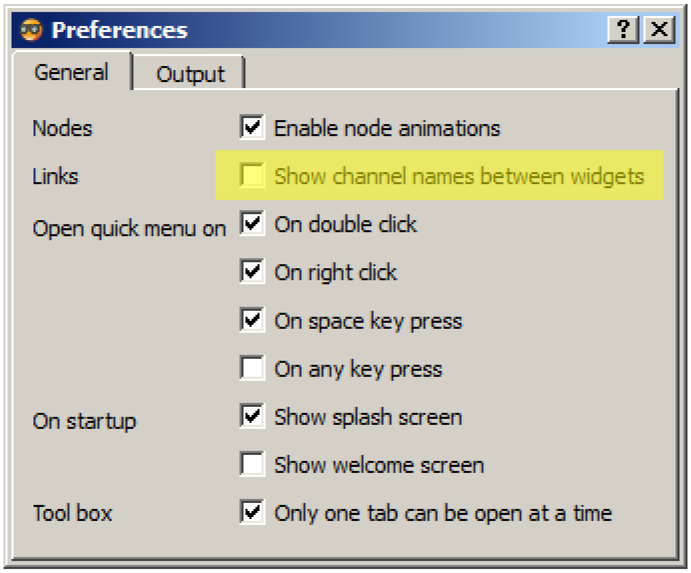


Fig. 9: Figure 2: Opening the Settings dialog on Mac OSX.

Once the dialog has been opened, the **Show channel names between widgets** checkbox should be deselected, as in *figure 3*.



Preferences

General | Output

Nodes ☑ Enable node animations

Links □ Show channel names between widgets

Open quick menu on ☑ On double click

☑ On right click

☑ On space key press

□ On any key press

On startup ☑ Show splash screen

□ Show welcome screen

Tool box ☑ Only one tab can be open at a time

Fig. 10: Figure 3: Deactivating the display of channel names on widget connections.

1.1.5 Credits

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

The largest part of funding was initially provided by the Unil's Teaching innovation fund (Fonds d'innovation pédagogique - FIP), and led to the release of Textable v1.0 in summer 2012.

Textable's development has continued between 2012 and 2013, still carried on by LangTech Sàrl, while the program was being gradually integrated to courses taught at Unil's department of SLI (where most of the tutorials that would later become the Getting started section of this documentation have been created).

In automn 2013, Textable became a registered Orange Canvas add-on and was renamed to Orange Textable (v1.3). This promotion has made it possible to reach a much larger pool of users, as witnessed by a steadily increasing number of downloads.

In early 2014, Unil's FIP has renewed its support to Orange Textable by granting a maintenance funding. This has made it possible for LangTech Sàrl to collaborate with the creators of Orange Canvas, University of Ljubljana's Biolab for producing version v1.4 of Orange Textable.

In the meantime, Unil's Faculty of Arts has granted additional funding for translating Orange Textable's User guide from French to English, then converting it into the electronic form you're currently reading. Unil's department of language and information sciences has provided some financial support to the project in 2015, which made it possible to handle warnings and error messages in a more user-friendly fashion in Orange Textable v1.5.2.

Besides LangTech Sàrl and Aris Xanthos who have been involved at about every step of Orange Textable's conception, implementation, documentation, and so on, a special mention should be made to Benjamin Gay (specifications, conception and implementation), people at Biolab (in particular Blaž Zupan and Aleš Erjavec for conception and implementation work), Corinne Morey (French to English translation of the user guide, preparation of the online version of the documentation, and creation of most cookbook recipes), Douglas Duhaime (case study design and write-up), and many students (and a growing number of scholars) mostly at Unil for their indispensable feedback as users of Orange Textable.

1.1.6 Citing

If Orange Textable has been useful in preparing a scientific publication of yours, a citation would be a great way to say so. Here is the relevant bibliographic reference:

Xanthos, Aris (2014). Textable: programmation visuelle pour l'analyse de données textuelles. In *Actes des 12èmes Journées internationales d'analyse statistique des données textuelles (JADT 2014)*, pp. 691-703. [read online]

1.2 Textable's basics

This part of the documentation is a tutorial that introduces the basic usage patterns of Orange Textable. It is meant to be read in the indicated order. Note that a basic familiarity with the interface of Orange Canvas is assumed; if needed,'this short tutorial <http://orange.biolab.si/getting-started/>'_ should provide you with the necessary backround.

Orange Textable is mostly about taking text in input and producing tables in output. What makes the transition from text to tables possible and hopefully easy is the concept of "segmentation", which is at the heart of Orange Textable.

In this section, you'll learn about segmentations and closely related topics such as strings, segments, widget labels and annotations. First, you'll learn how to import texts, second to segment it, third to annotate those segmentations in order to transform it into tables. Tables enable users to analyze text data (context, count, length, variety, cooccurrences, etc).

1.2.1 Strings, segments, and segmentations

The main purpose of Orange Textable is to turn text strings into data tables. 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, widget labelling and segmentation display* or *Cookbook: Import text from keyboard*. 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).

Figure 1 : A simple string.

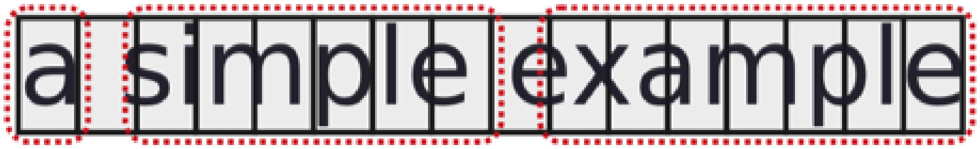
What makes the transition from text strings to data tables possible is the concept of a segmentation. What is a segmentation ? A segmentation is a string analysis based on a ordered list of segments. For instance, a string like "a simple example" above can be analyzed in many different ways: it consists of 3 words but also 16 characters, 14 letters, 6 vowels, 3 e's, 2 mple's, etc.

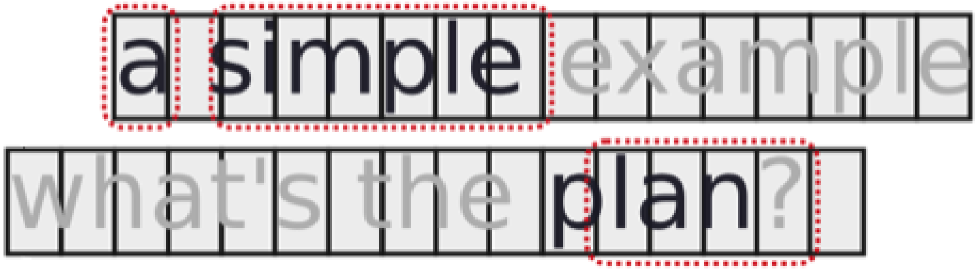
In the previous example, all the segments of a given segmentation refer to the same string. However, a segmentation can span several strings. Thus, the segments of a segmentation can cover different strings, as in the example below, where the segmentation "a", "simple", "plan" spans two strings ("a simple example" and "what's the plan"). All segments referring to a given string must be grouped together, in the order in which they appear in the string.

Figure 2 : A segmentation can span several strings.

See also

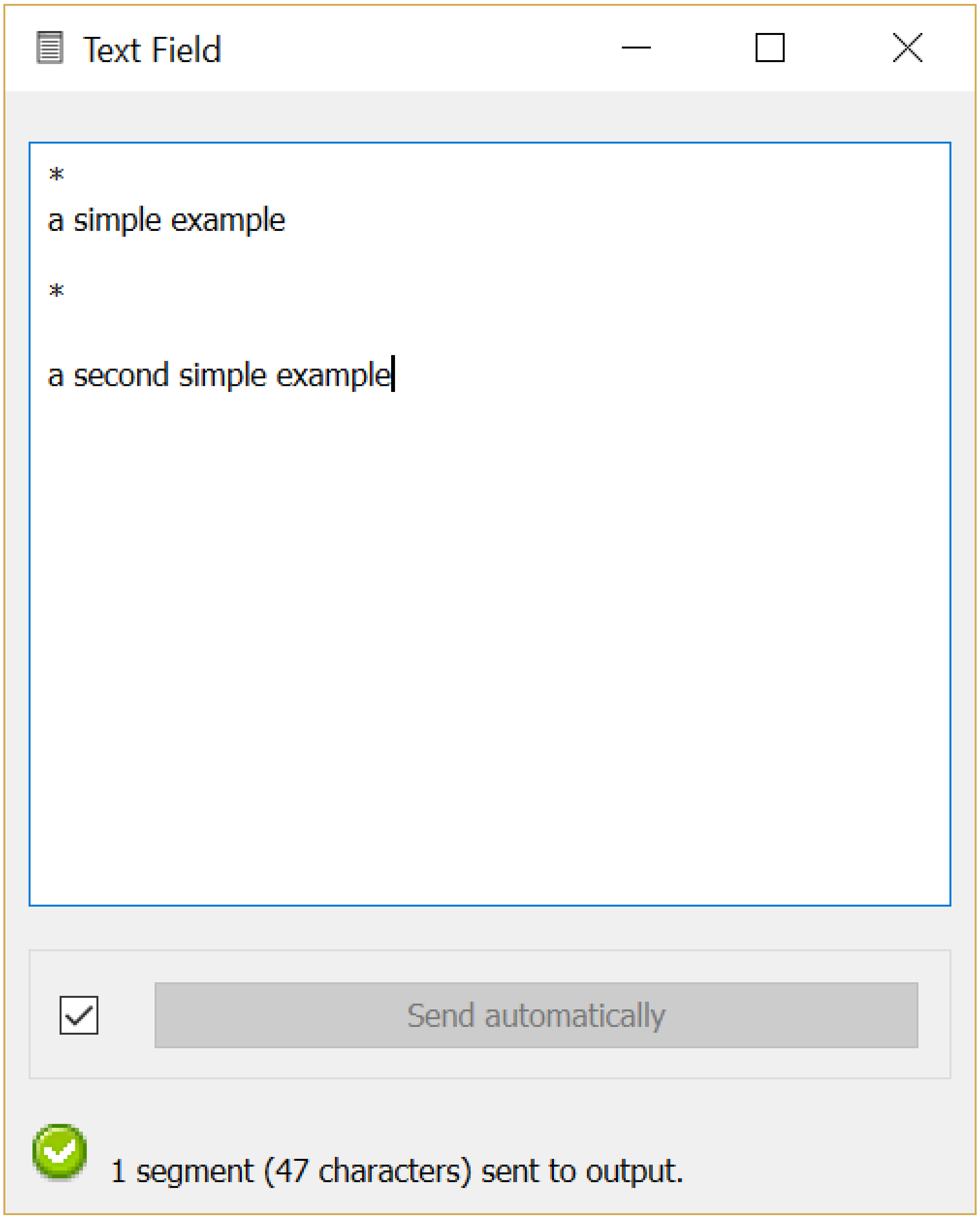
- *Getting started: Keyboard input and segmentation display*
- *Cookbook: Import text from keyboard*


a simple example


a simple example
what's the plan?

1.2.2 Keyboard input, widget labelling and segmentation display

Typing text in a *Text Field* widget is the simplest way to import a string in Orange Textable. As a result, the widget creates a segmentation with a single segment covering the entire string. (see *figure 1* below):



*
a simple example

*

a second simple example

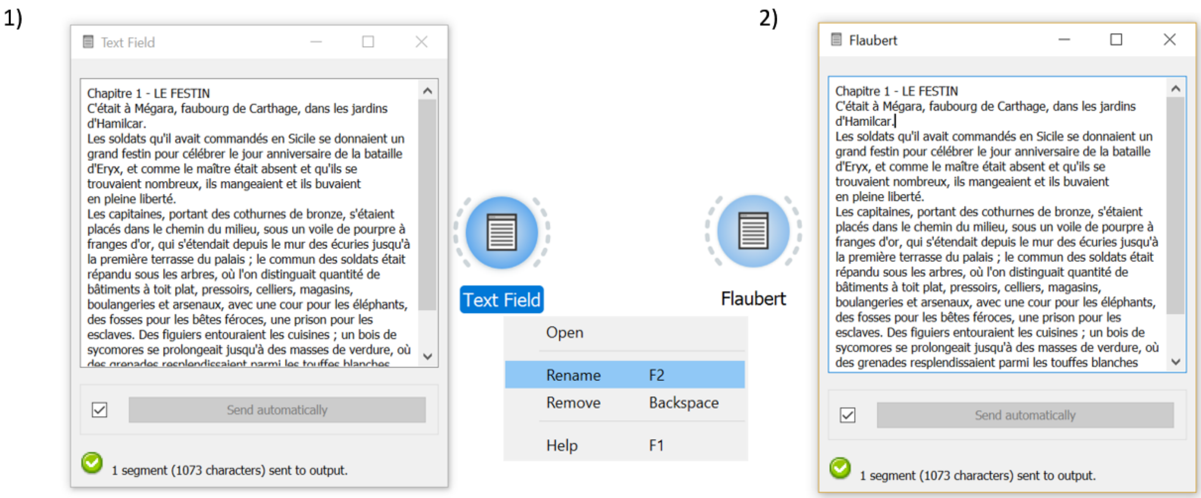
1 segment (47 characters) sent to output.

Fig. 11: Figure 1: Typing some simple examples in widget *Text Field*.

Each segmentation is identified by a label which is the name of the widget that creates the segmentation. You can rename this widget to make the label more meaningful (see :ref: *figure 2 <keyboard_input_segmentation_fig2>* below):

As we will see later, a segmentation can also store annotations associated with segments.

This widget's simplicity makes it most adequate for pedagogic purposes. Later, we will discover other, more powerful ways of importing strings such as Text Files and URLs. Those importation widgets create a segmentation with one segment for each imported file or URL.



Text Field

Chapitre 1 - LE FESTIN
C'était à Mégara, faubourg de Carthage, dans les jardins d'Hamilcar.
Les soldats qu'il avait commandés en Sicile se donnaient un grand festin pour célébrer le jour anniversaire de la bataille d'Eryx, et comme le maître était absent et qu'ils se trouvaient nombreux, ils mangeaient et ils buvaient en pleine liberté.
Les capitaines, portant des cothurnes de bronze, s'étaient placés dans le chemin du milieu, sous un voile de pourpre à franges d'or, qui s'étendait depuis le mur des écuries jusqu'à la première terrasse du palais ; le commun des soldats était répandu sous les arbres, où l'on distinguait quantité de bâtiments à toit plat, pressoirs, celliers, magasins, boulangeries et arsenaux, avec une cour pour les éléphants, des fosses pour les bêtes féroces, une prison pour les esclaves. Des figuiers entouraient les cuisines ; un bois de sycomores se prolongeait jusqu'à des masses de verdure, où des grenades resplendissaient parmi les touffes blanches

☑ Send automatically

1 segment (1073 characters) sent to output.

Flaubert

Chapitre 1 - LE FESTIN
C'était à Mégara, faubourg de Carthage, dans les jardins d'Hamilcar.
Les soldats qu'il avait commandés en Sicile se donnaient un grand festin pour célébrer le jour anniversaire de la bataille d'Eryx, et comme le maître était absent et qu'ils se trouvaient nombreux, ils mangeaient et ils buvaient en pleine liberté.
Les capitaines, portant des cothurnes de bronze, s'étaient placés dans le chemin du milieu, sous un voile de pourpre à franges d'or, qui s'étendait depuis le mur des écuries jusqu'à la première terrasse du palais ; le commun des soldats était répandu sous les arbres, où l'on distinguait quantité de bâtiments à toit plat, pressoirs, celliers, magasins, boulangeries et arsenaux, avec une cour pour les éléphants, des fosses pour les bêtes féroces, une prison pour les esclaves. Des figuiers entouraient les cuisines ; un bois de sycomores se prolongeait jusqu'à des masses de verdure, où des grenades resplendissaient parmi les touffes blanches

☑ Send automatically

1 segment (1073 characters) sent to output.

Fig. 12: Figure 2: Typing an extract of *Salammbô* in widget :ref: *Text Field* and giving it a label (Flaubert).

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 3* below).

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

See also

- *Reference: Text Field widget*
- *Reference: Display widget*
- *Cookbook: Import text from keyboard*
- *Cookbook: Display text content*

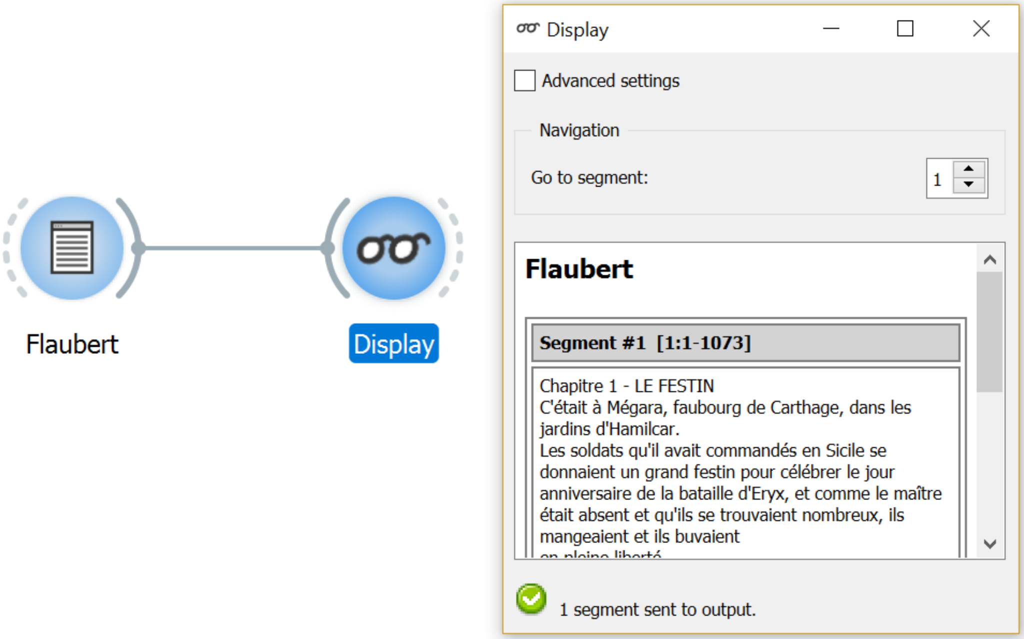
Footnotes

[#] 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.

1.2.3 Merging and segmenting

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*.



Display

□ Advanced settings

Navigation

Go to segment: 1

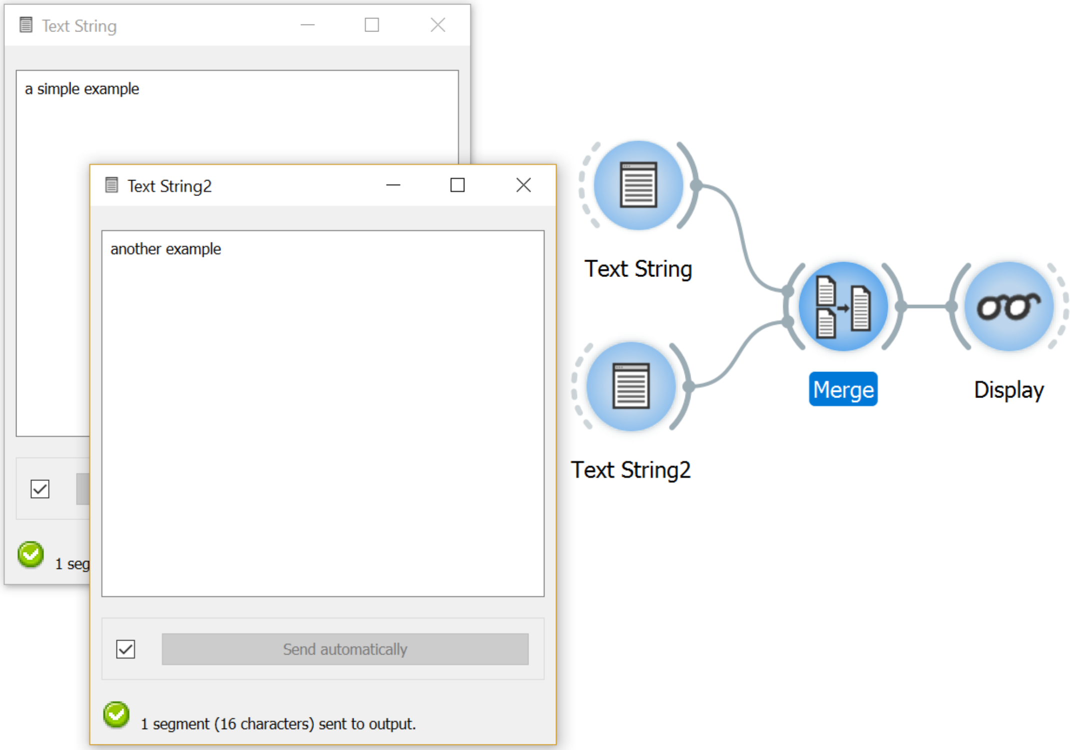
Flaubert

Segment #1 [1:1-1073]

Chapitre 1 - LE FESTIN
C'était à Mégara, faubourg de Carthage, dans les jardins d'Hamilcar.
Les soldats qu'il avait commandés en Sicile se donnaient un grand festin pour célébrer le jour anniversaire de la bataille d'Eryx, et comme le maître était absent et qu'ils se trouvaient nombreux, ils mangeaient et ils buvaient
en pleine liberté.

1 segment sent to output.

Fig. 13: Figure 3 : Viewing *Salammbô* in widget *Display*.



Text String

a simple example

☑

1 seg

Text String2

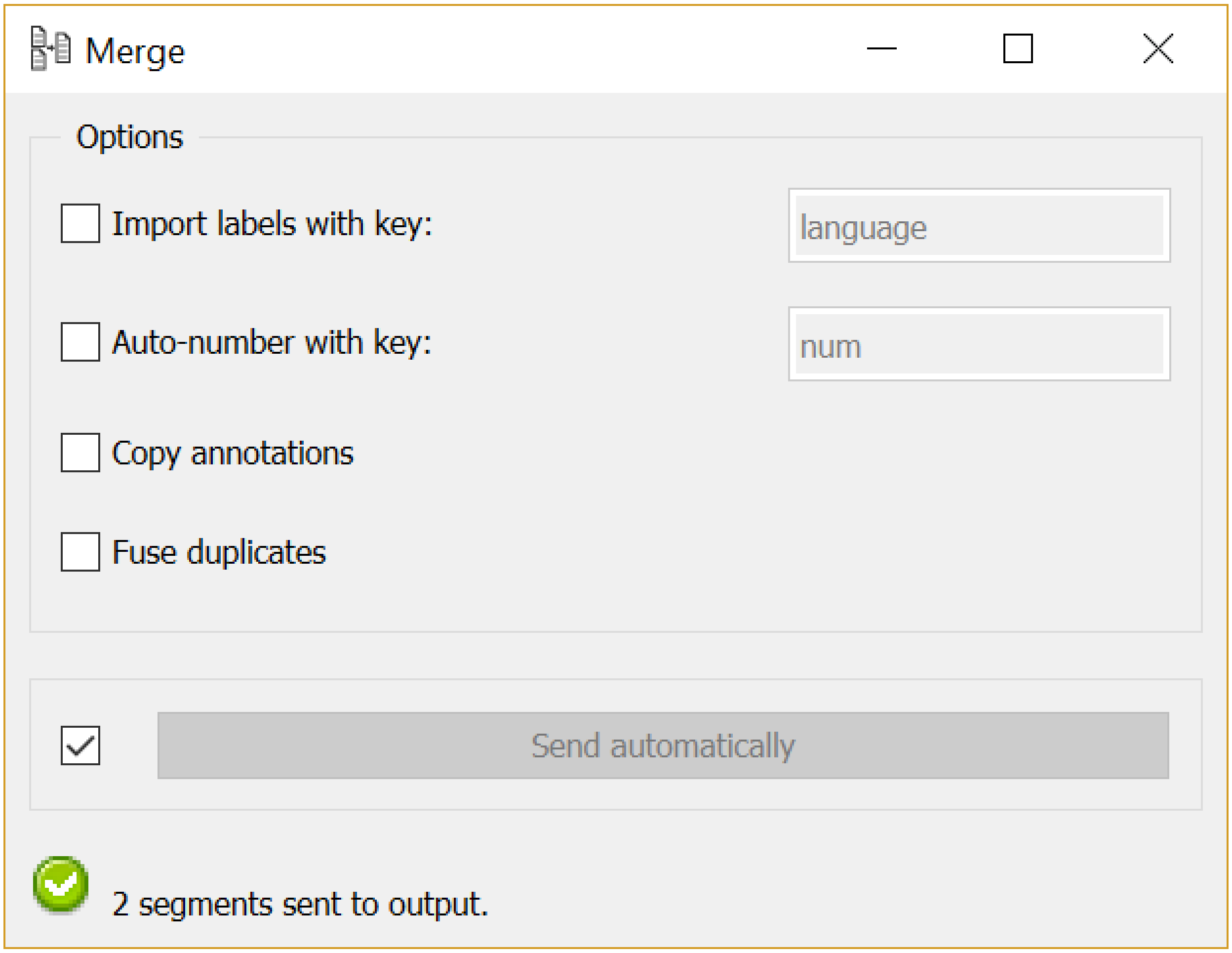
another example

☑ Send automatically

1 segment (16 characters) sent to output.

Fig. 14: Figure 1: Grouping *a simple example* with *another example* using widget *Merge*.

The interface of widget *Merge* (see *figure 2* below) features 4 options : 2 annotation keys; the possibility of copying segment inputs annotations if any and the option of fusing segments that have the same adress.



Merge

Options

☐ Import labels with key: language

☐ Auto-number with key: num

☐ Copy annotations

☐ Fuse duplicates

☑ Send automatically

2 segments sent to output.

Fig. 15: Figure 2: Interface of widget *Merge*.

We will return later to the purpose of checkbox **Import labels with key**, as well as **Auto-number with key**. Leave them unchecked for now.

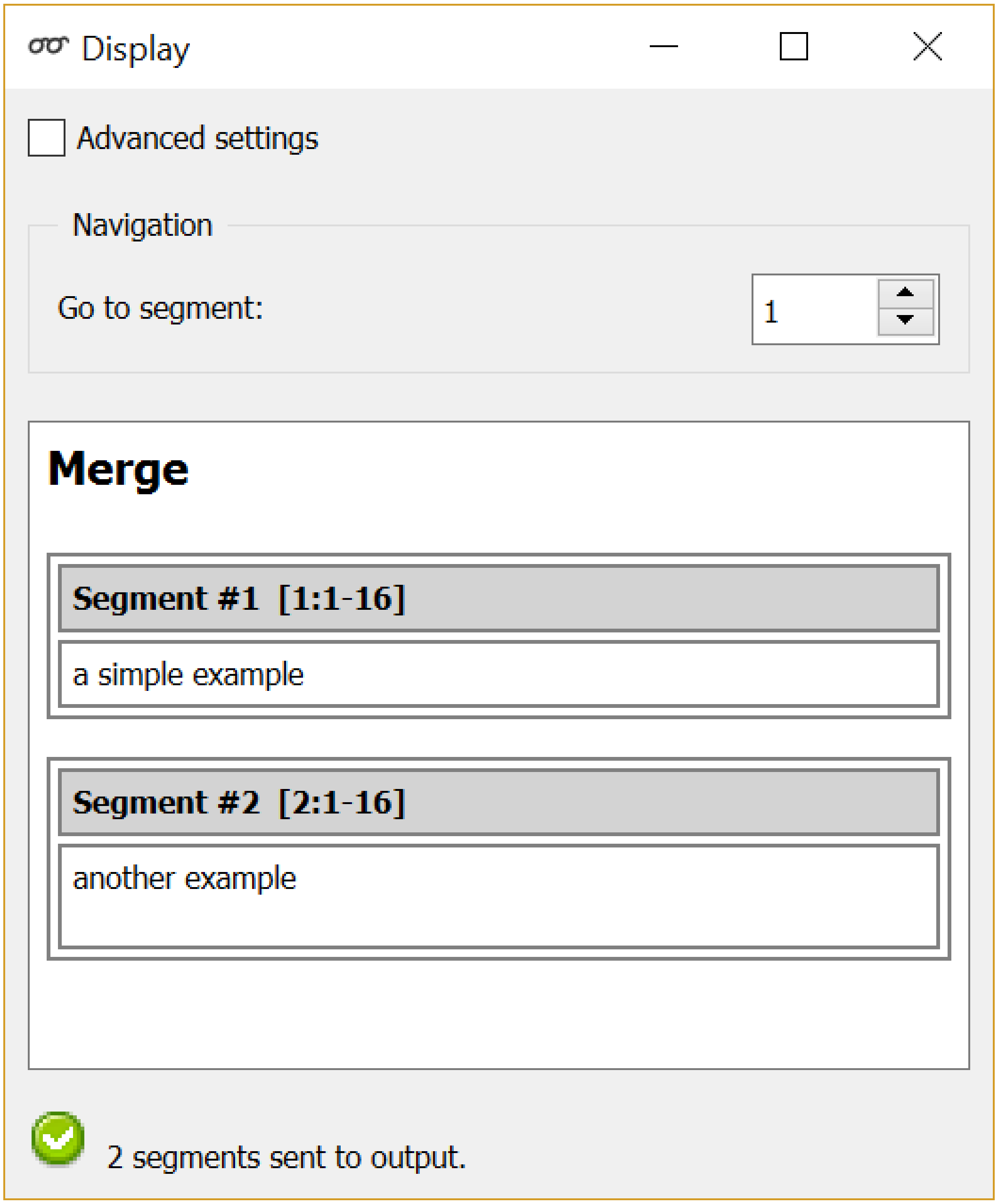
Figure 3 above shows the resulting merged segmentation, as displayed by widget *Display*. As can be seen, *Merge* makes it easy to concatenate several strings into a single segmentation. If the incoming segmentations contained several segments, each of them would appear in the output segmentation, in the order they have been linked to the Merge widget.

Exercise: Can you add a new instance of *Merge* to the schema 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*)

See also

- *Reference: Merge widget*
- *Cookbook: Merge several texts*



□ Advanced settings

Go to segment: 1

Merge

Segment #1 [1:1-16]

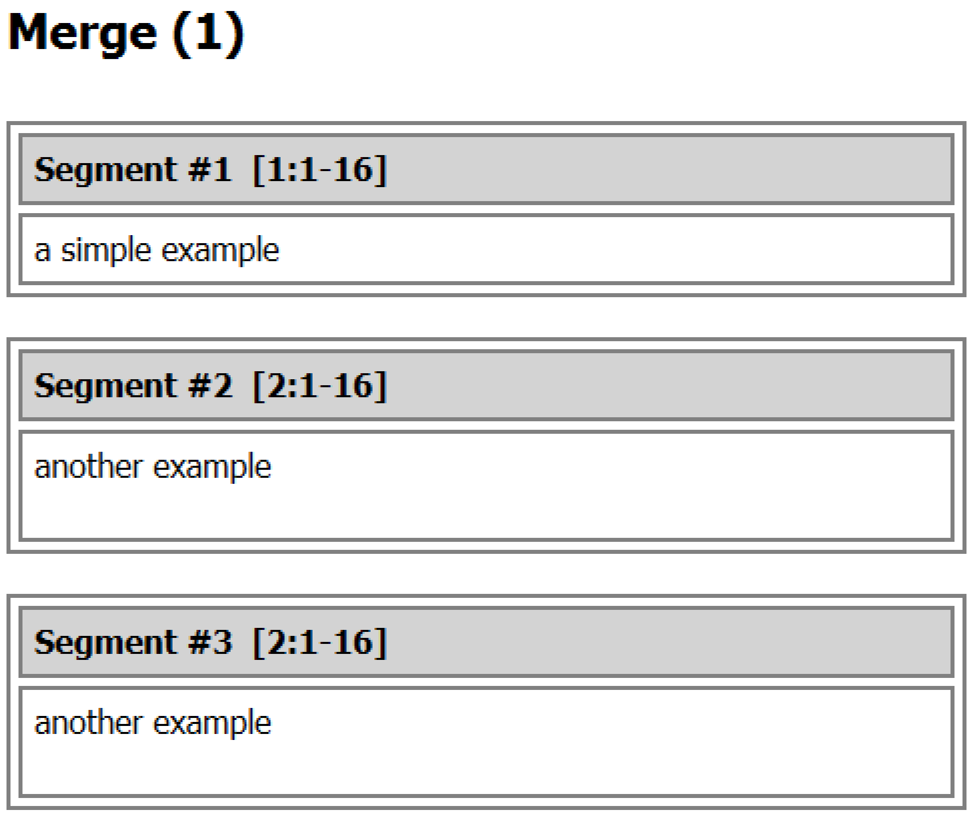
a simple example

Segment #2 [2:1-16]

another example

2 segments sent to output.

Fig. 16: Figure 3: Merged segmentation.



Merge (1)

Segment #1 [1:1-16]

a simple example

Segment #2 [2:1-16]

another example

Segment #3 [2:1-16]

another example

Fig. 17: Figure 4: The segmentation requested in the *exercise*.

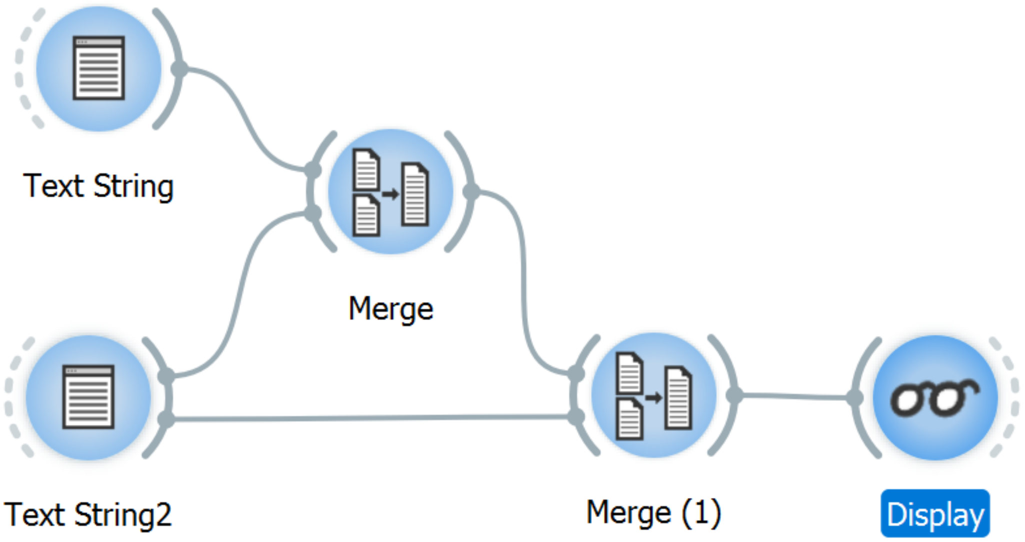


Fig. 18: Figure 5: Solution to the *exercise*.

1.2.4 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 schema 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*.

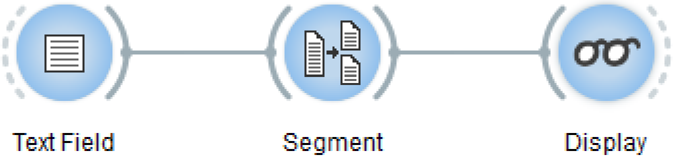
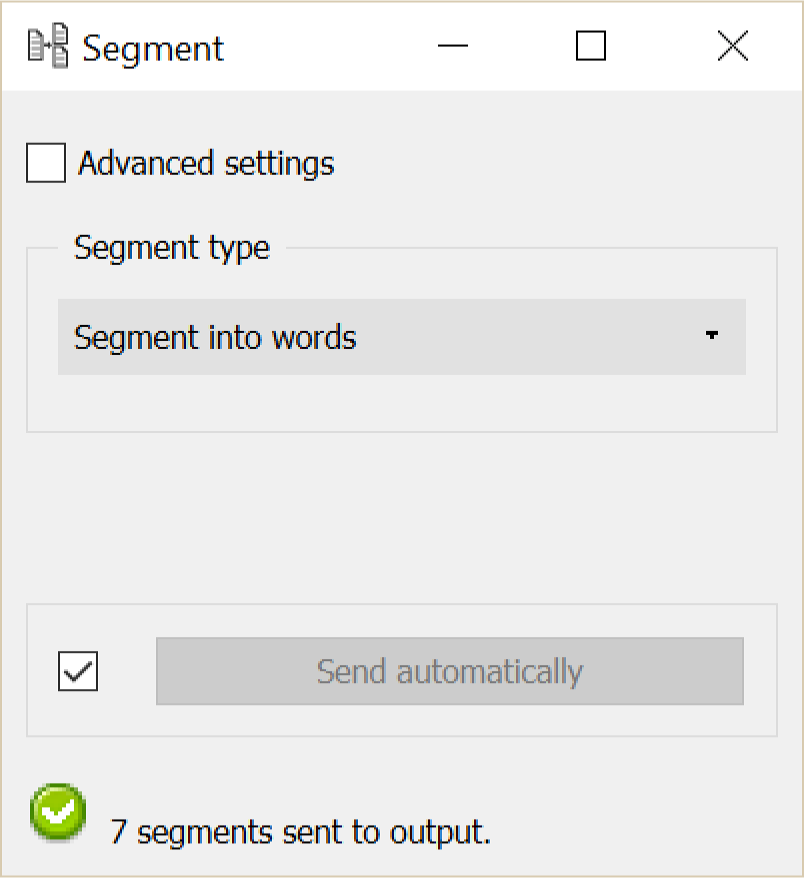


Fig. 19: Figure 1: A schema for testing the *Segment* widget

In its basic form (i.e. with **Advanced settings** unchecked, see *figure 2* below), *Segment* offers four parameters in the drop-down menu named segment type. The string can be segmented into lines, letters, words or using a regex. If chose, 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.


Segment

□ Advanced settings

Segment type

Segment into words

☑ Send automatically

7 segments sent to output.

Fig. 20: Figure 2: Interface of the *Segment* widget, configured for word 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, 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 inflected forms of *retrieve* with `\bretriev(e|es|ed|ing)\b`.

With the Advanced settings checked (see figure 3 below), several regexes can be added to the list. Regexes can be tokenized or splited, depending on your research goal. For more information, see *Segment widget*

See also

- *Reference: Segment widget*
- *Cookbook: Segment text in smaller units*

1.2.5 The uses of annotating segmentations

Annotations are bits of information attached to text segments. They let you go beyond what's in the text, and extend Orange Textable's analytic capacities from textual content to user-provided interpretative information and metadata.

In Orange Textable, an *annotation* is a piece of information attached to a segment. Annotations consist of two parts : *key* and *value* . For instance, in the now classical case of the word segmentation of *a simple example* (see :ref: *figure 1<uses_annotating_segmentations_fig1>* below), segment *simple* could be associated with the annotation *{part of speech : adjective}*; this annotation's key is *part of speech* and its value is *adjective* .

Figure 1 : Annotating *simple* as an adjective.

A segment can have zero, one, or several annotations attached to it. The same segment could be simultaneously associated with another annotation such as *{word category : lexical}* , or any *{key : value}* pair deemed relevant.

Figure 2 : Segments with various annotations

Note that annotations keys are unique : Since they serve to recognize various annotation values attached to a single segment, annotation keys cannot be duplicated within the segment. On :ref: *figure 2 <uses_annotating_segmentations_fig2>* above, "simple" can only have one value at a time for key "category" .

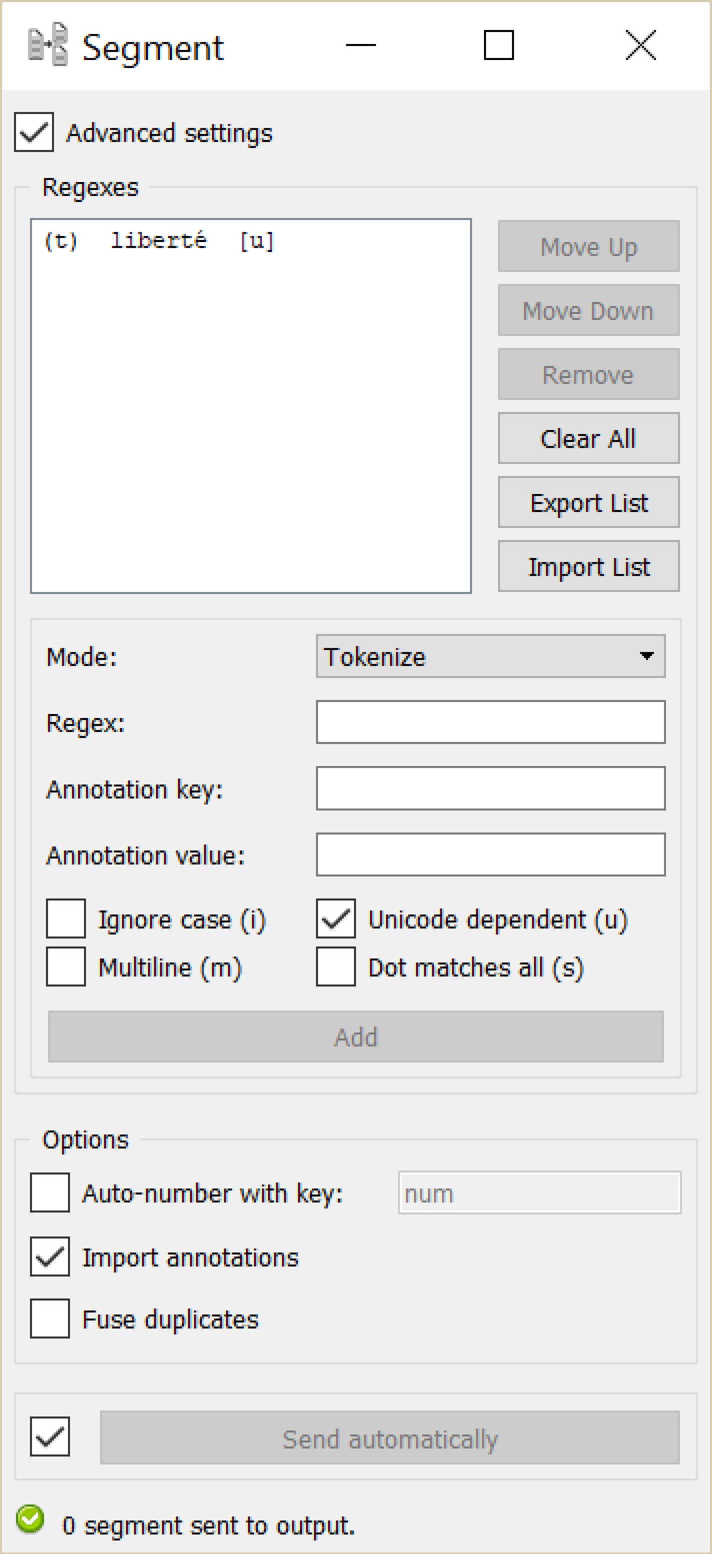
Even though we have carefully ignored them so far, annotations play a fundamental role in text data processing and analysis. They make it possible to go beyond the basic level of forms that are "physically" present in a text and tap into the more abstract–and often more interesting–level of the *interpretation* of these forms.

For instance, the texts composing a given corpus could be annotated with respect to their genre (*novel* , *short story* , and so on), and the parts of these texts might be annotated with regard to their discourse type (*narrative* , *description* , *dialogue* , and so on). Such data could be exploited to study the distribution of discourse types as a function of genre, which would be at best extremely difficult, if ever possible, without having encoded the relevant information by means of annotations.

In the following section, we will see a simple method for creating annotations in Orange Textable using the :ref: *Merge* widget, and then various ways of exploiting such annotations.

1.2.6 Merging and annotating

Whenever Textable widgets manipulate text contents, they can manipulate annotations instead: you can search for segments attached to specific annotations, count annotations, merge data based on their annotations, etc.



☑ Advanced settings

```
(t)  liberté  [u]
```

Mode: Tokenize

Regex:

Annotation key:

Annotation value:

☐ Ignore case (i) ☑ Unicode dependent (u)

☐ Multiline (m) ☐ Dot matches all (s)

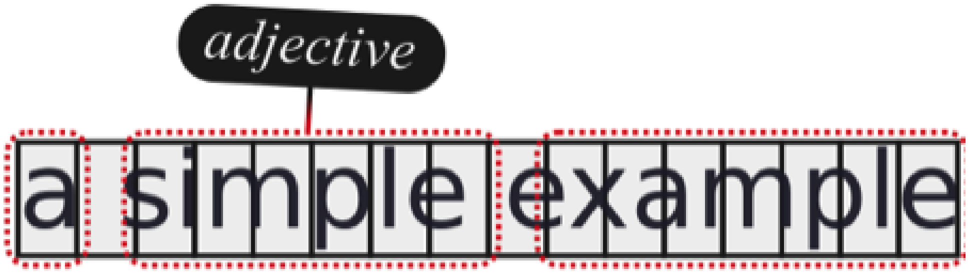
☐ Auto-number with key: num

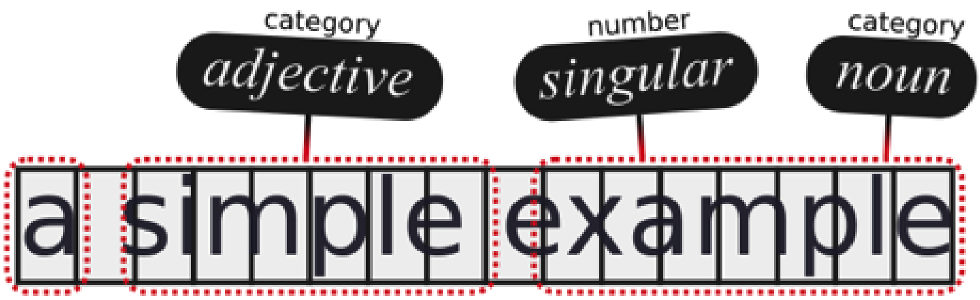
☑ Import annotations

☐ Fuse duplicates

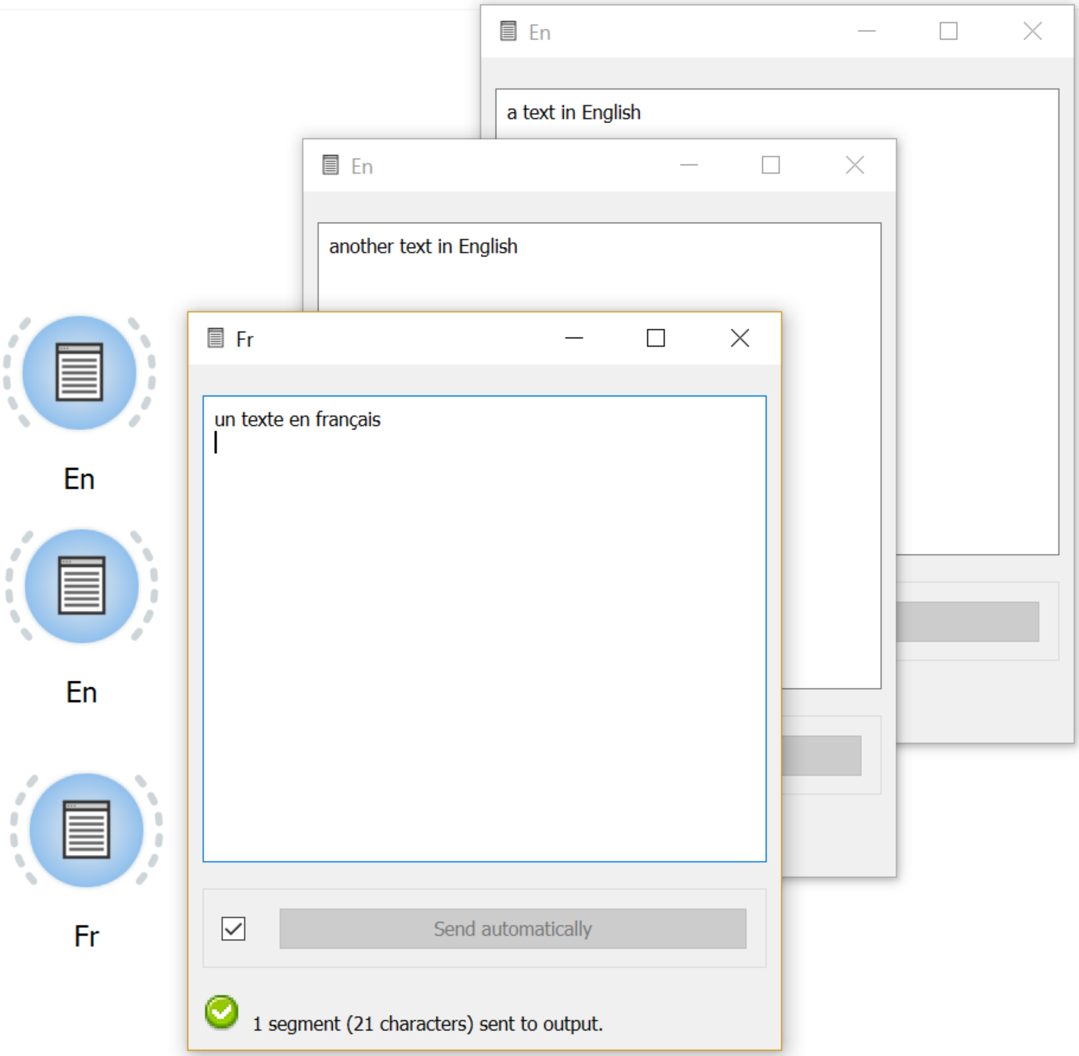
☑ Send automatically

0 segment sent to output.


adjective
a simple example


category
adjective
number
singular
category
noun
a simple example

Widget *Merge* makes it possible to convert the labels of its input segmentations into annotation values. Suppose for instance that three instances of *Text Field* have been created: two instances containing a text in English, and one containing a text in French. We might want to merge these three segmentations into a single one, where each segment would be associated with an annotation whose key is *language* and whose value is either *en* or *fr*. The first step would then be to rename each *Text Field* instance with the desired annotation value for this text, as shown on *figure 1* below.



a text in English

another text in English

un texte en français

☑ Send automatically

✓ 1 segment (21 characters) sent to output.

Fig. 21: Figure 1: Specifying annotations values using the label of *Text Field* instances.

The three instances of *Text Field* should then be connected to an instance of *Merge* as shown on *figure 2* below.

One must still specify, in the interface of *Merge*, the annotation key to which values *en* and *fr* should be associated. This can be done by entering the string *language* in field **Import labels with key**, having previously ensured that labels would actually be converted into annotation values by checking the box at the left of this line (see figure 3 below). In order to give a value to each string, check **Auto-number with key** box. As a key, you can choose text, num, author, etc. Each segment will be given a specific number. .. _annotating_merging_fig3:

The result of these operations can be viewed using an instance of *Merge*, whose output is shown on figure 4 below. For each segment in the merged segmentation, an annotation value *en* or *fr* associated with key *language* is displayed between the segment's address and its content. Note that the auto-number value offers the possibility to access each

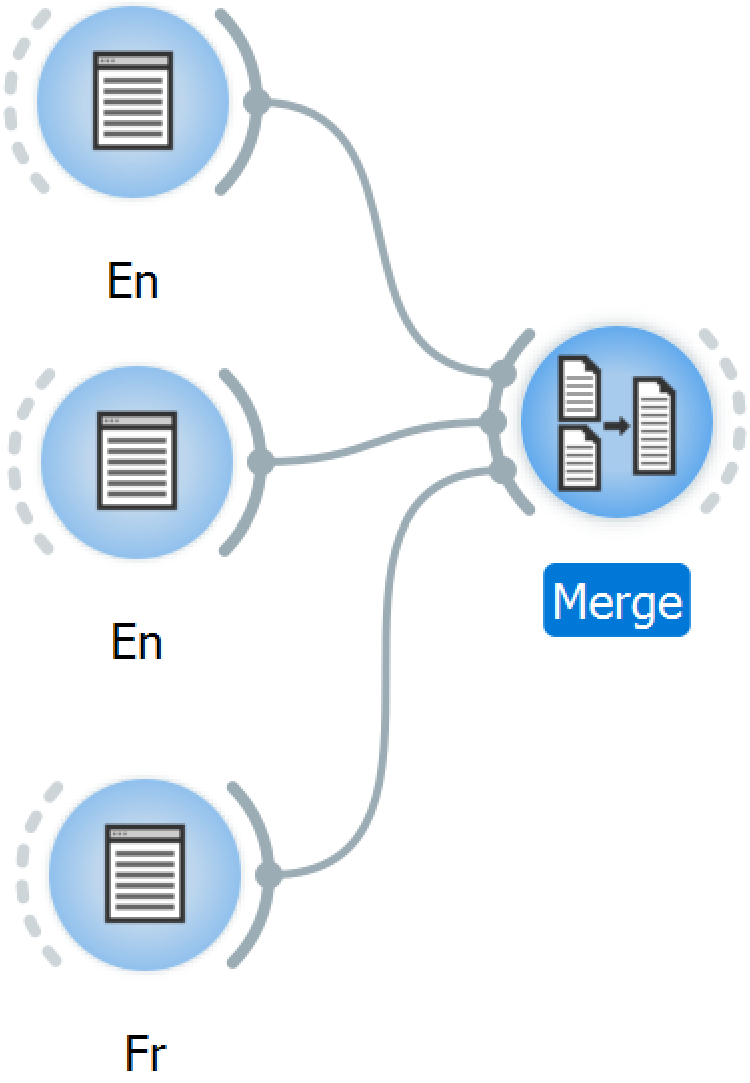
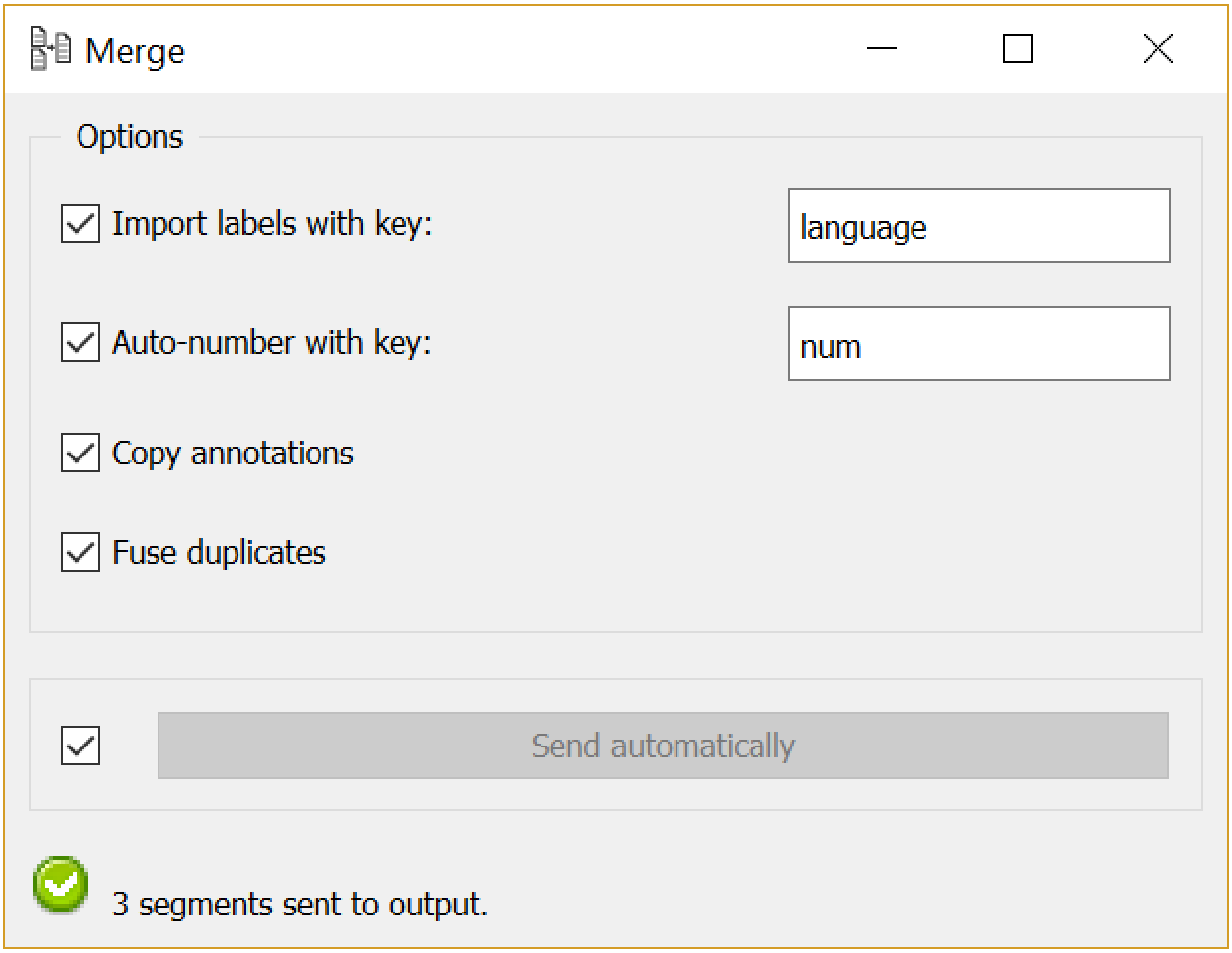


Fig. 22: Figure 2: Example schema for creating annotations with *Merge*.



Merge

Options

☑ Import labels with key: language

☑ Auto-number with key: num

☑ Copy annotations

☑ Fuse duplicates

☑ Send automatically

3 segments sent to output.

Fig. 23: Figure 3: Importing labels as annotation values with *Merge*.

segment by using the drop-down menu **Go to segment**. .. _annotating_merging_fig4:

See also

- *Reference: Text Field widget*
- *Reference: Merge widget*

1.2.7 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 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: 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

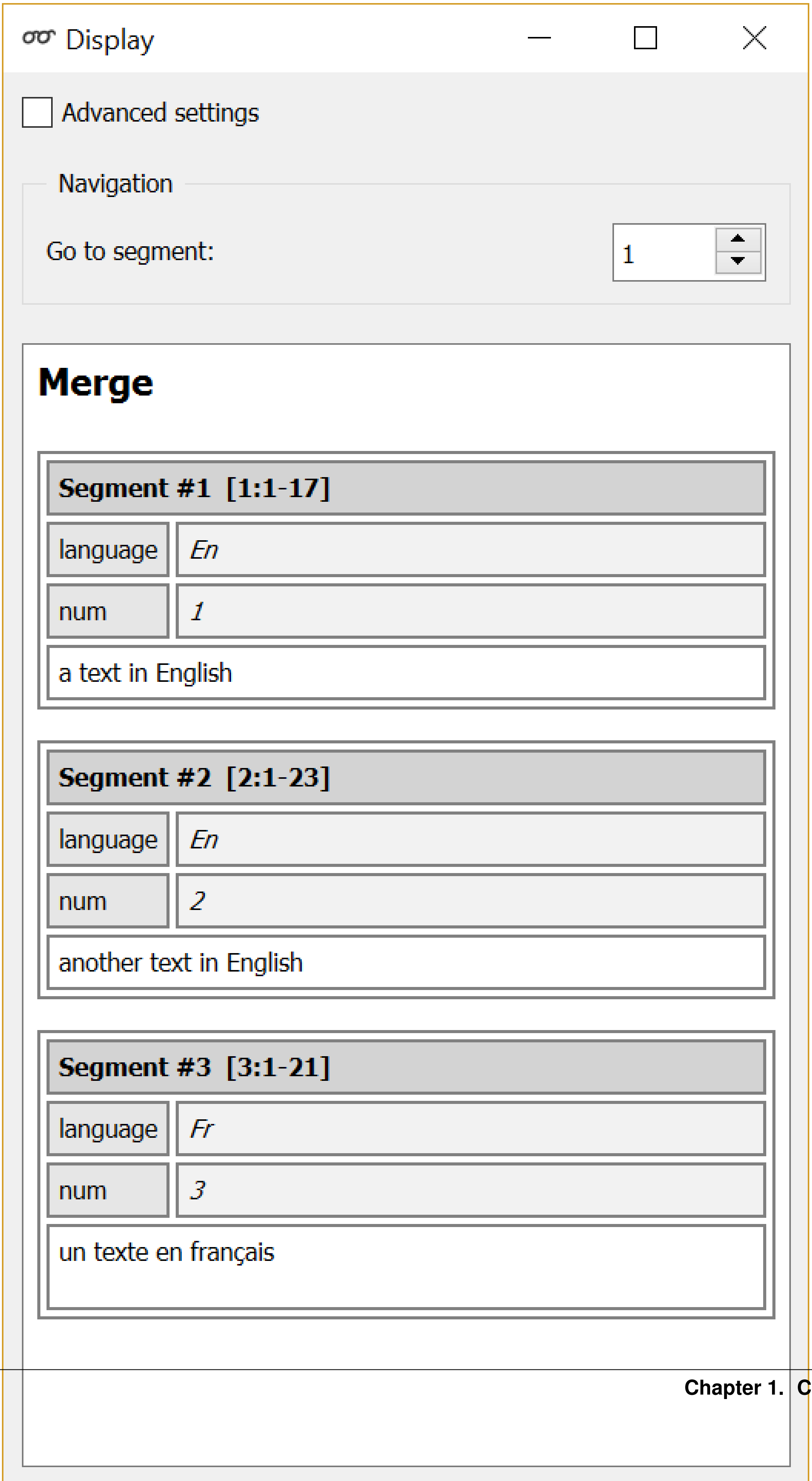
In a real application, rows could correspond to the writings of an author and columns to selected prepositions, for instance. The general idea is to determine the number of occurrences of various *units* in various *contexts*. Such data can then be further analyzed, typically by means of a statistical 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).

See also

- *Reference: Table construction widgets*

1.2.8 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 schema 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 **Widget Segment label:** *letters*). The default configuration of *Data Table* (from the **Data** tab of Orange Canvas) needs not be modified for this example.



Display

□ Advanced settings

Navigation

Go to segment: 1

Merge

Segment #1 [1:1-17]	
language	*En*
num	*1*
a text in English	

Segment #2 [2:1-23]	
language	*En*
num	*2*
another text in English	

Segment #3 [3:1-21]	
language	*Fr*
num	*3*
un texte en français	

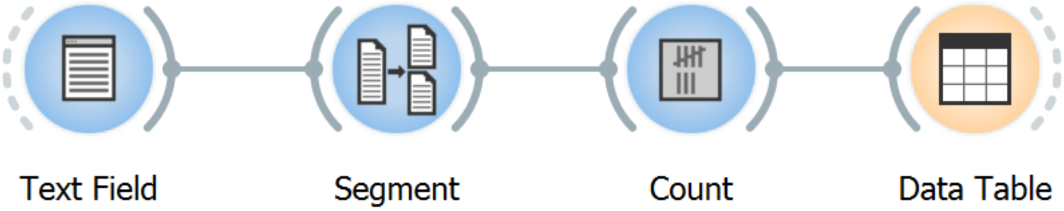


Fig. 25: Figure 1: Schema for testing the *Count* widget.

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, 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).

Note that checkbox *Send automatically* is unchecked by default so that the user must click on **Send** 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 3: Table 1: Letter bigram frequency.

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

See also

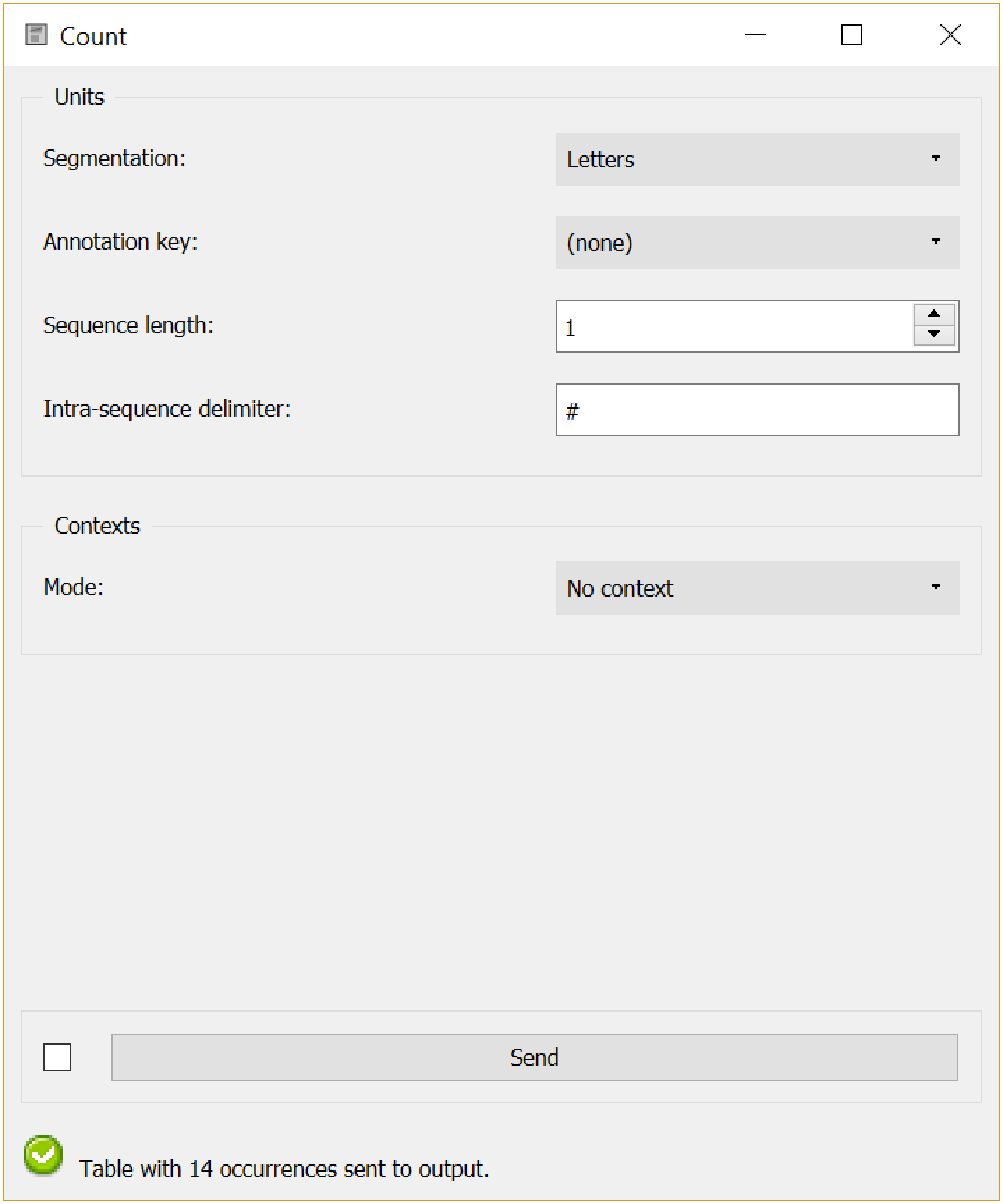
- *Getting started: From segmentations to tables*
- *Reference: Count widget*
- *Reference: Table construction widgets*
- *Cookbook: Count unit frequency*

1.2.9 Counting in specific contexts

Section **Contexts** of widget *Count*'s interface lets the user define the *contexts* in which units should be counted. Thus, while the settings of section **Units** affect the *columns* of the resulting table, those of section **Contexts** affect its *rows*.

In the example of the *previous section*, setting **Mode** to **No context** indicated that units were to be counted *globally* in the selected segmentation; as a result, the resulting table contained a single row (aside from the header row). Orange Textable offers three other modes corresponding to three different definitions of contexts.

When **Mode** is set to **Sliding window** (see *figure 1* above), context is defined as a "window" of *n* consecutive segments which "slides" from the beginning to the end of the segmentation. In the case of the letter segmentation of *a simple*



Segmentation: Letters

Annotation key: (none)

Sequence length: 1

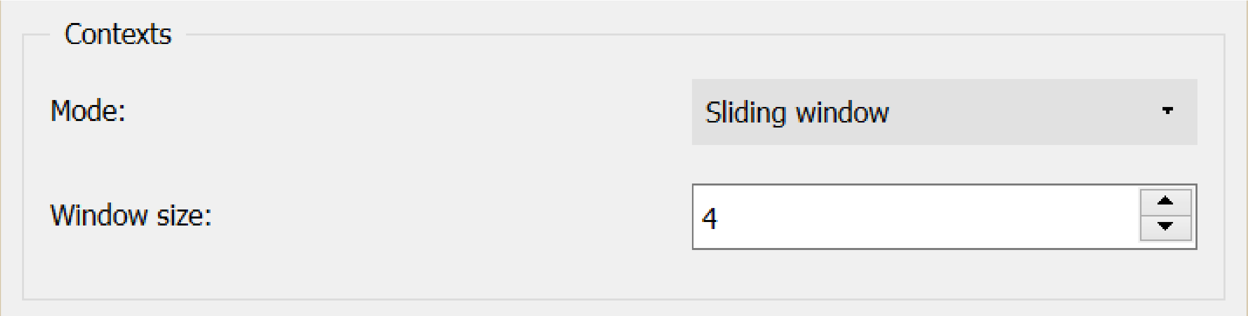
Intra-sequence delimiter: #

Mode: No context

□ Send

Table with 14 occurrences sent to output.

Fig. 26: Figure 2: Counting the frequency of letter types with widget *Count*.



Contexts

Mode: Sliding window

Window size: 4

Fig. 27: Figure 1: Interface of widget *Count*, Sliding window mode.

example (as obtained with the schema illustrated in the *previous section*), setting the number of segments in the window (**Window size**) to 5 yields the following successive contexts: *asimp*, *simpl*, *imple*, *mplee*, *pleex*, and so on (see *table 1* below). This mode is useful for studying the evolution of unit frequencies throughout a segmentation.

Table 4: Table 1: Frequency of letters in a "sliding window" of size 5.

	a	*e*	*i*	*m*	*l*	*p*	*s*	*x*
1	1	0	1	1	0	1	1	0
2	0	0	1	1	1	1	1	0
3	0	1	1	1	1	1	0	0
4	0	2	0	1	1	1	0	0
5	0	2	0	0	1	1	0	1
6	1	2	0	0	1	0	0	1
7	1	2	0	1	0	0	0	1
8	1	1	0	1	0	1	0	1
9	1	0	0	1	1	1	0	1
10	1	1	0	1	1	1	0	0

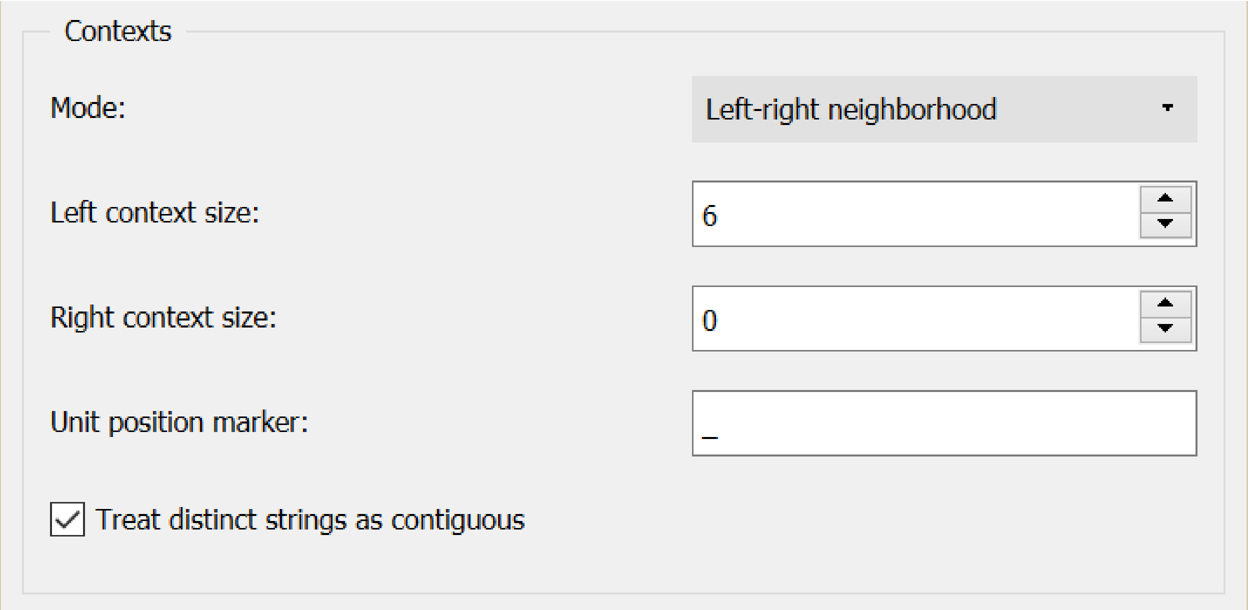
When **Mode** is set to **Left-right neighborhood** (see *figure 2*), context is defined on the basis of adjacent segment types occurring to the left and/or right of each position.

For instance, setting **Left context size** to 1 and **Right context size** to 0 amounts to counting the frequency of each segment type given the type that occurs immediately to its left. This particular table is often called "transition matrix" (see *table 2* below). The string selected in the **Unit position marker** string is used to indicate the position where units appear in the context. Thus, *table 2* shows that both *m* and *s* appear once immediately to the right of an *a* (i.e. in context *a_*). To take another example, setting **Right context size** to 2, we would find that *e* occurs once both in context *l_ex* and *e_xa*.

Table 5: Table 2: Frequency of letter (row) to letter (column) transitions.

	a	*e*	*i*	*m*	*l*	*p*	*s*	*x*
*a_*	0	0	0	1	0	0	1	0
*s_*	0	0	1	0	0	0	0	0
*i_*	0	0	0	1	0	0	0	0
*m_*	0	0	0	0	0	2	0	0
*p_*	0	0	0	0	2	0	0	0
*l_*	0	2	0	0	0	0	0	0
*e_*	0	1	0	0	0	0	0	1
*x_*	1	0	0	0	0	0	0	0

Finally, when **Mode** is set to **Containing segmentation**, unit types are counted whithin the segment types of a second



Contexts

Mode: Left-right neighborhood

Left context size: 6

Right context size: 0

Unit position marker: _

☑ Treat distinct strings as contiguous

Fig. 28: Figure 2: Interface of widget *Count*, Left-right neighborhood mode.

segmentation, as illustrated in table 2 *here* (frequency of letters whithin words). Segment *A* is considered to be contained within segment *B* if the following three conditions are met:

- A and B refer to the same string (their addresses have the same string index)
- A's initial position is greater than or equal to B's initial position
- A's final position is lesser than or equal to B's initial position

To try this mode out, modify the schema used in the *previous section* as illustrated on *figure 3* below.

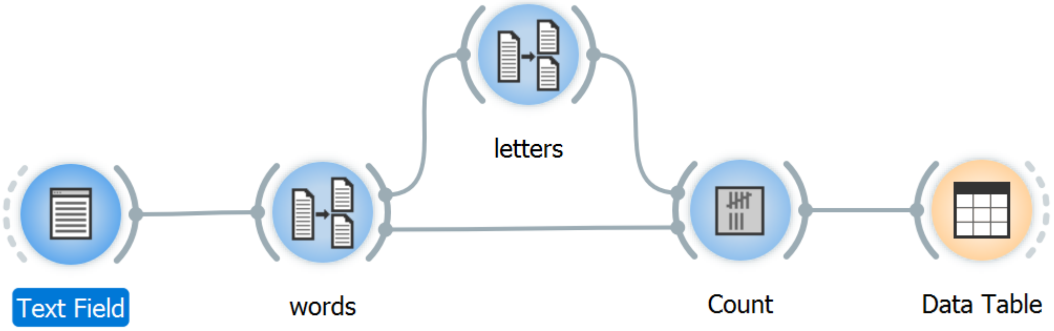
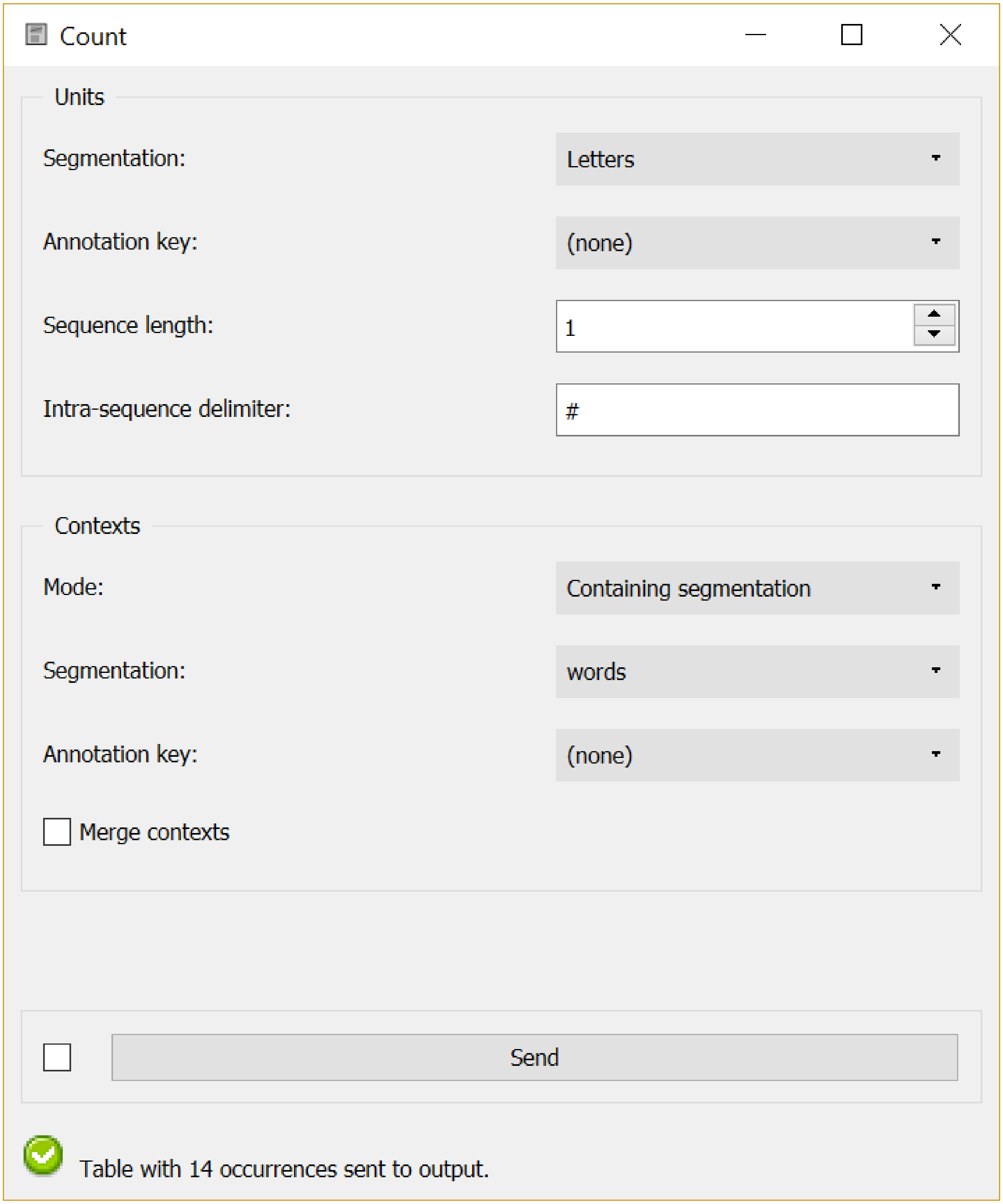


Fig. 29: Figure 3: Schema for testing the *Count* widget (Containing segmentation mode).

The first instance of *Segment* produces a word segmentation (**Regex:** `\w+` and **Widget label:** *Words*) which the second instance (the upper one) further decomposes into letters (**Regex:** `\w` and **Widget label:** *Letters*). The instance of *Count* is configured as shown on *figure 4* below. The resulting table is the same as table 2 *here* (possibly with a different ordering of columns).

Note that in this mode, checking the **Merge contexts** box still restricts counting to those units that are contained



Count

Units

Segmentation: Letters

Annotation key: (none)

Sequence length: 1

Intra-sequence delimiter: #

Contexts

Mode: Containing segmentation

Segmentation: words

Annotation key: (none)

□ Merge contexts

□ Send

Table with 14 occurrences sent to output.

Fig. 30: Figure 4: Configuration of widget *Count* for counting letters in words.

whithin the segments of another segmentation, but without treating each context type separately. In the case of letters whithin words, there is no difference between this mode and mode **No context** (see *previous section*). It does however make a difference in the case of letter bigram counting, because those bigrams that straddle a word boundary will be excluded in this case (contrary to what can be seen in table 1 *here*).

See also

- *Getting started: Counting segment types*
- *Getting started: From segmentations to tables*
- *Reference: Count widget*
- *Cookbook: Count unit frequency*
- *Cookbook: Count occurrences of smaller units in larger segments*
- *Cookbook: Count transition frequency between adjacent units*
- *Cookbook: Examine the evolution of unit frequency along the text*

1.2.10 Tagging table rows with segments and labels

There are several situations in which annotations attached to a segment can be used in place of this segment's content. A particularly common case consists in using annotations for tagging the rows of a table built with an instance of *Count*, *Length*, *Variety*, or *Category*.

Consider the example of the texts in English and French introduced here. Suppose that after having merged them into a single segmentation with an instance of *Merge* (**Widget Merge label:** *Texts* ; **Import labels with key:** *language*), we segment these three texts into letters with an instance of *Segment* (**Regex** `\w`, **Widget Segment label:** *letters*), as in the schema shown on *figure 1* below; both segmentations (*texts* and *letters*) can then be sent to an instance of *Count* for building a table with the frequency of each letter in each text.

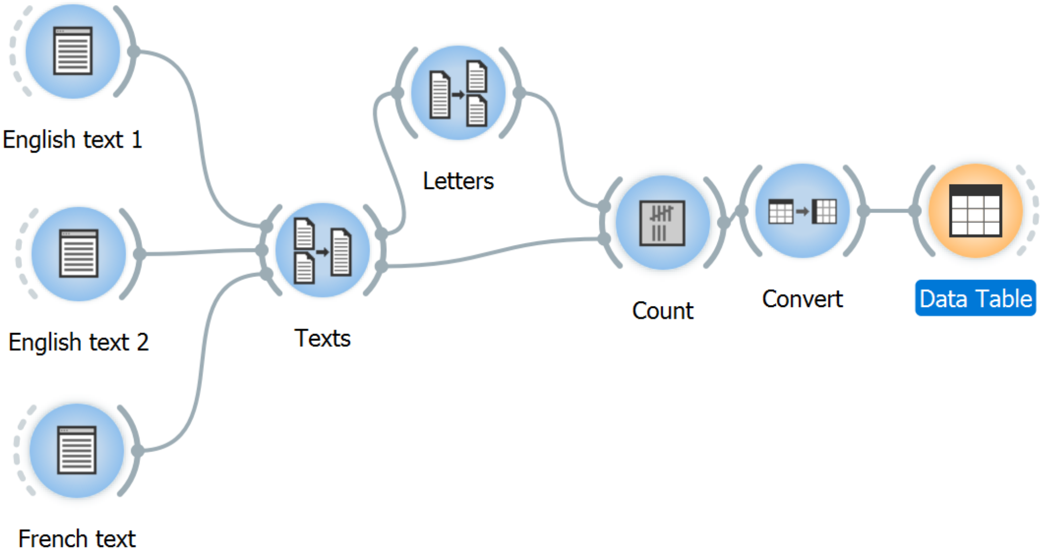
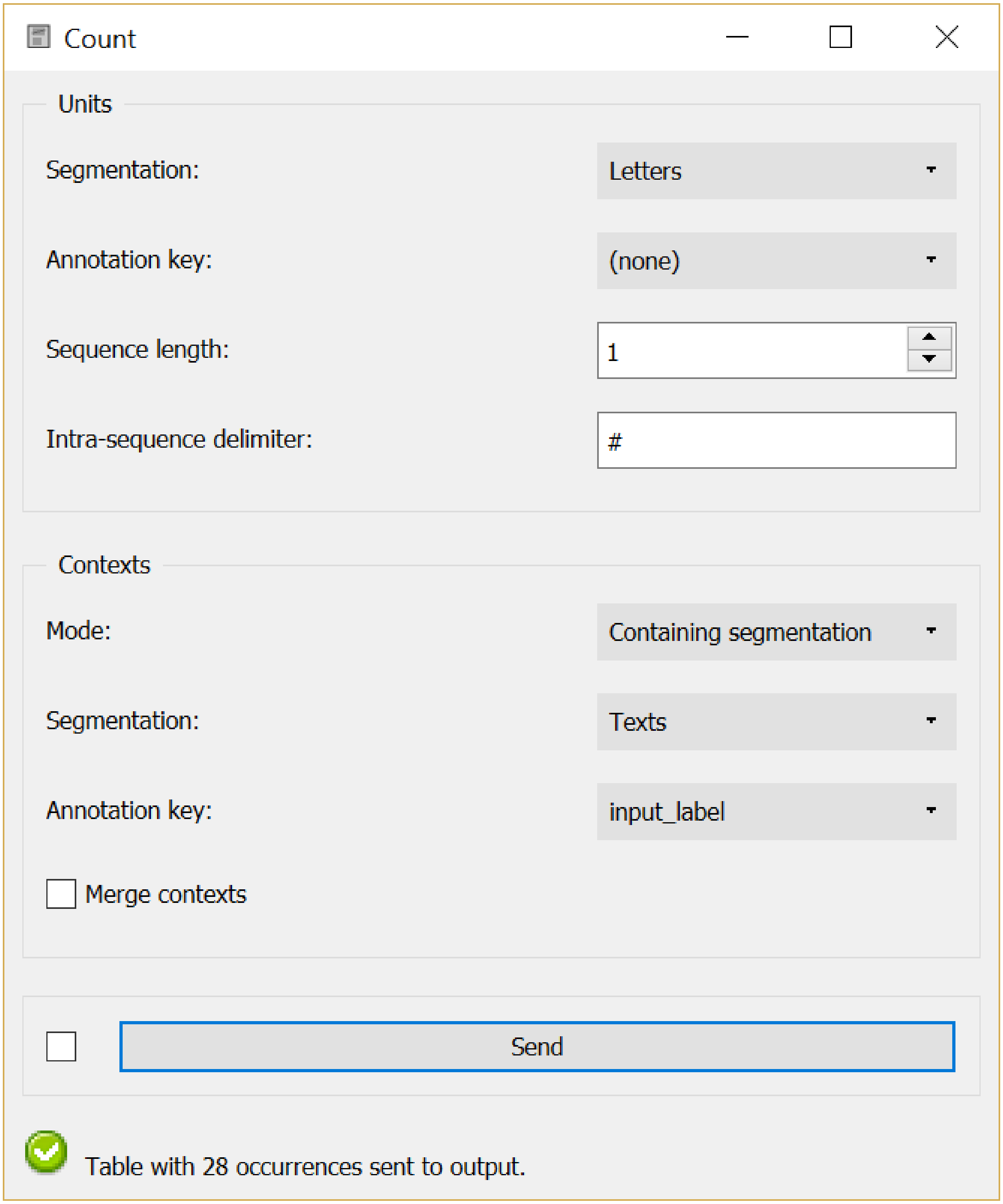


Fig. 31: Figure 1: Schema for counting letter frequency in three texts.

Let us suppose, first, that the instance of *Count* is configured as shown on *figure 2* below, so that the definition of contexts–that is, rows of the frequency table–is based on the content of the three texts.



Units

Segmentation: Letters

Annotation key: (none)

Sequence length: 1

Intra-sequence delimiter: #

Contexts

Mode: Containing segmentation

Segmentation: Texts

Annotation key: input_label

□ Merge contexts

□

Table with 28 occurrences sent to output.

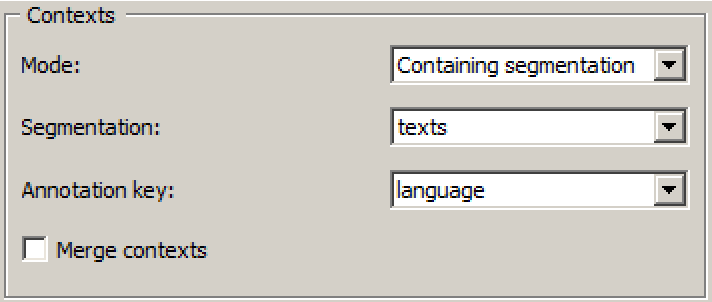
Fig. 32: Figure 2: Counting letter frequency in texts.

Here is the resulting table, disregarding possible variations in row and/or column order:

Table 6: Table 1: Letter frequency in three texts.

	a	t	e	x	i	n	E	g	l	s	h	o	r	u	f	ç
a text in English	1	2	1	1	2	2	1	1	1	1	1	0	0	0	0	0
another text in English	1	3	2	1	2	3	1	1	1	1	2	1	1	0	0	0
un texte en français	2	2	3	1	1	3	0	0	0	1	0	0	1	1	1	1

As can be seen, the default header of each row is the entire content of each text. While this may not be a problem in a pedagogic example like this one, it is easy to see why it would compromise the table's readability in a real application, where texts often contain thousand or even millions of characters. To avoid that, it is useful to tag the table's rows with annotation values attached to segments rather than with these segments' content. To that effect, the desired annotation key must be selected in the **Contexts** section of widget *Count*'s interface.



Contexts

Mode: Containing segmentation

Segmentation: texts

Annotation key: language

□ Merge contexts

Fig. 33: Figure 3: Tagging contexts with annotation values.

In the example of *figure 3* above key *language* has been selected, so that the resulting frequency table looks like this:

Table 7: Table 2: Letter frequency in two text types.

	a	t	e	x	i	n	E	g	l	s	h	o	r	u	f	ç
en	2	5	3	2	4	5	2	2	2	2	3	1	1	0	0	0
fr	2	2	3	1	1	3	0	0	0	1	0	0	1	1	1	1

Besides the substitution of segment content by annotation values in row headers, this example demonstrates an important consequence of this manipulation: contexts associated with the same annotation value are, in effect, collapsed together so that they form a single row. If this behavior is not desired, it can be avoided by assigning distinct annotation values to the contexts that must be kept separated (e.g. *en_1* and *en_2*).

See also

- Getting started: Annotating by merging
- *Reference: Merge widget*
- *Reference: Segment widget*
- *Reference: Count widget*
- *Reference: Table construction widgets*

1.3 Advanced topics

Doing text mining and working on text statistics softwares require the knowledge of various textual formalisms. Those formalisms are useful to make complex queries to text databases. To benefit from the whole potential of Textable, you'll need to learn how to manipulate XML markup and how to use some Regular expressions (Regex).

1.3.1 Converting XML markup to annotations

Often, the best way (and sometimes the only way) to add a specific type of annotation to a text is by "manually" adding it to the data. This is frequently done with XML markup. For instance, the text that appears in the *Text Field* instance of *figure 1* below is segmented into words by means of <*w*> tags whose *type* attribute indicates the "part of speech" associated with each word (e.g. *DET*, *NOUN*, *PREP*, and so on).

The role of widget *Extract XML* is to convert XML markup into annotated segments (in the sense of Orange Textable). In its basic version (see *figure 2* below), the widget's interface essentially requires the user to specify the name of the XML tags that must be imported, namely *w* in this example. The **Remove markup** checkbox indicates whether further markup (if any) detected *within* imported tags must be removed (there is no further markup in this example, so that this option has no effect here).

After connecting the above *Text Field* and *Extract XML* instances, and the latter to an instance of *Display*, the reader can verify that the resulting segmentation contains a segment for the content of each <*w*> tag in the input text, and that this segment is annotated with key *type* and value *DET*, *NOUN*, or *PREP* (the three first such segments are shown on *figure 3* below). Each attribute-value pair of each XML tag has indeed been automatically converted to a *{key: value}* annotation.

See also

- *Reference: Text Field widget*
- *Reference: Extract XML widget*
- *Cookbook: Convert XML tags to Orange Textable annotations*

1.3.2 Merging units with XML annotations

Annotations can also be used for merging *units* (that is, columns) during counting operations in particular. Consider again the example of annotations extracted from XML data developed *here*. The segmentation produced by *Extract XML* can be sent to an instance of *Count* as on the schema shown on *figure 1* below.

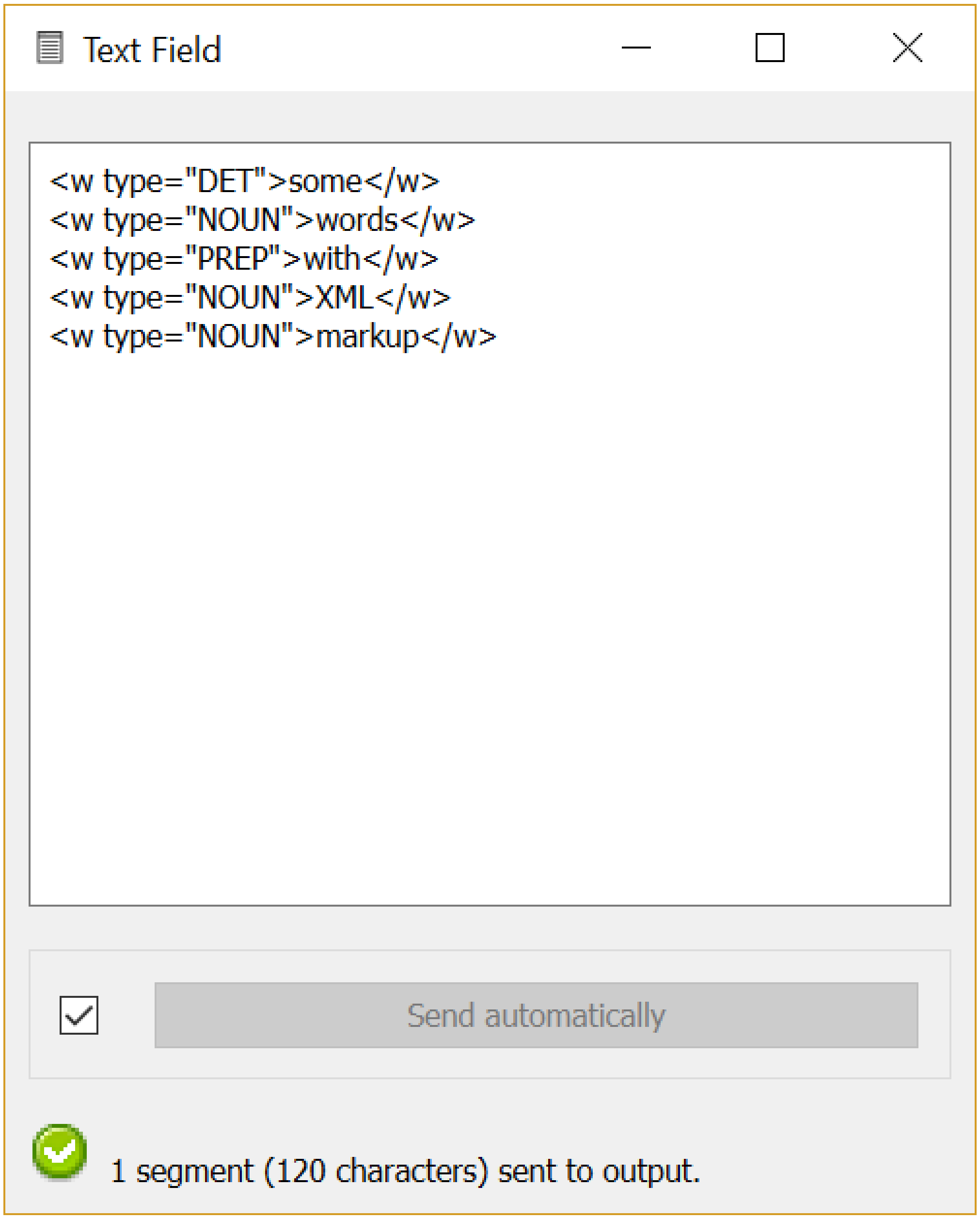
If the *type* annotation key is selected in section **Units** of widget *Count*'s interface (see *figure 2* below), the annotation values correponding to this key (namely part of speech) will be counted in place of the segments' content.

The resulting table is as follows:

Table 8: Table 1: Part of speech frequency.

NOUN	*DET*	*PREP*
3	1	1

Of course, annotations may be used to merge units *and* contexts simultaneously.

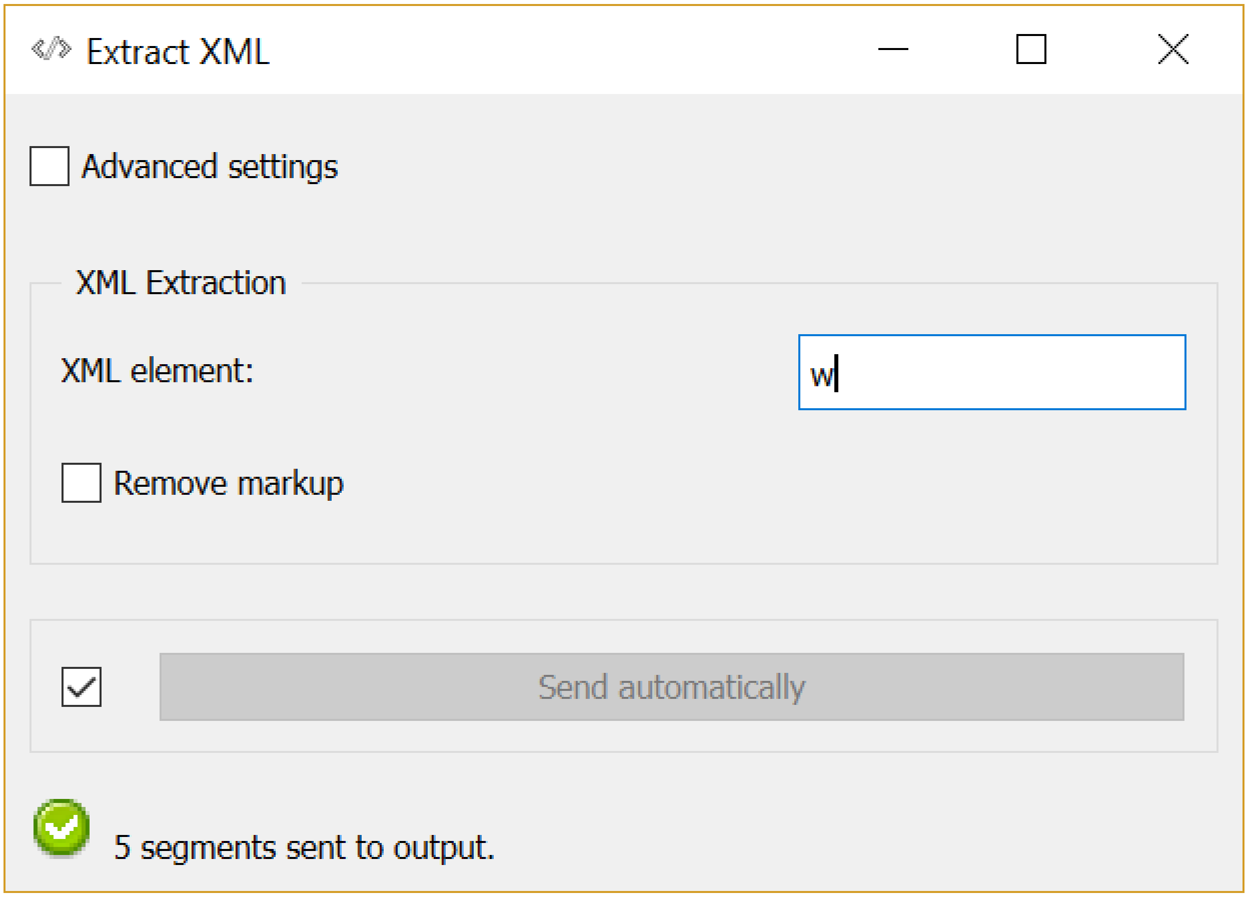


```
<w type="DET">some</w>
<w type="NOUN">words</w>
<w type="PREP">with</w>
<w type="NOUN">XML</w>
<w type="NOUN">markup</w>
```

☑ Send automatically

1 segment (120 characters) sent to output.

Fig. 34: Figure 1: Sample text annotated using XML markup.



□ Advanced settings

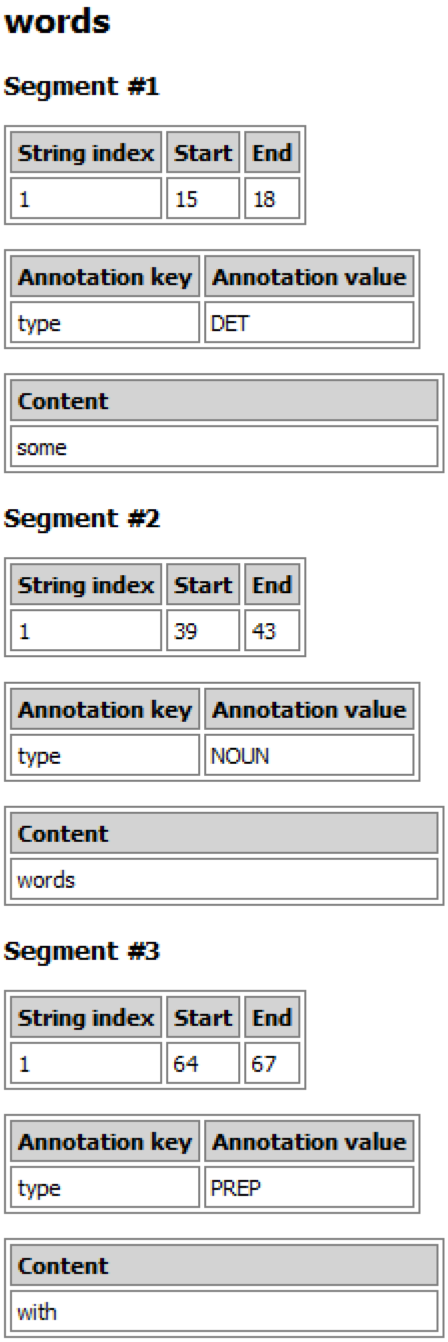
XML element: w

□ Remove markup

☑ Send automatically

5 segments sent to output.

Fig. 35: Figure 2: Interface of the *Extract XML* widget.



words

Segment #1

String index	Start	End
1	15	18

Annotation key	Annotation value
type	DET

Content
some

Segment #2

String index	Start	End
1	39	43

Annotation key	Annotation value
type	NOUN

Content
words

Segment #3

String index	Start	End
1	64	67

Annotation key	Annotation value
type	PREP

Content
with

Fig. 36: Figure 3: Annotations imported using *Extract XML*.

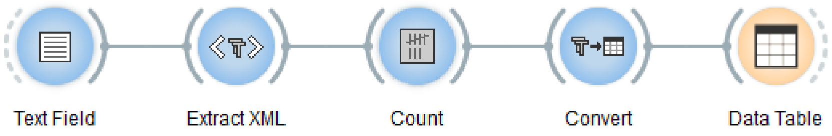
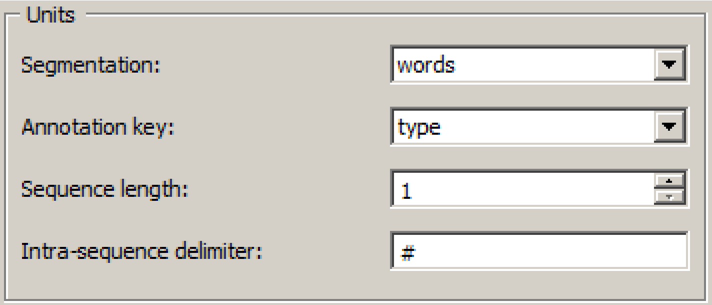


Fig. 37: Figure 1: Counting segments extracted from XML data.



Units

Segmentation: words

Annotation key: type

Sequence length: 1

Intra-sequence delimiter: #

Fig. 38: Figure 2: Merging units using annotation values.

See also

- *Getting started: Converting XML markup to annotations*
- *Reference: Extract XML widget*
- *Reference: Count widget*

1.3.3 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

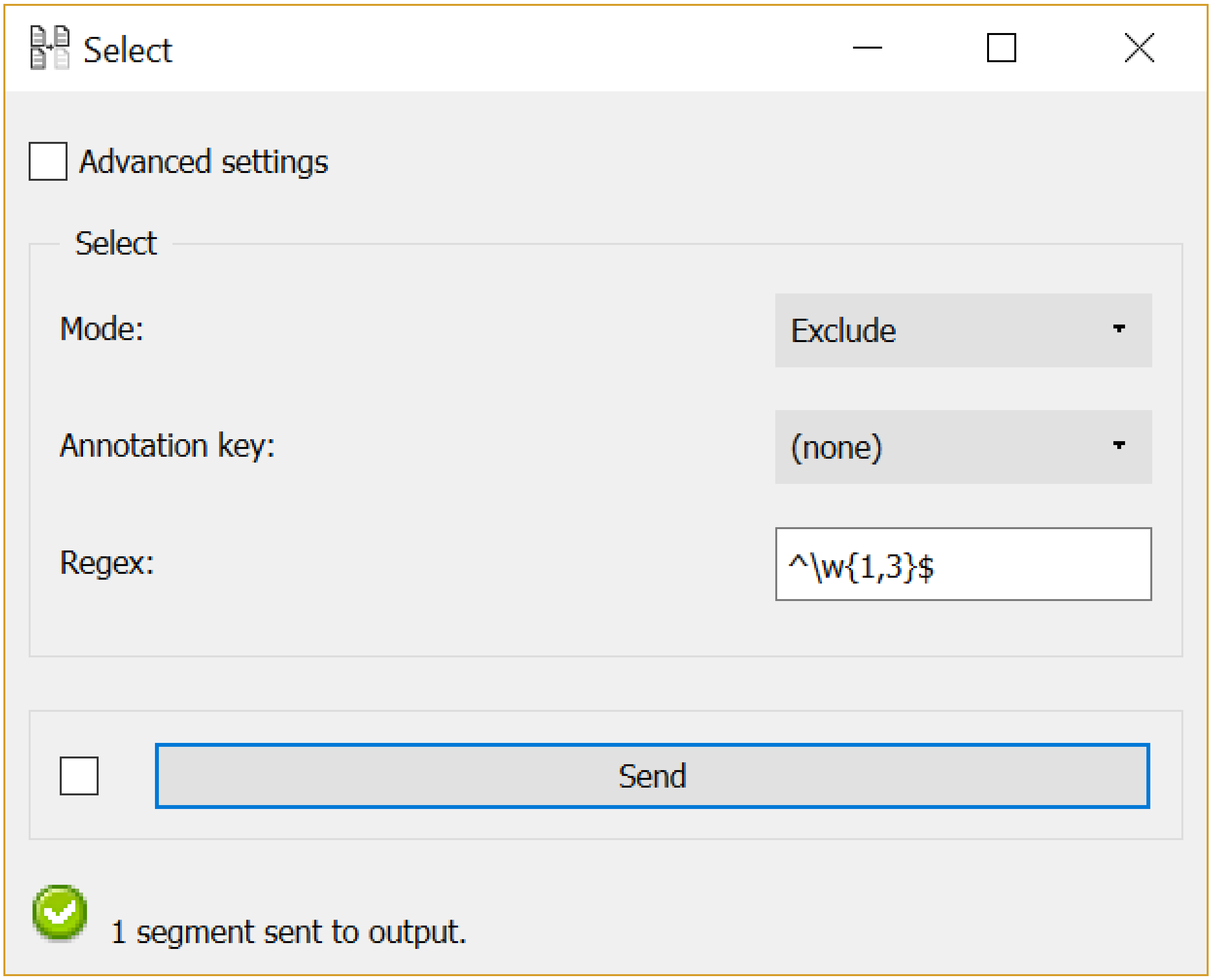
Also recommended are the following:

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

1.3.4 Partitioning segmentations using a regex

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.)



Select

□ Advanced settings

Select

Mode: Exclude

Annotation key: (none)

Regex: ^\w{1,3}$

□ Send

1 segment sent to output.

Fig. 39: 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 connection and select **Reset Signals** (see *figure 2* below).

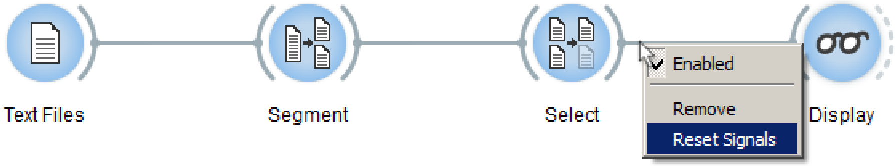


Fig. 40: Figure 2: Right-clicking on a connection and requesting to **Reset Signals**.

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 connection. Clicking **OK** validates the modification and sends the discarded data through the connection.

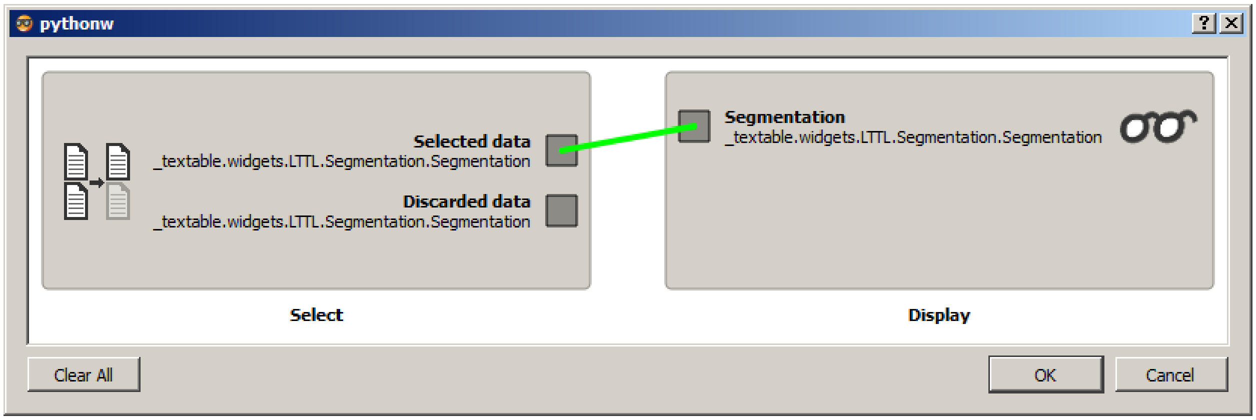

pythonw
?
Selected data
_textable.widgets.LTTL.Segmentation.Segmentation
Discarded data
_textable.widgets.LTTL.Segmentation.Segmentation
Segmentation
_textable.widgets.LTTL.Segmentation.Segmentation
Select
Display
Clear All
OK
Cancel

Fig. 41: Figure 3: This dialog allows the user to select a non-default connection between two widgets.

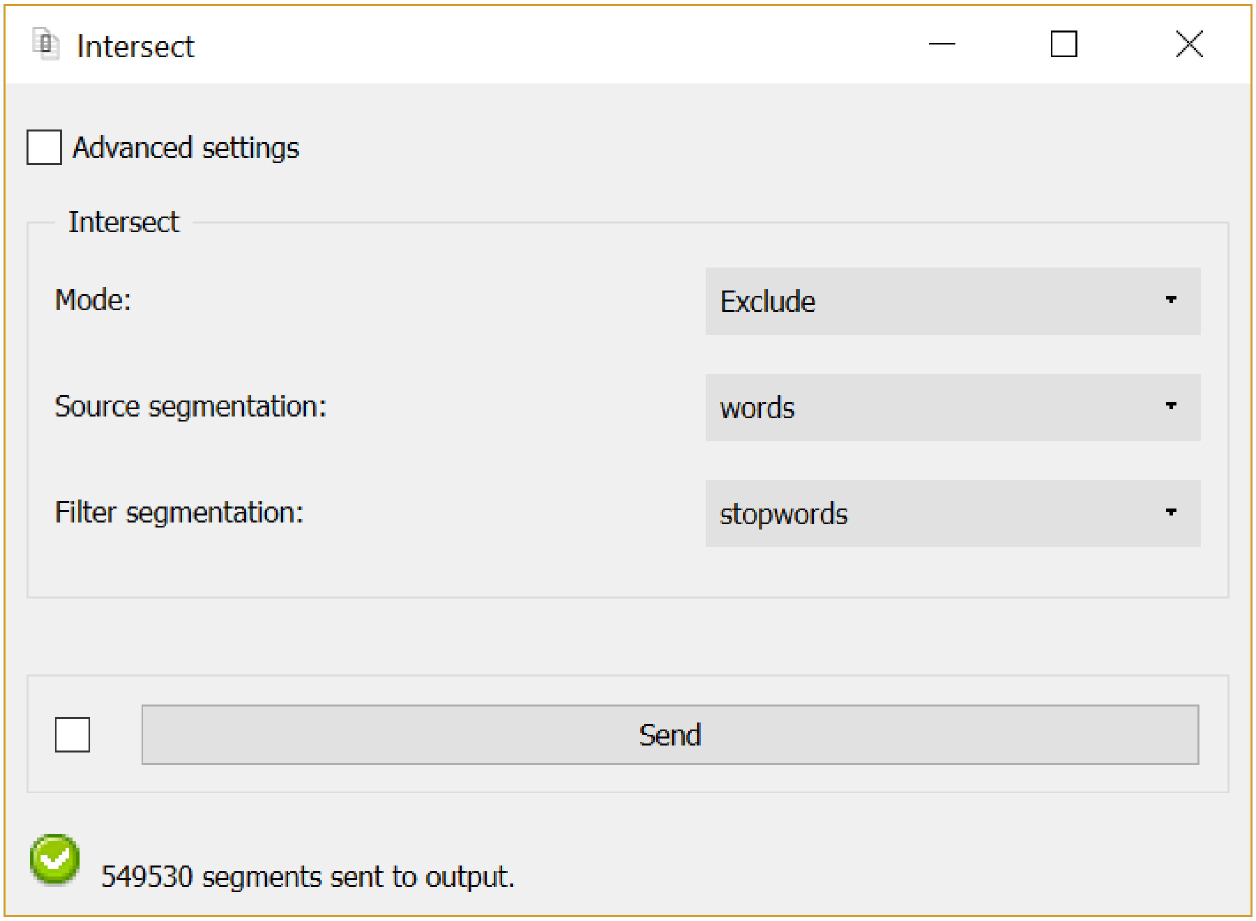
See also

- *Reference: Select widget*
- *Cookbook: Include/exclude segments based on a pattern*

1.3.5 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).



□ Advanced settings

Intersect

Mode: Exclude

Source segmentation: words

Filter segmentation: stopwords

□ Send

549530 segments sent to output.

Fig. 42: Figure 1: Interface of widget *Intersect* configured for stopword removal.

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 schema 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) and is labelled as such, 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 Segment Widget label , i.e. *words* for the segmentation of the UDHR and *stopwords* for the segmentation of *Text Field (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
```

(continues on next page)

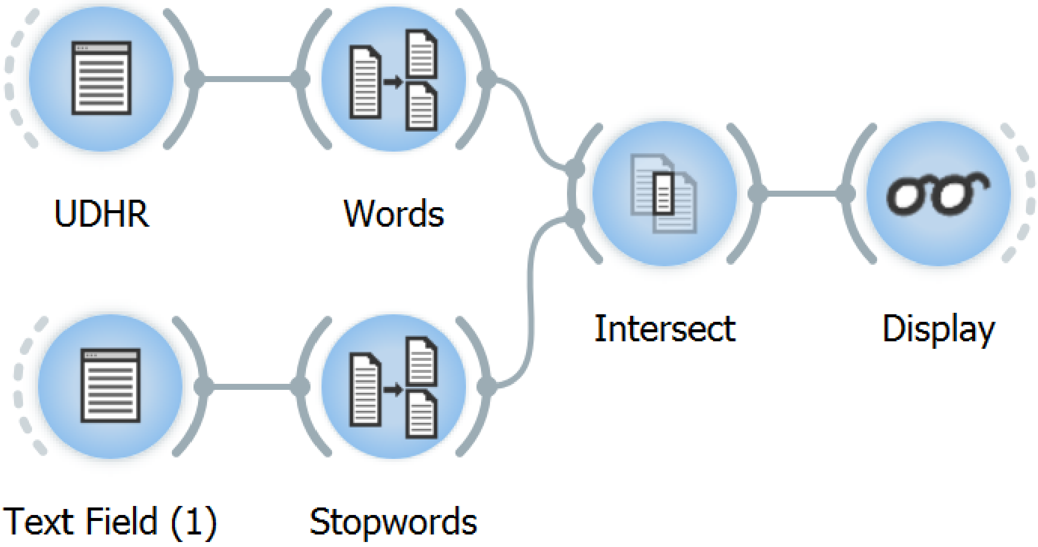


Fig. 43: Figure 2: Example schema for removing stopword using widget *Intersect* .

(continued from previous page)

```
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 schema (*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*) 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.

(*back to the exercise*)

See also

- *Reference: Select widget*
- *Reference: Intersect widget*

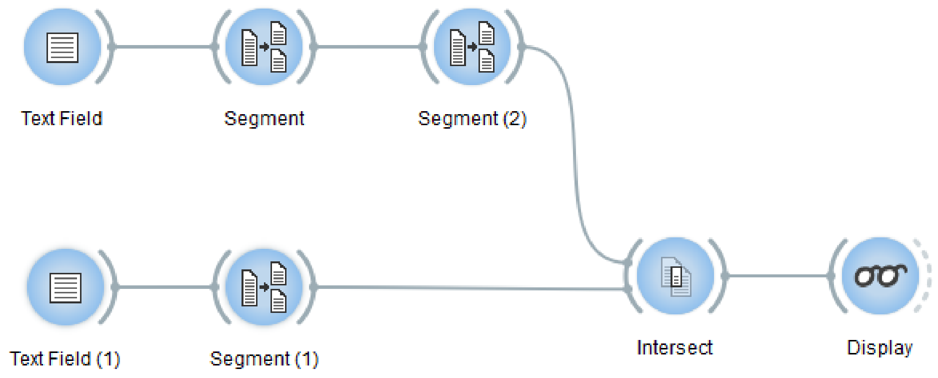


Fig. 44: Figure 3: A possible solution.

- *Cookbook: Exclude segments based on a stoplist*

1.3.6 XML Annotation-based selection using a regex

Another common way of exploiting annotations consists in using them to select the segments that will be in-/excluded by an instance of *Select* (see *Partitioning segmentations*) or *Intersect* (see *Using a segmentation to filter another*). Thus, in the case of the XML data example introduced *here* (and further developed *there*), we might insert an instance of *Select* between those of *Extract XML* and *Count* (see *figure 1* below) in order to include only "content words".

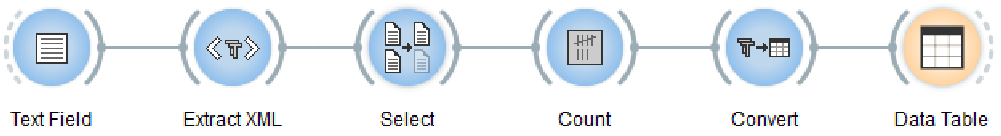
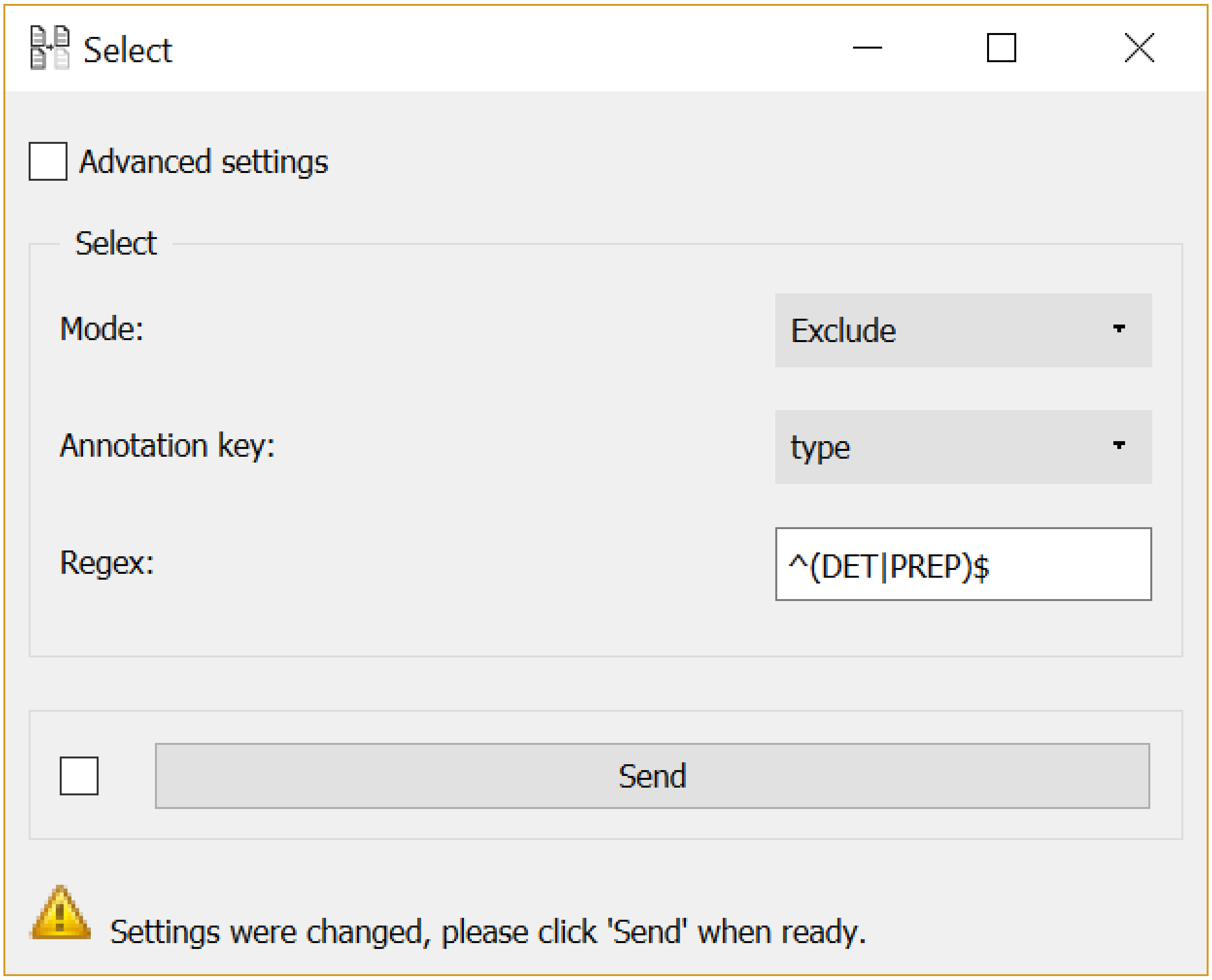


Fig. 45: Figure 1: Inserting an instance of *Select* to filter a segmentation.

In this simplified example, the *Select* instance could thus be parameterized as indicated on *figure 2* below), so as to exclude (**Mode: Exclude**) those segments whose annotation value for key *type* (**Annotation key**: *type*) is *DET* or *PREP* (**Regex:** `^(DET|PREP)$`).

See also

- *Getting started: Partitioning segmentations*
- *Getting started: Using a segmentation to filter another*
- *Getting started: Converting XML markup to annotations*
- *Getting started: Merging units with annotations*
- *Reference: Select widget*
- *Reference: Intersect widget*
- *Reference: Extract XML widget*
- *Reference: Count widget*



Select

☐ Advanced settings

Select

Mode: Exclude

Annotation key: type

Regex: ^(DET|PREP)$

☐ Send

Settings were changed, please click 'Send' when ready.

Fig. 46: Figure 2: Excluding segments based on annotation values with *Select*.

1.4 Cookbook

This section describes how to get a number of basic tasks done with Orange Textable. Each task is explained by means of a concise, illustrated recipe. The goal is to provide the user with a set of elementary operations which, once properly chained, may form the basic skeleton of various more ambitious projects.

1.4.1 Text input

Import text from keyboard

Goal

Input text using keyboard for further processing with Orange Textable.

Ingredients

Widget	*Text Field*
Icon	▤
Quantity	1

Procedure

1. Create an instance of *Text Field* on the canvas.
2. Open its interface by double-clicking on the created instance.
3. Type text in the text field at the top of the interface.
4. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
5. A segmentation covering the input text is then available on the *Text Field* instance's output connections; to display or export it, see *Cookbook: Text output*.

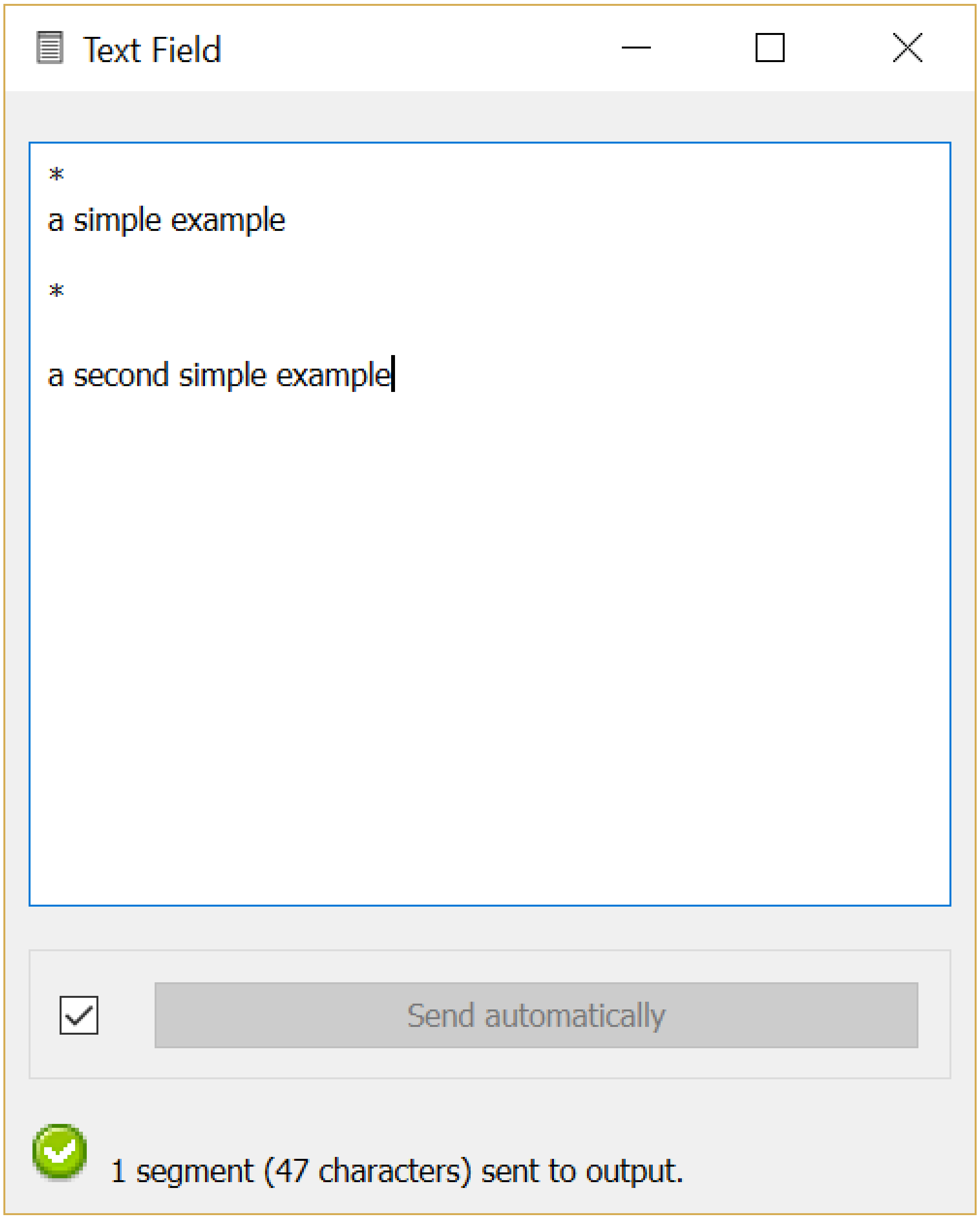
See also

- *Getting started: Keyboard input and segmentation display*
- *Reference: Text Field widget*
- *Cookbook: Text output*

Import text from file

Goal

Import the content of one or more raw text files for further processing with Orange Textable.



```
*
a simple example
*

a second simple example
```

1 segment (47 characters) sent to output.

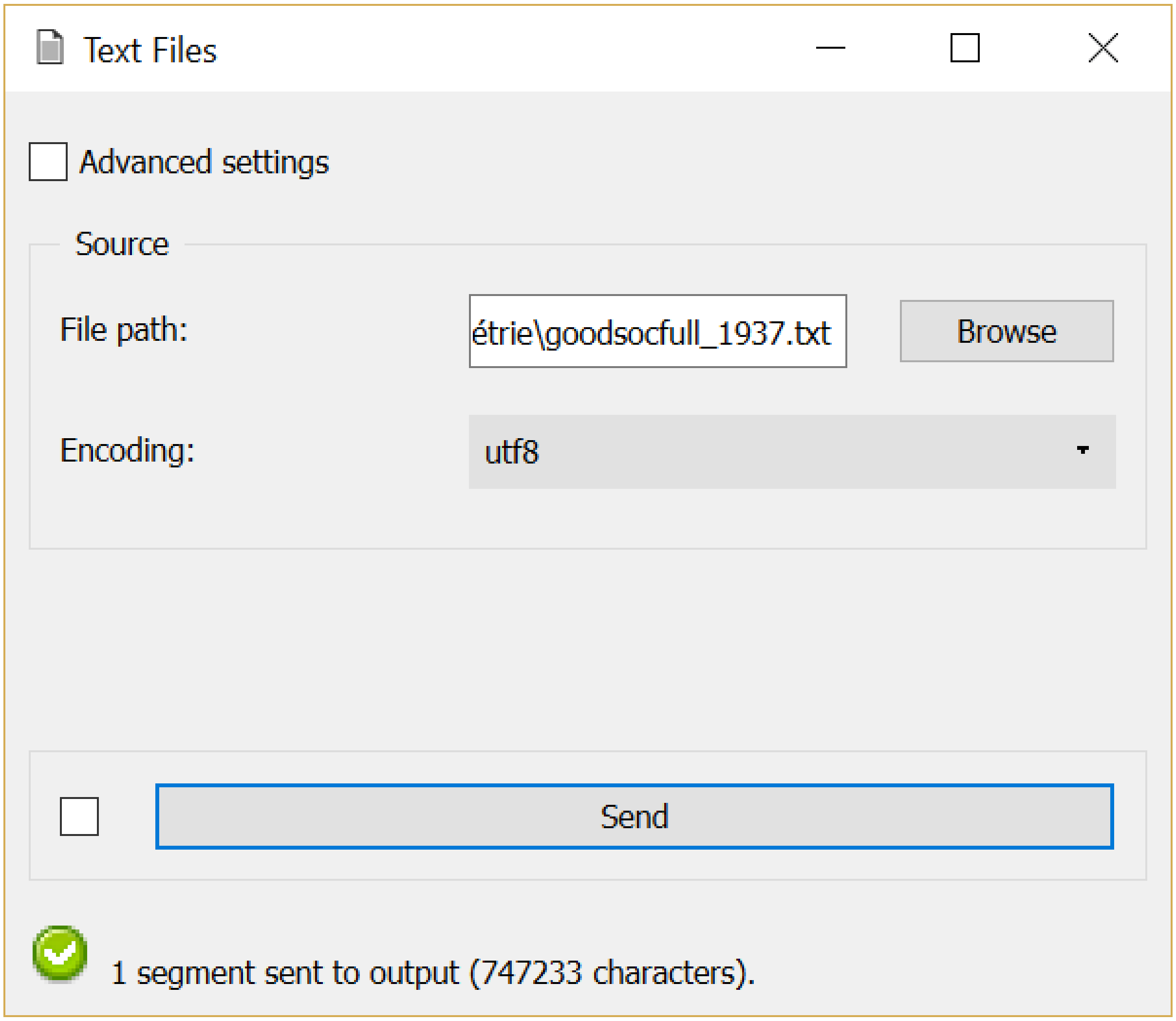
Fig. 47: Figure 1: Importing a string using widget *Text Field*.

Ingredients

Widget	*Text Files*
Icon	🗎
Quantity	1

Procedure

Single file



□ Advanced settings

File path: étrie\goodsocfull_1937.txt

Encoding: utf8

□ Send

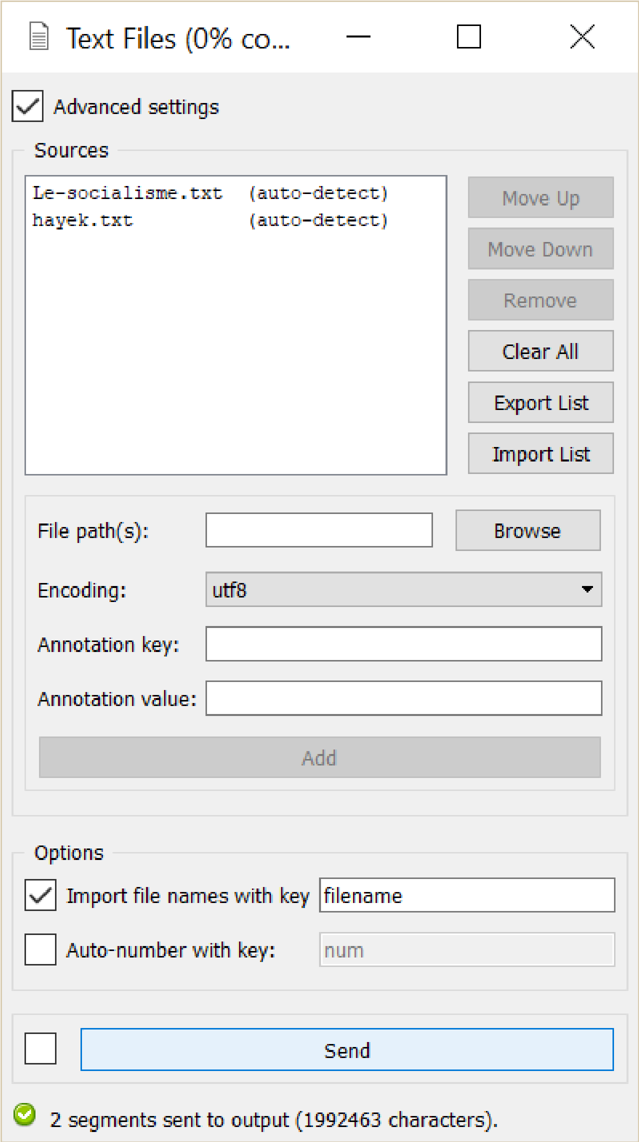
1 segment sent to output (747233 characters).

Fig. 48: Figure 1: Importing the content of a file using the *Text Files* widget.

1. Create an instance of *Text Files* on the canvas.
2. Open its interface by double-clicking on the created instance.

3. Make sure the **Advanced settings** checkbox is *not* selected.
4. Click the **Browse** button to open the file selection dialog.
5. Select the file you want to import and close the file selection dialog by clicking **Ok**.
6. In the **Encoding** drop-down menu, select the encoding that corresponds to your file.
7. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
8. A segmentation covering the file's content is then available on the *Text Files* instance's output connections; to display or export it, see *Cookbook: Text output*.

Multiple files



Text Files (0% co...

☑ Advanced settings

Sources

```
Le-socialisme.txt   (auto-detect)
hayek.txt           (auto-detect)
```

File path(s):

Encoding: utf8

Annotation key:

Annotation value:

Options

☑ Import file names with key filename

□ Auto-number with key: num

□ Send

2 segments sent to output (1992463 characters).

Fig. 49: Figure 2: Importing the content of several files using the *Text Files* widget.

1. Create an instance of *Text Files* on the canvas.

2. Open its interface by double-clicking on the created instance.
3. Make sure the **Advanced settings** checkbox *is* selected.
4. If needed, empty the list of imported files by clicking the **Clear all** button.
5. Click the **Browse** button to open the file selection dialog.
6. Select the first file you want to import. Select the encoding that corresponds to your file (if unknown, choose auto-detect in **Encoding**) .
7. Click the **Add** button to add your first file to the list of imported files.
8. Repeat steps 5 to 7 for adding all your files.
9. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
10. A segmentation containing a segment covering each imported file's content is then available on the *Text Files* instance's output connections; to display or export it, see *Cookbook: Text output*.

See also

- *Reference: Text Files widget*
- *Cookbook: Text output*

Import text from internet location

Goal

Import text content located at one or more URLs for further processing with Orange Textable.

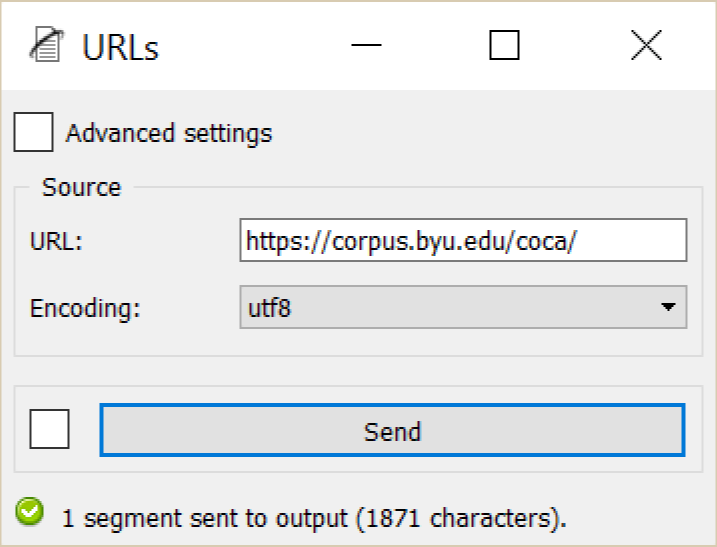
Ingredients

Widget	*URLs*
Icon	http
Quantity	1

Procedure

Single URL

1. Create an instance of *URLs* on the canvas.
2. Open its interface by double-clicking on the created instance.
3. Make sure the **Advanced settings** checkbox is *not* selected.
4. In the **URL** field, type the URL whose content you want to import (including the `http://` prefix).
5. In the **Encoding** drop-down menu, select the encoding that corresponds to this URL.
6. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).



URL: https://corpus.byu.edu/coca/

Encoding: utf8

1 segment sent to output (1871 characters).

Fig. 50: Figure 1: Importing text from an internet location using the *URLs* widget.

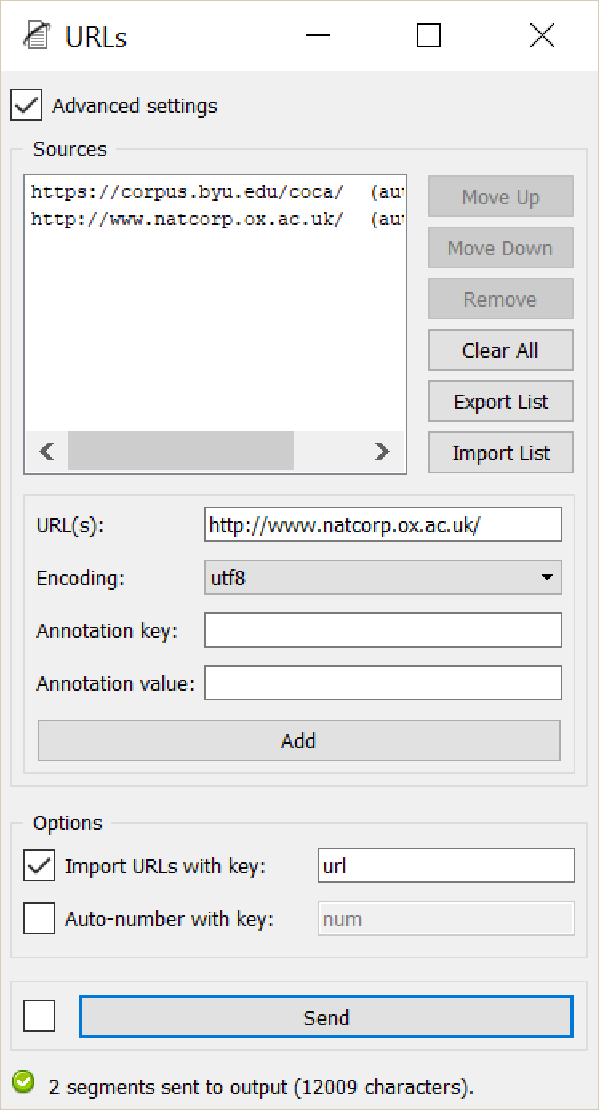
7. A segmentation covering the URL's content is then available on the *URLs* instance's output connections; to display or export it, see *Cookbook: Text output*.

Multiple URLs

1. Create an instance of *URLs* on the canvas.
2. Open its interface by double-clicking on the created instance.
3. Make sure the **Advanced settings** checkbox *is* selected.
4. If needed, empty the list of imported URLs by clicking the **Clear all** button.
5. In the **URL(s)** field, enter the URLs you want to import (including the `http://` prefix), separated by the string " / " (space + slash + space); make sure they all have the same encoding (you will be able to add URLs that have other encodings later).
6. In the **Encoding** drop-down menu, select the encoding that corresponds to the set of selected URLs.
7. Click the **Add** button to add the set of selected URLs to the list of imported URLs.
8. Repeat steps 5 to 7 for adding URLs in other encoding(s).
9. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
10. A segmentation containing a segment covering each imported URL's content is then available on the *URLs* instance's output connections; to display or export it, see *Cookbook: Text output*.

See also

- *Reference: URLs widget*
- *Cookbook: Text output*



☑ Advanced settings

Sources

```
https://corpus.byu.edu/coca/   (au
http://www.natcorp.ox.ac.uk/   (au
```

URL(s): http://www.natcorp.ox.ac.uk/

Encoding: utf8

Annotation key:

Annotation value:

Options

☑ Import URLs with key: url

□ Auto-number with key: num

□

2 segments sent to output (12009 characters).

Fig. 51: Figure 2: Importing text from several internet locations using the *URLs* widget.

1.4.2 Text output

Display text content

Goal

Display the content of a text (segmentation).

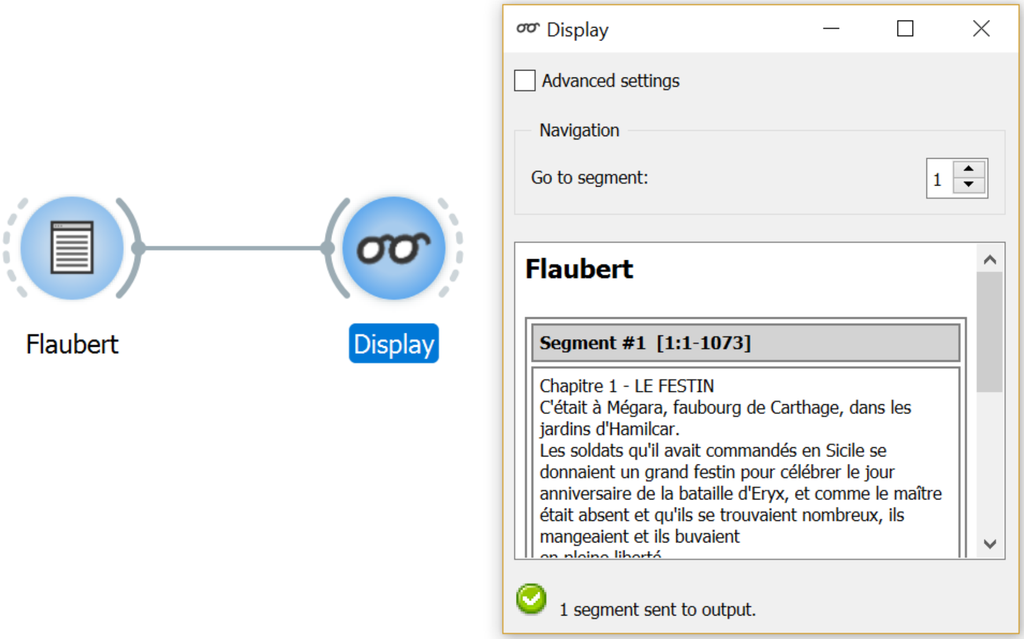
Prerequisites

Some text has been imported in Orange Textable (see *Cookbook: Text input*) and possibly further processed (see *Cookbook: Segmentation manipulation*).

Ingredients

Widget	*Display*
Icon	👓
Quantity	1

Procedure



Display

☐ Advanced settings

Navigation

Go to segment: 1

Flaubert

Segment #1 [1:1-1073]

Chapitre 1 - LE FESTIN
C'était à Mégara, faubourg de Carthage, dans les jardins d'Hamilcar.
Les soldats qu'il avait commandés en Sicile se donnaient un grand festin pour célébrer le jour anniversaire de la bataille d'Eryx, et comme le maître était absent et qu'ils se trouvaient nombreux, ils mangeaient et ils buvaient
en pleine liberté

1 segment sent to output.

Fig. 52: Figure 1: Viewing text with an instance of *Display*.

1. Create an instance of *Display* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that emits the segmentation to be displayed (e.g. *Text Field*) to the *Display* instance's input connection (lefthand side).
3. Open the *Display* instance's interface by double-clicking on its icon on the canvas to view the text content.

Comment

- If the input data consist of a large number of segments (thousands or more), the time necessary to display them can be prohibitively long.

See also

- *Getting started: Keyboard input and segmentation display*
- *Reference: Display widget*
- *Cookbook: Text input*
- *Cookbook: Segmentation manipulation*

Export text content (and/or change text encoding)

Goal

Export the content of a text (segmentation).

Prerequisites

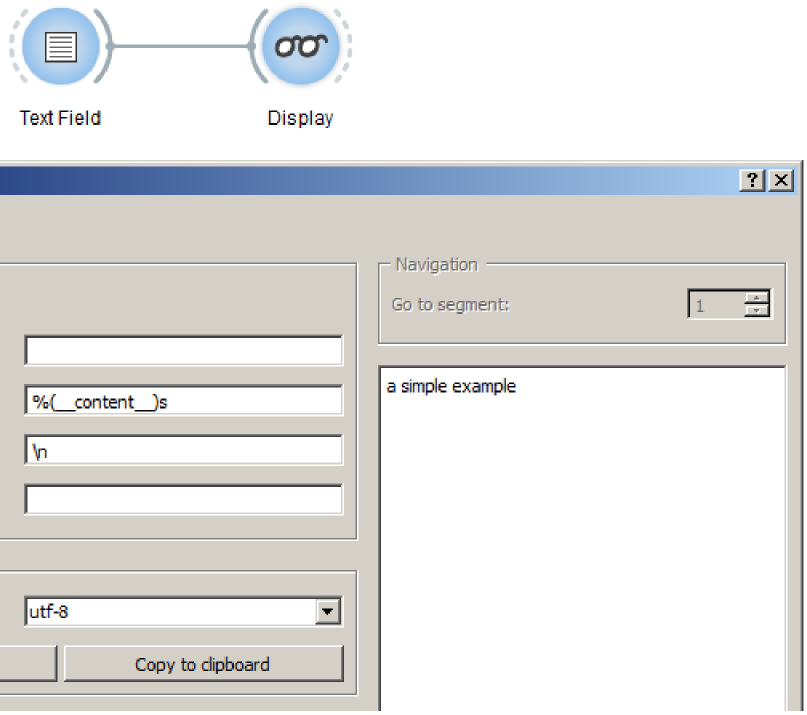
Some text has been imported in Orange Textable (see *Cookbook: Text input*) and possibly further processed (see *Cookbook: Segmentation manipulation*).

Ingredients

Widget	*Display*
Icon	👓
Quantity	1

Procedure

1. Create an instance of *Display* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that emits the segmentation to be displayed (e.g. *Text Field*) to the *Display* instance's input connection (lefthand side).
3. Open the *Display* instance's interface by double-clicking on its icon on the canvas to view the imported text.
4. Tick the **Advanced settings** checkbox.



☑ Advanced settings

Formatting

☑ Apply custom formatting

Header:

Format: %(__content__)s

Segment delimiter: \n

Footer:

Export

File encoding: utf-8

Go to segment: 1

a simple example

Fig. 53: Figure 1: Export text with an instance of *Display*.

5. In the **Formatting** section, tick the **Apply custom formatting** checkbox.
6. In the **Export** section, you can choose the encoding for the text that will be exported using the **File encoding** drop-down menu.
7. Click on **Export to file** button to open the file selection dialog.
8. Select the location you want to export your file to and close the file selection dialog by clicking on **Ok**.

Comment

- If you rather want to *copy* the text content in order to later paste it in another program, click on **Copy to clipboard**; note that in this case, the encoding is by default utf8 and cannot be changed.
- If the input data contains *several* texts (segments) you can specify a string that will be inserted between each successive text in **Segment delimiter**; note that the default segment delimiter `\n` represents a carriage return.
- If the input data consist of a large number of segments (thousands or more), the time necessary to display them can be prohibitively long.

See also

- *Reference: Display widget*
- *Cookbook: Text input*
- *Cookbook: Segmentation manipulation*

1.4.3 Text preprocessing and recoding

Convert text to lower or upper case

Goal

Convert text to lower or upper case.

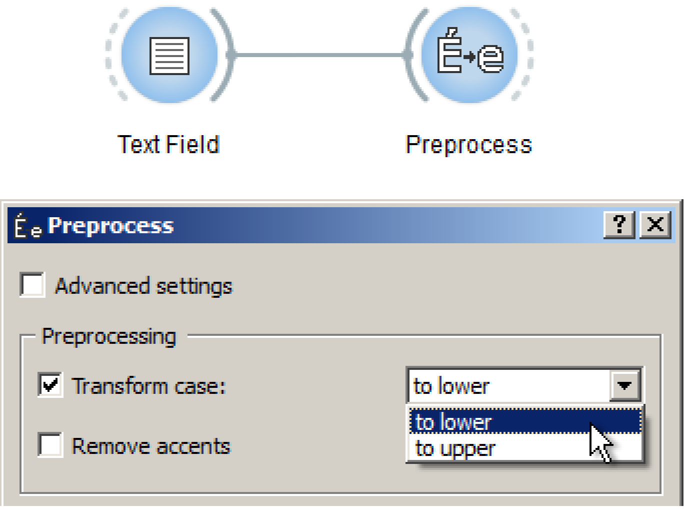
Prerequisites

Some text has been imported in Orange Textable (see *Cookbook: Text input*) and possibly further processed (see *Cookbook: Segmentation manipulation*).

Ingredients

Widget	*Preprocess*
Icon	É→e
Quantity	1

Procedure



Preprocess

☐ Advanced settings

Preprocessing

☑ Transform case: to lower

☐ Remove accents

Fig. 54: Figure 1: Convert text to lower or upper case with an instance of *Preprocess*.

1. Create an instance of *Preprocess* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that emits the segmentation to be modified (e.g. *Text Field*) to the *Preprocess* instance's input connection (lefthand side).
3. Open the *Preprocess* instance's interface by double-clicking on its icon on the canvas.
4. In the **Processing** section, tick the **Transform case** checkbox.

5. Choose **to lower** or **to upper** in the drop-down menu on the right.
6. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
7. A segmentation containing the modified text is then available on the *Preprocess* instance's output connections; to display or export it, see *Cookbook: Text output*.

See also

- *Reference: Preprocess widget*
- *Cookbook: Text input*
- *Cookbook: Segmentation manipulation*
- *Cookbook: Text output*

Remove accents from text

Goal

Remove all accents from text.

Prerequisites

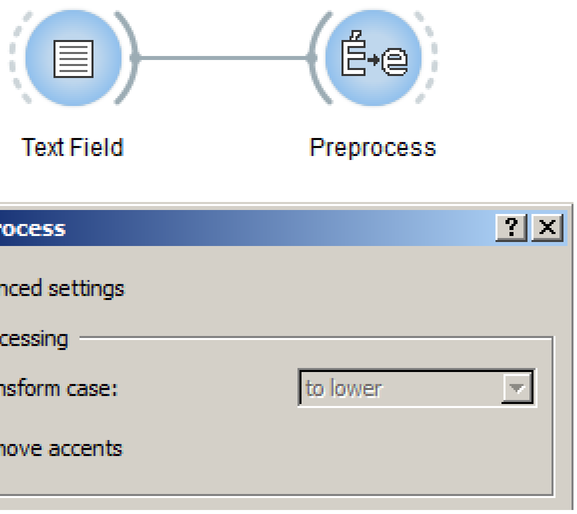
Some text has been imported in Orange Textable (see *Cookbook: Text input*) and possibly further processed (see *Cookbook: Segmentation manipulation*).

Ingredients

Widget	*Preprocess*
Icon	É→e
Quantity	1

Procedure

1. Create an instance of *Preprocess* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that that emits the segmentation to be modified (e.g. *Text Field*) to the *Preprocess* instance's input connection (lefthand side).
3. Open the *Preprocess* instance's interface by double-clicking on its icon on the canvas.
4. In the **Processing** section, tick the **Remove accents** checkbox.
5. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
6. A segmentation containing the modified text is then available on the *Preprocess* instance's output connections; to display or export it, see *Cookbook: Text output*.



Preprocess

□ Advanced settings

Preprocessing

□ Transform case: to lower

☑ Remove accents

Fig. 55: Figure 1: Remove accents from text with an instance of *Preprocess*.

See also

- *Reference: Preprocess widget*
- *Cookbook: Text input*
- *Cookbook: Segmentation manipulation*
- *Cookbook: Text output*

Replace all occurrences of a string/pattern

Goal

Replace all occurrences of a string (or pattern) in a text with another string.

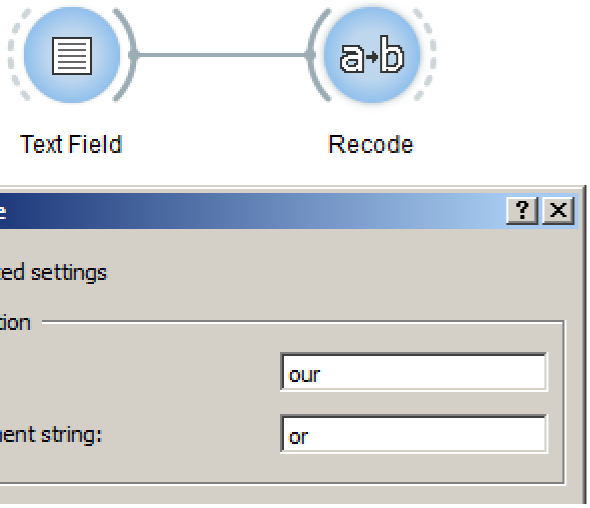
Prerequisites

Some text has been imported in Orange Textable (see *Cookbook: Text input*) and possibly further processed (see *Cookbook: Segmentation manipulation*).

Ingredients

Widget	*Recode*
Icon	a→b
Quantity	1

Procedure



ab Recode

□ Advanced settings

Substitution

Regex: our

Replacement string: or

Fig. 56: Figure 1: Replace all occurrences of a string with an instance of *Recode*.

1. Create an instance of *Recode* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that emits the segmentation to be modified (e.g. *Text Field*) to the *Recode* instance's input connection (lefthand side).
3. Open the *Recode* instance's interface by double-clicking on its icon on the canvas.
4. In the **Substitution** section, insert the string that will be replaced in the **Regex** field.
5. In the **Replacement string** field insert the replacement string.
6. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
7. A segmentation containing the modified text is then available on the *Recode* instance's output connections; to display or export it, see *Cookbook: Text output*.

Comment

- In the **Regex** field you can use all the syntax of Python's regular expression (*cf.* Python documentation).
- In our example, we choose to replace all occurrences of British *-our* with American *-or* (for example, from *colour* to *color*); unless otherwise specified (typically using word boundary "anchor" `\b`), replacements will also occur within words, i.e. *coloured* to *colored*.

See also

- *Reference: Recode widget*
- *Cookbook: Text input*
- *Cookbook: Segmentation manipulation*
- *Cookbook: Text output*

1.4.4 Segmentation manipulation

Segment text in smaller units

Goal

Segment text in smaller units (e.g. lines, words, letters, etc.).

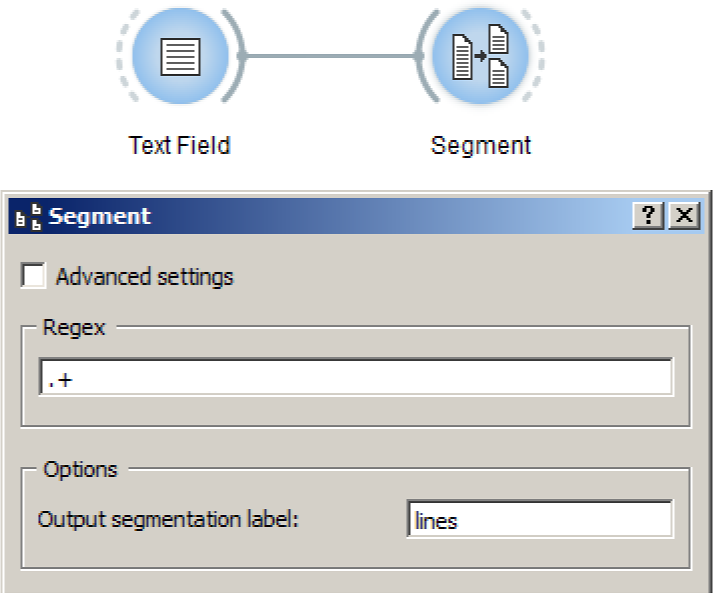
Prerequisites

Some text has been imported in Orange Textable (see *Cookbook: Text input*) and possibly further processed (see *Cookbook: Segmentation manipulation*).

Ingredients

Widget	*Segment*
Icon	
Quantity	1

Procedure



Segment

□ Advanced settings

Regex

.+

Options

Output segmentation label: lines

Fig. 57: Figure 1: Segment text in lines with an instance of *Segment*.

1. Create an instance of *Segment* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that emits the segmentation to be segmented (e.g. *Text Field*) to the *Segment* instance's input connection (lefthand side).

3. Open the *Segment* instance's interface by double-clicking on its icon on the canvas.
4. In the **Regex** section, insert the regular expression describing the units that will be segmented (for example to segment a text in lines use `.+`, in words `\w+`, in letters `\w`, in characters `.`, and so on) then click on the validation button on the right.
5. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
6. A segmentation containing a segment for each specified unit (e.g. line) is then available on the *Segment* instance's output connections; to display or export it, see *Cookbook: Text output*.

Comment

- In the **Regex** field you can use all the syntax of Python's regular expression (*cf.* Python documentation).

See also

- *Getting started: Segmenting data into smaller units*
- *Reference: Segment widget*
- *Cookbook: Text input*
- *Cookbook: Segmentation manipulation*
- *Cookbook: Text output*

Merge several texts

Goal

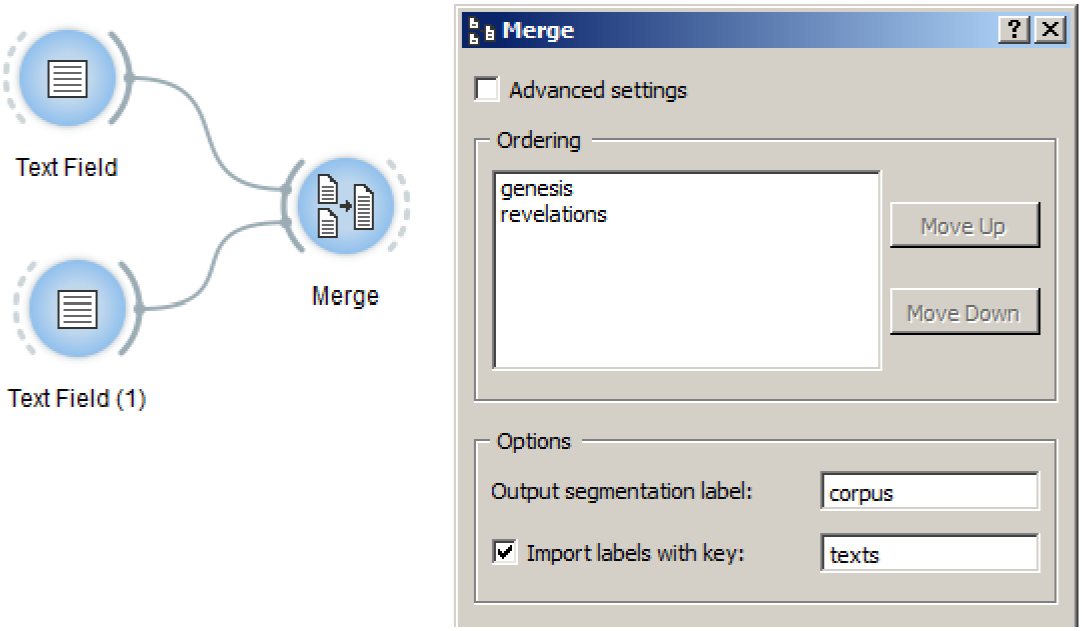
Merge several texts together so they can be further processed as a whole.

Prerequisites

Two or more text have been imported in Orange Textable (see *Cookbook: Text input*) and possibly further processed (see *Cookbook: Segmentation manipulation*).

Ingredients

Widget	*Merge*
Icon	
Quantity	1



□ Advanced settings

Ordering

genesis
revelations

Options

Output segmentation label: corpus

☑ Import labels with key: texts

Fig. 58: Figure 1: Merge several texts with an instance of *Merge*

Procedure

1. Create an instance of *Merge* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instances that emit the segmentations to be merged together (e.g. two instances of *Text Field*) to the *Merge* instance's input connection (lefthand side).
3. Open the *Merge* widget instance's interface by double-clicking on its icon on the canvas.
4. All input data appear in the **Ordering** section; you can change their ordering by selecting a line and clicking on **Move Up** or **Move Down**.
5. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
6. A segmentation containing all input data merged together is then available on the *Merge* instance's output connections; to display or export it, see *Cookbook: Text output*.

See also

- Getting started: Merging segmentations together
- *Reference: Merge widget*
- *Cookbook: Text input*
- *Cookbook: Segmentation manipulation*
- *Cookbook: Text output*

Include/exclude segments based on a pattern

Goal

Include or exclude segments from a segmentation using a regular expression

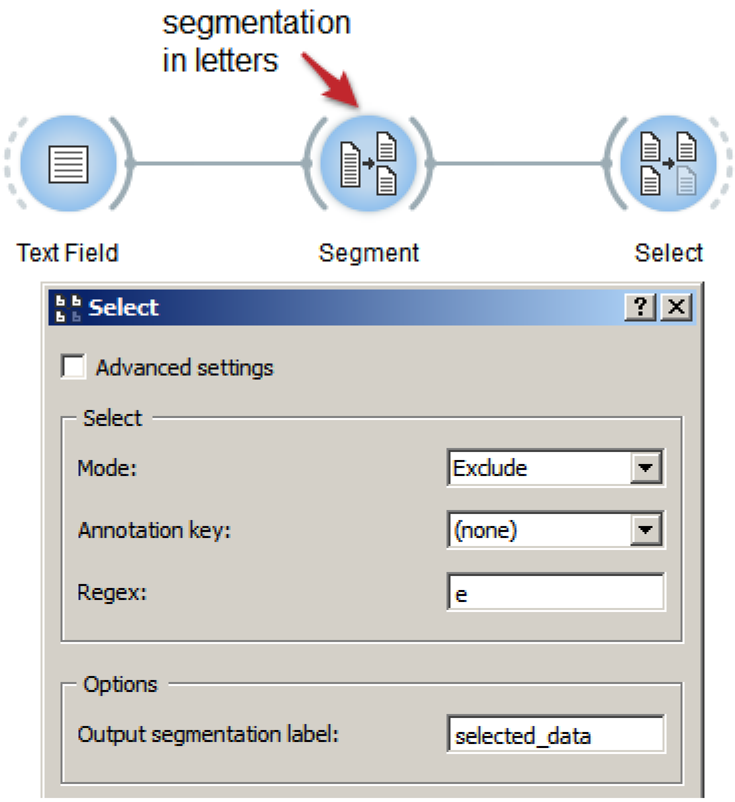
Prerequisites

Some text has been imported in Orange Textable (see *Cookbook: Text input*) and in all likelihood it has been segmented in smaller units (see *Cookbook: Segment text in smaller units*).

Ingredients

Widget	*Select*
Icon	
Quantity	1

Procedure



Select

☐ Advanced settings

Select

Mode: Exclude

Annotation key: (none)

Regex: e

Options

Output segmentation label: selected_data

Fig. 59: Figure 1: Using the *Select* widget to include/exclude segments from a segmentation based on a regular expression

1. Create an instance of *Select* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that emits the segmentation to be filtered (e.g. an instance of *Segment*) to the *Select* instance's input connection (lefthand side).
3. Open the *Select* instance's interface by double-clicking on its icon on the canvas.
4. In the **Select** section, choose either **Mode: Include** or **Exclude**.
5. In the **Regex** field, insert the pattern that will select the units to be included or excluded, such as the single letter `e` in our example.
6. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
7. A segmentation containing the selected segments is then available on the *Select* instance's output connections; to display or export it, see *Cookbook: Text output*.

Comment

- In the **Regex** field you can use all the syntax of Python's regular expression (*cf.* Python documentation).
- The *Select* widget emits on a second output connection (not selected by default) a segmentation containing the segments that were *not* selected.

See also

- *Getting started: Partitioning segmentations*
- *Reference: Select widget*
- *Cookbook: Text input*
- *Cookbook: Segment text in smaller units*
- *Cookbook: Text output*

Filter segments based on their frequency

Goal

Filter out the most rare and/or frequent segments of a segmentation.

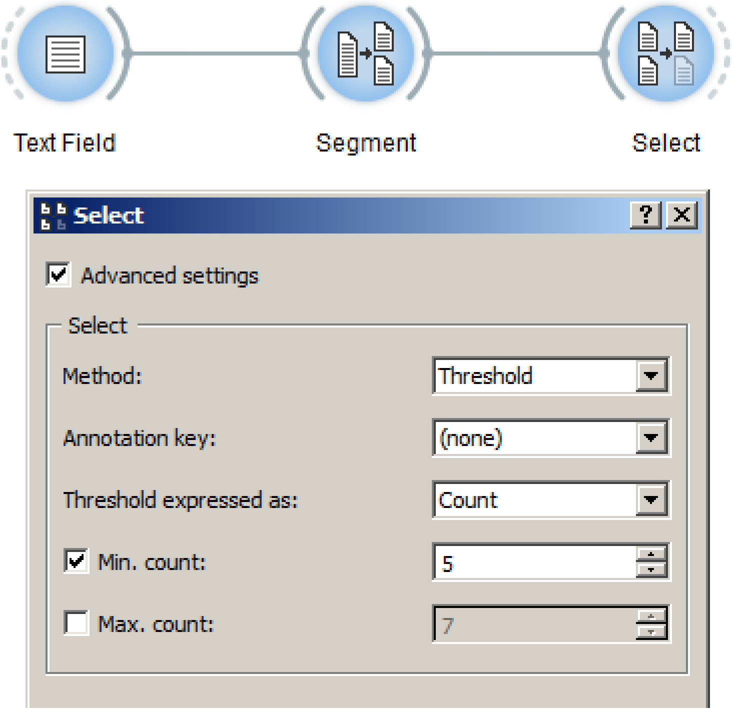
Prerequisites

Some text has been imported in Orange Textable (see *Cookbook: Text input*) and in all likelihood it has been segmented in smaller units (see *Cookbook: Segment text in smaller units*).

Ingredients

Widget	*Select*
Icon	
Quantity	1

Procedure



☑ Advanced settings

Select

Method: Threshold

Annotation key: (none)

Threshold expressed as: Count

☑ Min. count: 5

☐ Max. count: 7

Fig. 60: Figure 1: Filtering out low-frequency segments with an instance of *Select*

1. Create an instance of *Select* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that emits the segmentation to be filtered (e.g. an instance of *Segment*) to the *Select* instance's input connection (lefthand side).
3. Open the *Select* instance's interface by double-clicking on its icon on the canvas.
4. Tick the **Advanced settings** checkbox.
5. In the **Select** section, choose **Threshold** in the **Method** drop-down menu.
6. Under **Threshold expressed as**, choose whether you want to express frequency thresholds in terms of **Count** (i.e. number of tokens) or of **Proportion** (i.e. percentage of tokens).
7. If you want to set a minimum frequency threshold, tick the **Min. count** (respectively **Min. proportion (%)**) checkbox and indicate the minimum frequency that a segment type must have in order to be included in the output.
8. If you want to set a maximum frequency threshold, tick the **Max. count** (respectively **Max. proportion (%)**) checkbox and indicate the maximum frequency that a segment type can have in order to be included in the output.
9. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
10. A segmentation containing the selected segments is then available on the *Select* instance's output connections; to display or export it, see *Cookbook: Text output*.

Comment

- The *Select* widget emits on a second output connection (not selected by default) a segmentation containing the segments that were *not* selected.

See also

- *Reference: Select widget*
- *Cookbook: Text input*
- *Cookbook: Segment text in smaller units*
- *Cookbook: Text output*

Create a random selection or sample of segments

Goal

Create a random sample of segments.

Prerequisites

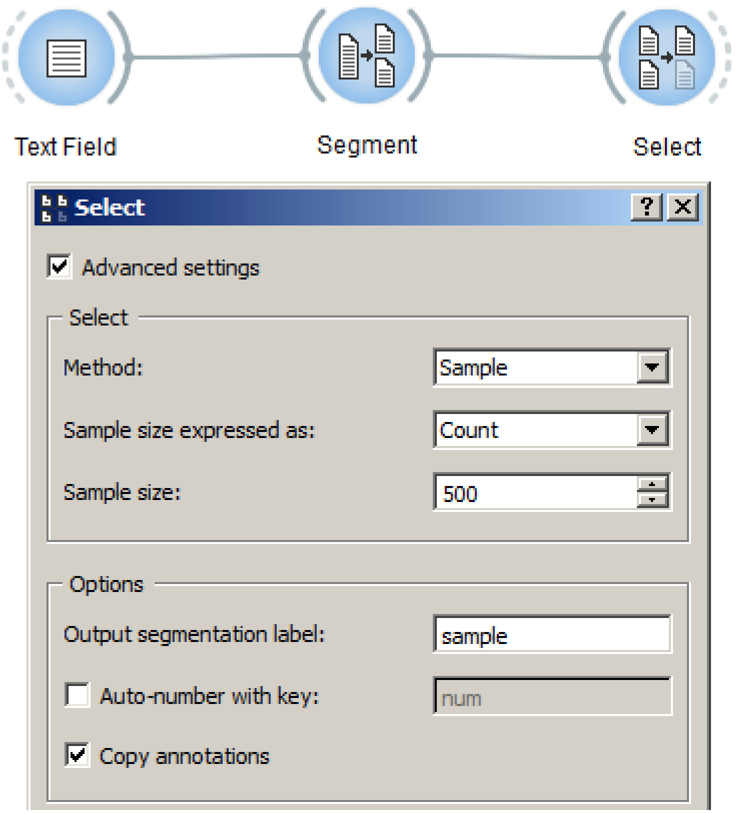
Some text has been imported in Orange Textable (see *Cookbook: Text input*) and in all likelihood it has been segmented in smaller units (see *Cookbook: Segment text in smaller units*).

Ingredients

Widget	*Select*
Icon	📄
Quantity	1

Procedure

1. Create an instance of *Select* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that emits the segmentation to be sampled (e.g. an instance of *Segment*) to the *Select* instance's input connection (lefthand side).
3. Open the *Select* instance's interface by double-clicking on its icon on the canvas.
4. Tick the **Advanced settings** checkbox.
5. In the **Select** section, choose the **Method: Sample**.
6. Under **Sample size expressed as**, choose whether you want to express sample size in terms of **Count** (i.e. number of tokens) or of **Proportion** (i.e. percentage of tokens).
7. In the **Sample size** control, choose the number of segments that will be randomly sampled (respectively, choose the percentage of segments in the **Sampling rate (%)** control).



Select

☑ Advanced settings

Select

Method: Sample

Sample size expressed as: Count

Sample size: 500

Options

Output segmentation label: sample

□ Auto-number with key: num

☑ Copy annotations

Fig. 61: Figure 1: Create a random selection or sample of segments with an instance of *Select*

8. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
9. A segmentation containing the sampled segments is then available on the *Select* instance's output connections; to display or export it, see *Cookbook: Text output*.

Comment

- The *Select* widget emits on a second output connection (not selected by default) a segmentation containing the segments that were *not* selected.

See also

- *Reference: Select widget*
- *Cookbook: Text input*
- *Cookbook: Segment text in smaller units*
- *Cookbook: Text output*

Exclude segments based on a stoplist

Goal

Filter out segments based on a stoplist.

Prerequisites

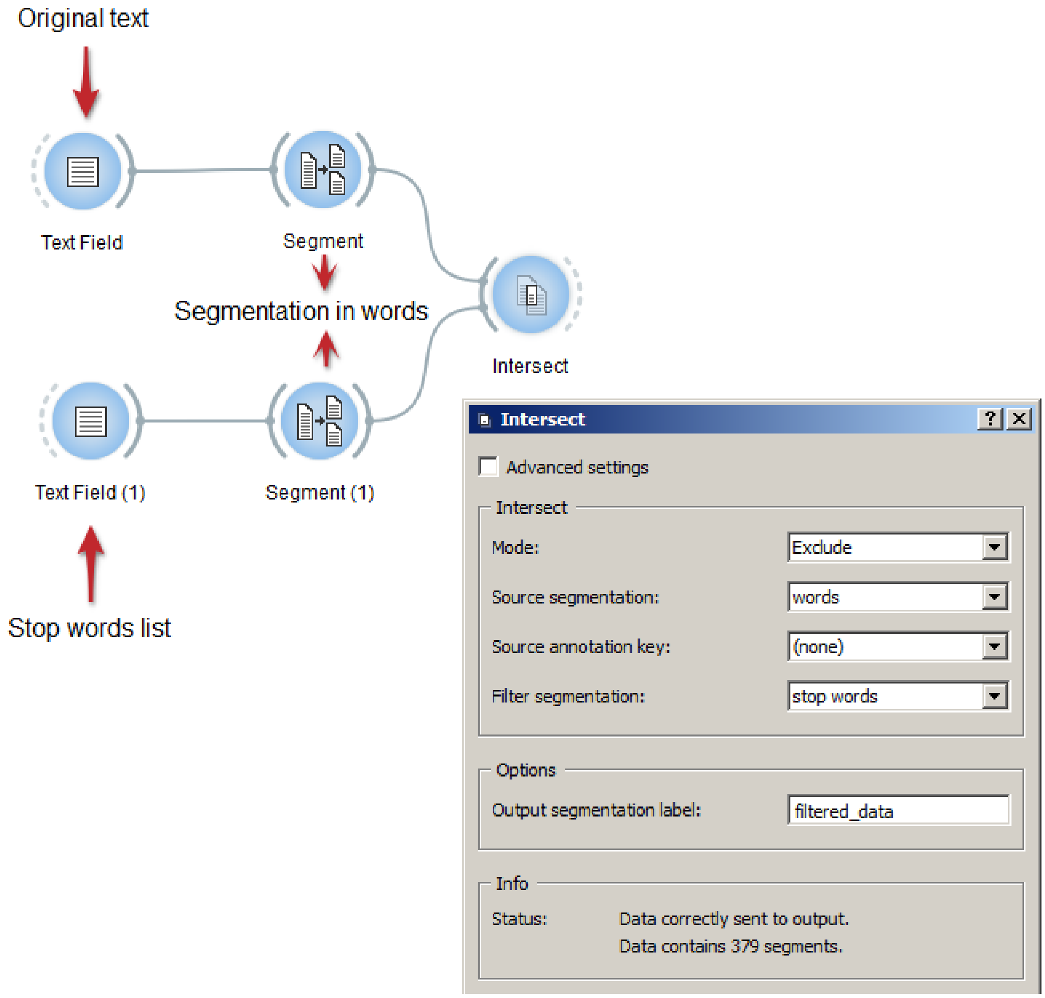
Some text has been imported in Orange Textable (see *Cookbook: Text input*) and it has been segmented into words (see *Cookbook: Segment text in smaller units*).

Ingredients

Widget	*Text Field*	*Segment*	*Intersect*
Icon			figures/intersect_36.png
Quantity	1	1	1

Procedure

1. Create an instance of *Text Field* on the canvas and paste into it the stoplist you want to use.
2. Follow the indications given in *Cookbook: Segment text in smaller units* in order to segment the stoplist into words; in what follows, it is assumed that the label of the resulting segmentation is *stop words*.
3. Create an instance of *Intersect* on the canvas.



Intersect

□ Advanced settings

Intersect

Mode: Exclude

Source segmentation: words

Source annotation key: (none)

Filter segmentation: stop words

Options

Output segmentation label: filtered_data

Info

Status: Data correctly sent to output.
Data contains 379 segments.

Fig. 62: Figure 1: Exclude segments based on a stoplist with instances of *Text Field*, *Segment* and *Intersect*

4. Drag and drop from the output connection (righthand side) of the widget instance that emits the segmentation to be filtered (here the top instance of *Segment*) to the *Intersect* instance's input connection (lefthand side).
5. Likewise, connect the *Segment* instance that emits the *stop words* segmentation to the *Intersect* instance.
6. Open the *Intersect* instance's interface by double-clicking on its icon on the canvas.
7. In the **Intersect** section, choose **Mode: Exclude**.
8. In the **Source segmentation** field, choose the label of the word segmentation to be filtered (here: *words*); in the **Filter segmentation** field, choose the label the segmentation containing the stopwords (here: *stop words*).
9. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
10. A segmentation containing the filtered segmentation is then available on the *Intersect* instance's output connections; to display or export it, see *Cookbook: Text output*.

Comment

- Stopword lists for various languages can be found here.

See also

- *Getting started: Using a segmentation to filter another*
- *Reference: Intersect widget*
- *Cookbook: Text input*
- *Cookbook: Segment text in smaller units*
- *Cookbook: Text output*

Convert XML tags into Orange Textable annotations

Goal

Convert XML markup into Orange Textable data structures such as segments and their annotations.

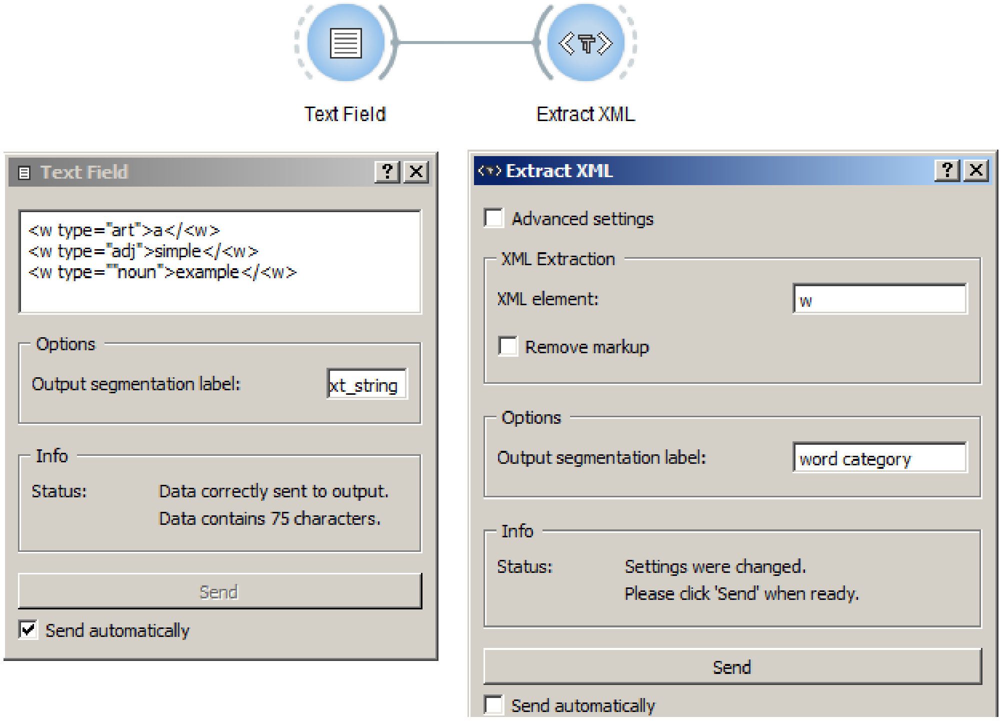
Prerequisites

Some text containing XML markup has been imported in Orange Textable (see *Cookbook: Text input*) and possibly further processed (see *Cookbook: Segmentation manipulation*).

Ingredients

Widget	*Extract XML*
Icon	⟪T⟫
Quantity	1

Procedure



Text Field

```
<w type="art">a</w>
<w type="adj">simple</w>
<w type=""noun">example</w>
```

Options

Output segmentation label: xt_string

Info

Status: Data correctly sent to output.
Data contains 75 characters.

Send

☑ Send automatically

Extract XML

☐ Advanced settings

XML Extraction

XML element: w

☐ Remove markup

Options

Output segmentation label: word category

Info

Status: Settings were changed.
Please click 'Send' when ready.

Send

☐ Send automatically

Fig. 63: Figure 1: Convert XML tags into Orange Textable annotations with an instance of *Extract XML*

1. Create an instance of *Extract XML* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that emits the data containing XML markup (e.g. *Text Field*) to the *Extract XML* widget instance's input connection (lefthand side).
3. Open the *Extract XML* instance's interface by double-clicking on its icon on the canvas.
4. In the **XML Extraction** section, insert the desired **XML element** (here `w`).
5. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
6. A segmentation containing a segment for each occurrence of the specified tag is then available on the *Segment* instance's output connections; to display or export it, see *Cookbook: Text output*.

Comment

- The XML tags that have been retrieved are actually discarded from the resulting segmentation: only their content is included in the output.
- The attributes of the XML tags are automatically converted to annotations associated with the created segments.
- Note that it is only possible to extract instances of a single XML element type at a time (here `w`).
- However, it is possible to chain several *Extract XML* instances in order to successively extract instances of different XML elements. For example, a first instance to extract `div` type elements, a second to extract `w` type elements, and so on. In this case, it is important to make sure that the **Remove markup** option is *not* selected.

See also

- *Getting started: Converting XML markup to annotations*
- *Reference: Extract XML widget*
- *Cookbook: Text input*
- *Cookbook: Segmentation manipulation*
- *Cookbook: Text output*

1.4.5 Text analysis

Count unit frequency

Goal

Count the frequency of each segment type that appears in a segmentation.

Prerequisites

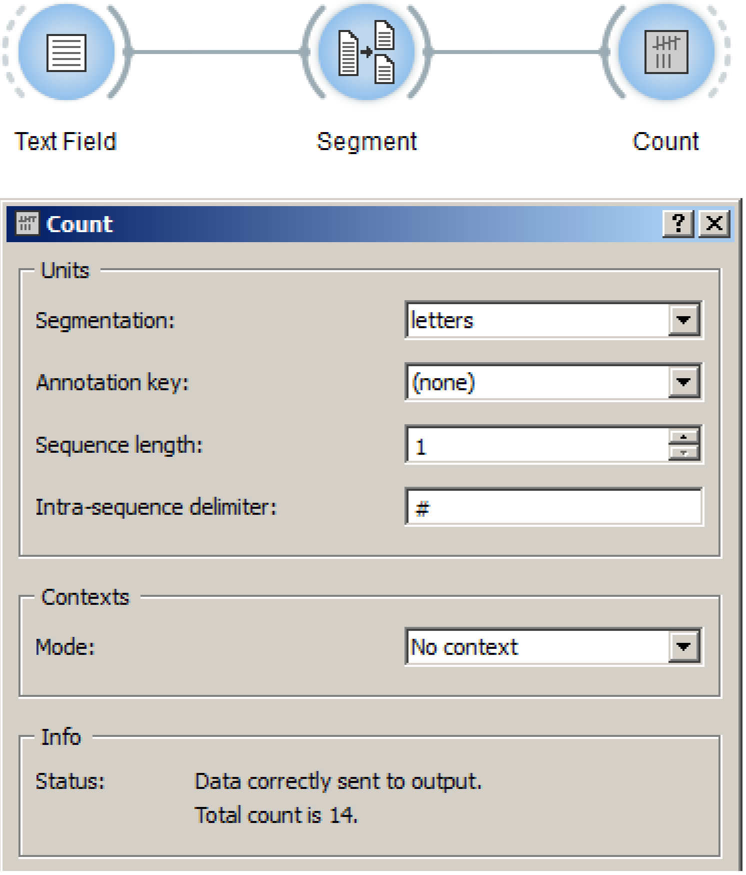
Some text has been imported in Orange Textable (see *Cookbook: Text input*) and it has been segmented in smaller units (see *Cookbook: Segment text in smaller units*).

Ingredients

Widget	*Count*
Icon	𝍸 |||
Quantity	1

Procedure

1. Create an instance of *Count* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that emits the segments that will be counted (e.g. *Segment*) to the *Count* widget instance's input connection (lefthand side).
3. Open the *Count* instance's interface by double-clicking on its icon on the canvas.
4. In the **Units** section, select the segmentation containing units to be counted in the **Segmentation** drop-down menu (here: *letters*).
5. Click the **Compute** button (or make sure the **Compute automatically** checkbox is selected).
6. A table showing the results is then available at the output connection of the *Count* instance; to display or export it, see *Cookbook: Table output*.



Count

Units

Segmentation: letters

Annotation key: (none)

Sequence length: 1

Intra-sequence delimiter: #

Contexts

Mode: No context

Info

Status: Data correctly sent to output.
Total count is 14.

Fig. 64: Figure 1: Count unit frequency globally with an instance of *Count*.

Comment

- The total number of segments in your segmentation appears in the **Info** section (here: 14).
- It is also possible to define units as segment pairs (*bigrams*), triples (*trigrams*), and so on, by increasing the **Sequence length** parameter in the **Units** section.
- If **Sequence length** is set to a value greater than 1, the string appearing in the **Intra-sequence delimiter** field will be inserted between the elements composing each *n*-gram in the column headers, which can enhance their readability. The default is # but you can change it by inserting the delimiter of your choice.

See also

- *Getting started: Counting segment types*
- *Reference: Count widget*
- *Cookbook: Text input*
- *Cookbook: Segment text in smaller units*
- *Cookbook: Table output*

Count occurrences of smaller units in larger segments

Goal

Count the occurrences of smaller units (for instance letters) in larger segments (for instance words), and report the results by means of a two-dimensional contingency table (e.g. with words in rows and letters in columns).

Prerequisites

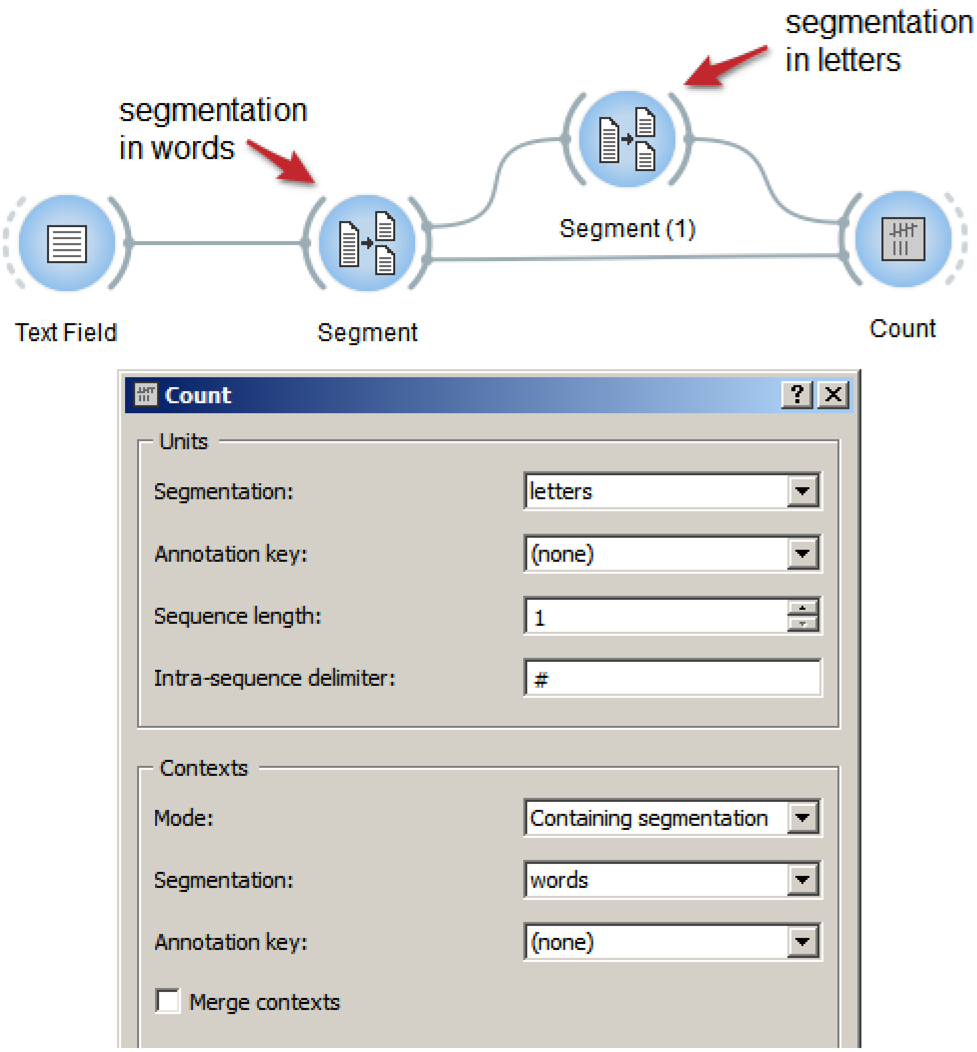
Some text has been imported in Orange Textable (see *Cookbook: Text input*) and it has been segmented in at least two hierarchical levels, e.g. words and letters (see *Cookbook: Segment text in smaller units*).

Ingredients

Widget	*Count*
Icon	
Quantity	1

Procedure

1. Create an instance of *Count* on the canvas.
2. Drag and drop from the output connection (righthand side) of both widget instances that have been used to segment the text (here the two instances of *Segment*) to the *Count* widget instance's input connection (lefthand side), thus forming a triangle.



Count

Units

Segmentation: letters

Annotation key: (none)

Sequence length: 1

Intra-sequence delimiter: #

Contexts

Mode: Containing segmentation

Segmentation: words

Annotation key: (none)

☐ Merge contexts

Fig. 65: Figure 1: Count occurrences of smaller units in larger segments with an instance of *Count*

3. Open the *Count* instance's interface by double-clicking on its icon on the canvas.
4. In the **Units** section, select the segmentation into smaller units (here: *letters*).
5. In the **Context** section, choose **Mode: Containing segmentation**.
6. In the **Segmentation** field, select the context segmentation, i.e. the segmentation into larger segments (here *words*).
7. Click the **Compute** button (or make sure the **Compute automatically** checkbox is selected).
8. A table showing the results is then available at the output connection of the *Count* instance; to display or export it, see *Cookbook: Table output*.

Comment

- The total number of segments in your segmentation appears in the **Info** section (here: 14).
- It is also possible to define units as segment pairs (*bigrams*), triples (*trigrams*), and so on, by increasing the **Sequence length** parameter in the **Units** section.
- If **Sequence length** is set to a value greater than 1, the string appearing in the **Intra-sequence delimiter** field will be inserted between the elements composing each *n*-gram in the column headers, which can enhance their readability. The default is # but you can change it by inserting the delimiter of your choice.

See also

- *Getting started: Counting in specific contexts*
- *Reference: Count widget*
- *Cookbook: Text input*
- *Cookbook: Segment text in smaller units*
- *Cookbook: Table output*

Count transition frequency between adjacent units

Goal

Count the frequency of transitions between adjacent segment types in a text.

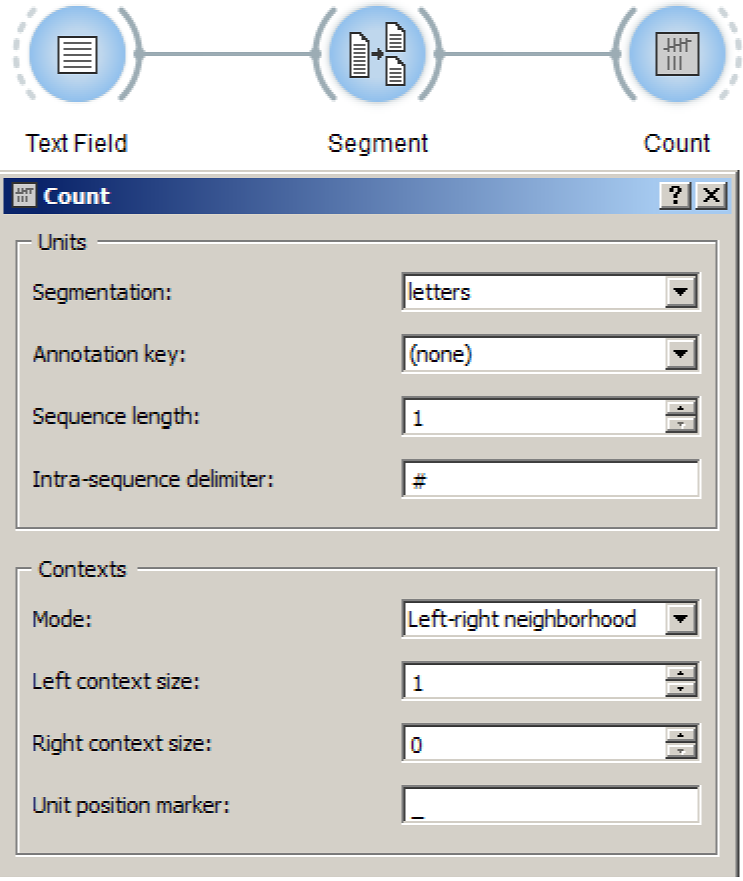
Prerequisites

Some text has been imported in Orange Textable (see *Cookbook: Text input*) and it has been segmented in smaller units (see *Cookbook: Segment text in smaller units*).

Ingredients

Widget	*Count*
Icon	
Quantity	1

Procedure



Units

Segmentation: letters

Annotation key: (none)

Sequence length: 1

Intra-sequence delimiter: #

Contexts

Mode: Left-right neighborhood

Left context size: 1

Right context size: 0

Unit position marker: _

Fig. 66: Figure 1: Count transition frequency with an instance of *Count*

1. Create an instance of *Count* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that has been used to segment the text (e.g. *Segment*) to the *Count* widget instance's input connection (lefthand side).
3. Open the *Count* instance's interface by double-clicking on its icon on the canvas.
4. In the **Units** section, select the segmentation in which transitions between units will be counted.
5. In the **Context** section, choose **Mode: Left-right neighborhood**.
6. Select **Left context size:** *1* and **Right context size:** *0*.

7. Click the **Compute** button (or make sure the **Compute automatically** checkbox is selected).
8. A table showing the results is then available at the output connection of the *Count* instance; to display or export it, see *Cookbook: Table output*.

Comment

- It is also possible to define units as segment pairs (*bigrams*), triples (*trigrams*), and so on, by increasing the **Sequence length** parameter in the **Units** section.
- If **Sequence length** is set to a value greater than 1, the string appearing in the **Intra-sequence delimiter** field will be inserted between the elements composing each *n*-gram in the column headers, which can enhance their readability. The default is # but you can change it by inserting the delimiter of your choice.
- Furthermore, it is possible to count the apparition of units in more complex contexts than simply the previous unit, such as: the *n* previous units (**Left context size**); the *n* following units (**Right context size**); or any combination of both.
- The **Unit position marker** is a string that indicates the separation between left and right contexts sides. The default is _ but you can change it by inserting the marker of your choice.

See also

- *Reference: Count widget*
- *Cookbook: Text input*
- *Cookbook: Segment text in smaller units*
- *Cookbook: Table output*

Examine the evolution of unit frequency along the text

Goal

Examine how the frequency of segment types evolves from the beginning to the end of a segmentation.

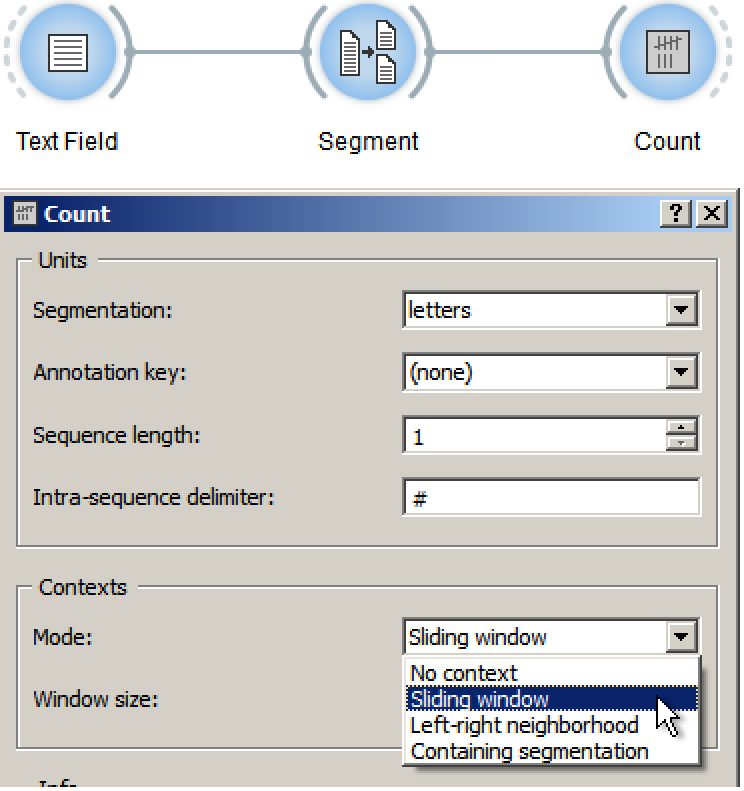
Prerequisites

Some text has been imported in Orange Textable (see *Cookbook: Text input*) and it has been segmented in smaller units (see *Cookbook: Segment text in smaller units*).

Ingredients

Widget	*Count*
Icon	𝍸
Quantity	1

Procedure



Count

Units

Segmentation: letters

Annotation key: (none)

Sequence length: 1

Intra-sequence delimiter: #

Contexts

Mode: Sliding window

Window size:

Fig. 67: Figure 1: Examine the evolution of unit frequency with an instance of *Count*

1. Create an instance of *Count* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that has been used to segment the text (e.g. *Segment*) to the *Count* widget instance's input connection (lefthand side).
3. Open the *Count* instance's interface by double-clicking on its icon on the canvas.
4. In the **Units** section, select the segmentation whose units will be counted.
5. In the **Context** section, choose **Mode: Sliding window**.
6. Set the **Window size** parameter to the desired value; with the minimum value of 1, frequency will be counted separately at every successive position in the segmentation, whereas a larger value $n > 1$ will have the effect that frequency will be counted in larger and partially overlapping spans (segments 1 to n, then 2 to $n + 1$, and so on), resulting in a smoother curve.
7. Click the **Compute** button (or make sure the **Compute automatically** checkbox is selected).
8. A table showing the results is then available at the output connection of the *Count* instance; to display or export it, see *Cookbook: Table output*.

Comment

- It is also possible to define units as segment pairs (*bigrams*), triples (*trigrams*), and so on, by increasing the **Sequence length** parameter in the **Units** section.

- If **Sequence length** is set to a value greater than 1, the string appearing in the **Intra-sequence delimiter** field will be inserted between the elements composing each *n*-gram in the column headers, which can enhance their readability. The default is `#` but you can change it by inserting the delimiter of your choice.

See also

- *Reference: Count widget*
- *Cookbook: Segment text*
- *Cookbook: Display table*

Build a concordance

Goal

Build a concordance to examine the context of occurrence of a given string.

Prerequisites

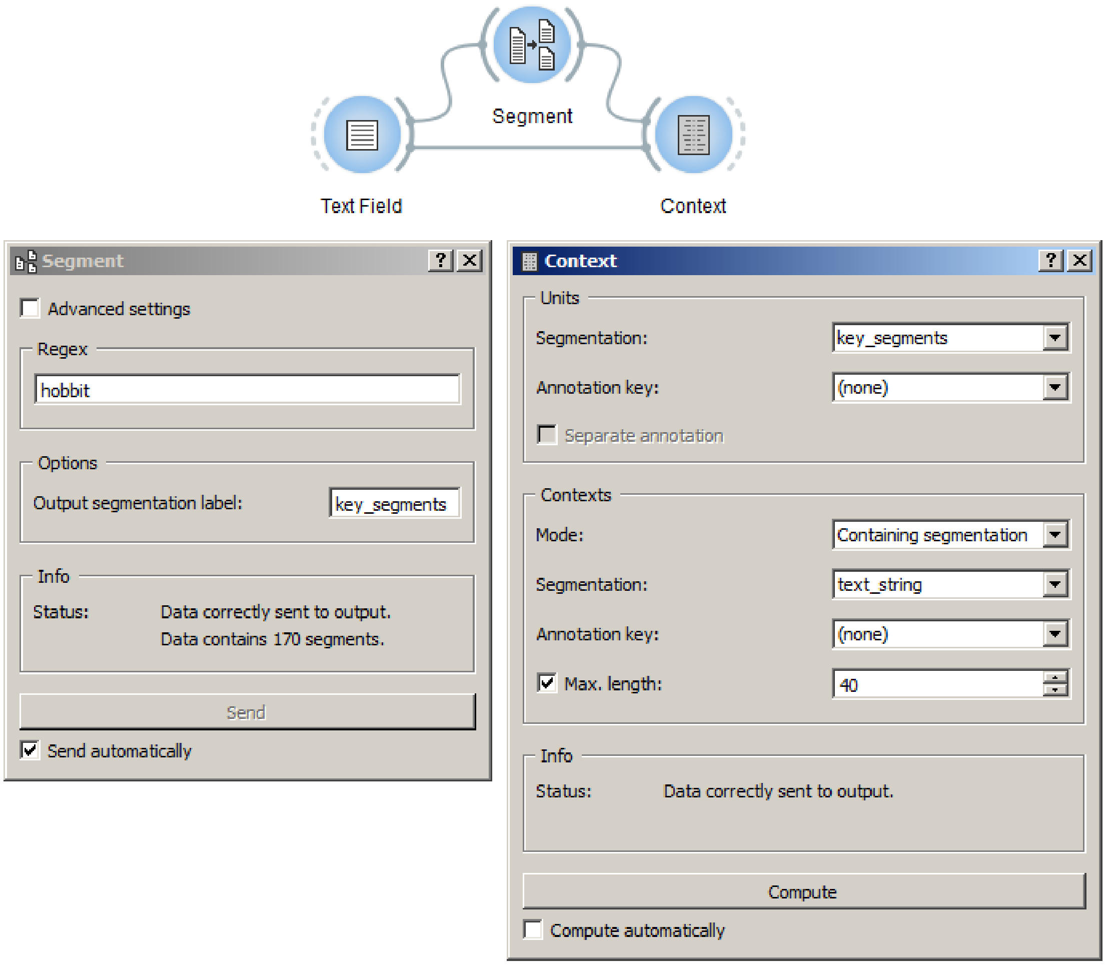
Some text has been imported in Orange Textable (see *Cookbook: Text input*) and possibly further processed (see *Cookbook: Segmentation manipulation*).

Ingredients

Widget	*Segment*	*Context*
Icon		
Quantity	1	1

Procedure

1. Create an instance of *Segment* and an instance of *Context* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that emits the segmentation in which occurrences of the query string will be retrieved (e.g. *Text Field*) to the *Segment* widget instance's input connection (lefthand side).
3. Also connect both the *Text Field* instance and the *Segment* instance to the *Context* instance (thus forming a triangle).
4. Open the *Segment* instance's interface by double-clicking on its icon on the canvas and type the string whose context of occurrence will be examined in the **Regex** field (here: `hobbit`); assign it a recognizable **Output segmentation label**, such as *key_segments* for instance.
5. Click the **Send** button (or make sure the **Send automatically** checkbox is selected).
6. Open the *Context* instance's interface by double-clicking on its icon on the canvas.
7. In the **Units** section, select the segmentation that contains the occurrences of the query string (here: *key_segments*) using the **Segmentation** drop-down menu.



Segment

□ Advanced settings

Regex: hobbit

Options

Output segmentation label: key_segments

Info

Status: Data correctly sent to output.
Data contains 170 segments.

Send

☑ Send automatically

Context

Units

Segmentation: key_segments

Annotation key: (none)

□ Separate annotation

Contexts

Mode: Containing segmentation

Segmentation: text_string

Annotation key: (none)

☑ Max. length: 40

Info

Status: Data correctly sent to output.

Compute

□ Compute automatically

Fig. 68: Figure 1: Widgets used build a concordance and their interfaces

8. In the **Contexts** section, choose **Mode: Containing segmentation** and select the segmentation that contains the original text (here: *text_string*, as emitted by the *Text Field* instance) using the **Segmentation** drop-down menu.
9. Tick the **Max. length** checkbox and set the maximum number of characters that should be displayed on either side of each occurrence of the query string.
10. Click the **Compute** button (or make sure the **Compute automatically** checkbox is selected).
11. A table showing the results is then available at the output connection of the *Count* instance; to display or export it, see *Cookbook: Table output*.

Comment

- In the **Regex** field of the *Segment* widget you can use all the syntax of Python's regular expression (*cf.* Python documentation); for instance, if you wish to restrict your search to entire words, you might frame the query string with word boundary anchors `\b` (in our example `\bhobbit\b`).

See also

- *Reference: Segment widget*
- *Reference: Context widget*
- *Cookbook: Text input*
- *Cookbook: Segmentation manipulation*
- *Cookbook: Table output*

1.4.6 Table output

Display table

Goal

Display an Orange Textable table.

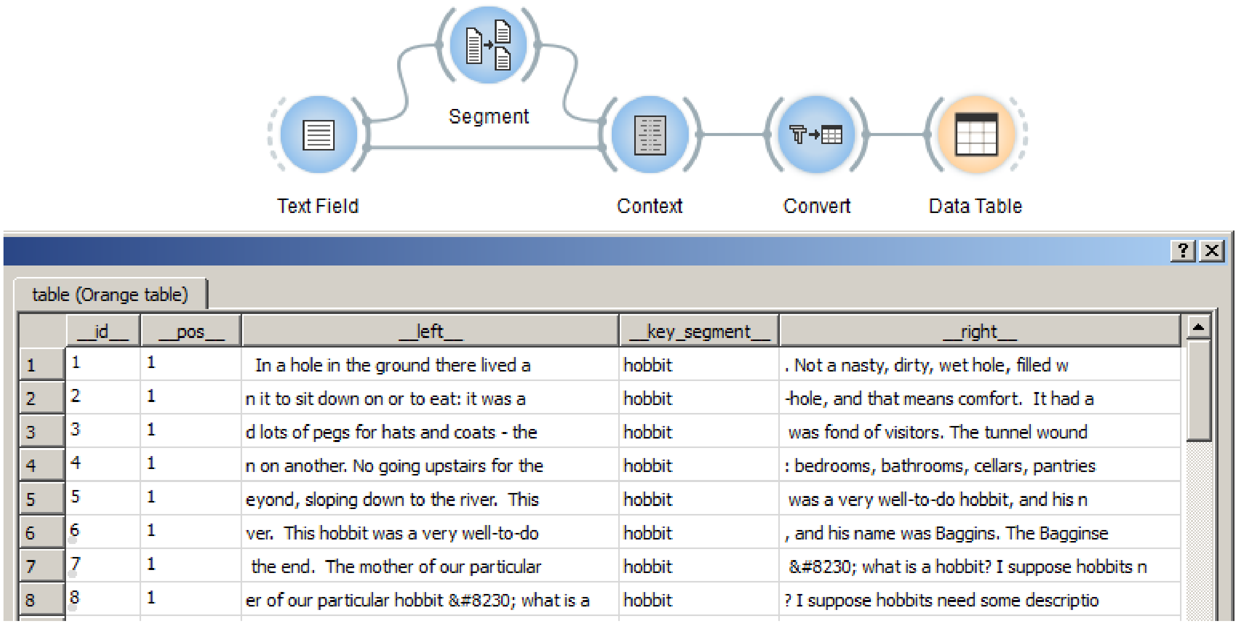
Prerequisites

Some text has been imported in Orange Textable (see *Cookbook: Text input*) and possibly further processed (see *Cookbook: Segmentation manipulation*). A table has been created by means of one of Orange Textable's *table construction widgets* (see *Cookbook: Text analysis*).

Ingredients

Widget	*Convert*	**Data Table**
Icon	T→▦	▦
Quantity	1	1

Procedure



	__id__	__pos__	__left__	__key_segment__	__right__
1	1	1	In a hole in the ground there lived a	hobbit	. Not a nasty, dirty, wet hole, filled w
2	2	1	n it to sit down on or to eat: it was a	hobbit	-hole, and that means comfort. It had a
3	3	1	d lots of pegs for hats and coats - the	hobbit	was fond of visitors. The tunnel wound
4	4	1	n on another. No going upstairs for the	hobbit	: bedrooms, bathrooms, cellars, pantries
5	5	1	eyond, sloping down to the river. This	hobbit	was a very well-to-do hobbit, and his n
6	6	1	ver. This hobbit was a very well-to-do	hobbit	, and his name was Baggins. The Bagginse
7	7	1	the end. The mother of our particular	hobbit	&#8230; what is a hobbit? I suppose hobbits n
8	8	1	er of our particular hobbit &#8230; what is a	hobbit	? I suppose hobbits need some descriptio

Fig. 69: Figure 1: Display an Orange Textable table with instances of *Convert* and **Data Table**.

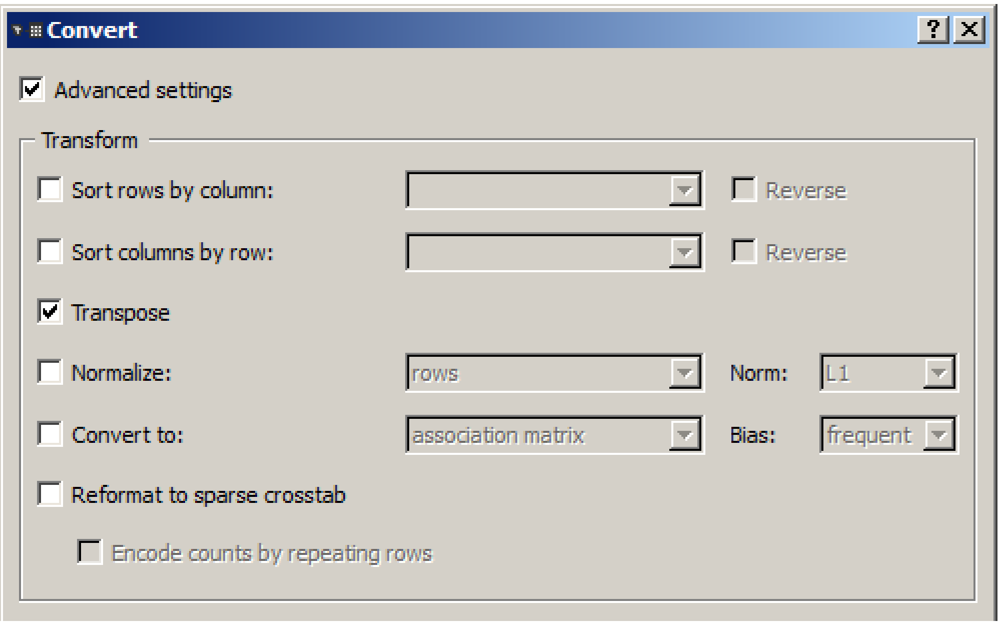
1. Create an instance of *Convert* and **Data Table** on the canvas (the latter is found in the **Data** tab of Orange Canvas).
2. Drag and drop from the output connection (righthand side) of the widget instance that has been used to build a table (e.g. *Context*) to the *Convert* widget instance's input connection (lefthand side).
3. Connect the *Convert* instance to the **Data Table** instance.
4. Open the **Data Table** instance's interface by double-clicking on its icon on the canvas to display the table.

Comment

- If the table is a frequency table, you may want to change its default orientation of the table to make it easier to read. To that effect, open the *Convert* instance's interface, tick the **Advanced settings** checkbox, and in the **Transform** section, tick the **transpose** checkbox.

See also

- Getting started: Converting between table formats
- *Reference: Convert widget*
- *Reference: Table construction widgets*
- *Cookbook: Text input*
- *Cookbook: Segmentation manipulation*
- *Cookbook: Text analysis*



Convert

☑ Advanced settings

Transform

☐ Sort rows by column: ☐ Reverse

☐ Sort columns by row: ☐ Reverse

☑ Transpose

☐ Normalize: rows Norm: L1

☐ Convert to: association matrix Bias: frequent

☐ Reformat to sparse crosstab

☐ Encode counts by repeating rows

Fig. 70: Figure 2: Change the orientation of an Orange Textable frequency table using an instance of *Convert*.

Export table

Goal

Export an Orange Textable table in a text file in order to later import it in another program (e.g. spreadsheet software).

Prerequisites

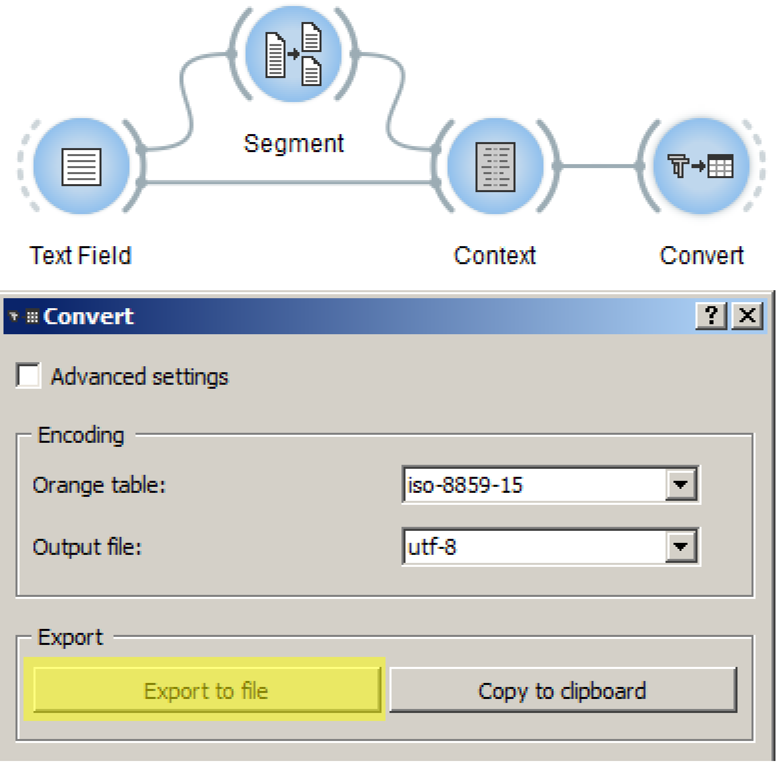
Some text has been imported in Orange Textable (see *Cookbook: Text input*) and possibly further processed (see *Cookbook: Segmentation manipulation*). A table has been created by means of one of Orange Textable's *table construction widgets* (see *Cookbook: Text analysis*).

Ingredients

Widget	*Convert*
Icon	¶→▦
Quantity	1

Procedure

1. Create an instance of *Convert* on the canvas.
2. Drag and drop from the output connection (righthand side) of the widget instance that has been used to build a table (e.g. *Context*) to the *Convert* widget instance's input connection (lefthand side).



☐ Advanced settings

Encoding

Orange table: iso-8859-15

Output file: utf-8

Export

Fig. 71: Figure 1: Export table with an instance of *Convert*

3. Open the *Convert* instance's interface by double-clicking on its icon on the canvas.
4. Select the desired encoding for the exported data (e.g. utf8).
5. Click the **Export to file** button to open the file selection dialog.
6. Select the location you want to export your file to and close the file selection dialog by clicking on **Ok**.

Comment

- If you rather want to *copy* the text content in order to later paste it in another program, click on **Copy to clipboard**; note that in this case, the encoding is by default utf8 and cannot be changed.
- The default column delimiter is `\t` but this can be modified to either comma (`,`) or semi-colon (`;`) by ticking the **Advanced settings** checkbox in the *Convert* instance's interface, then selecting the desired delimiter in the **Column delimiter** drop-down menu (**Export** section).

See also

- *Reference: Convert widget*
- *Reference: Table construction widgets*
- *Cookbook: Text input*
- *Cookbook: Segmentation manipulation*
- *Cookbook: Text analysis*

1.5 Case studies

This section aims to provide a repository of use cases illustrating the application of Orange Textable to realistic text analysis problems. The focus here is not so much on "how to" as it is on "why". Each case study comes with a downloadable Orange Textable scheme that can be studied interactively and adapted to the specific needs of the user.

1.5.1 Term frequency comparison in Melville's Moby Dick

(This use case was designed with the help of Douglas Duhaime and the following text was slightly adapted from a description kindly contributed by him.)

This case study is adapted from Matthew Jocker's excellent work *Text Analysis with R for Students of Literature* (46). The goal here is to visualize the frequency of the terms "Ahab" and "whale(s)" within Herman Melville's masterpiece *Moby Dick*. The workflow reproduced on *figure 1* below retrieves the text from Project Gutenberg, splits the work into its constitutive chapters, and measures the degree to which each of the target terms appears in each chapter.[1]

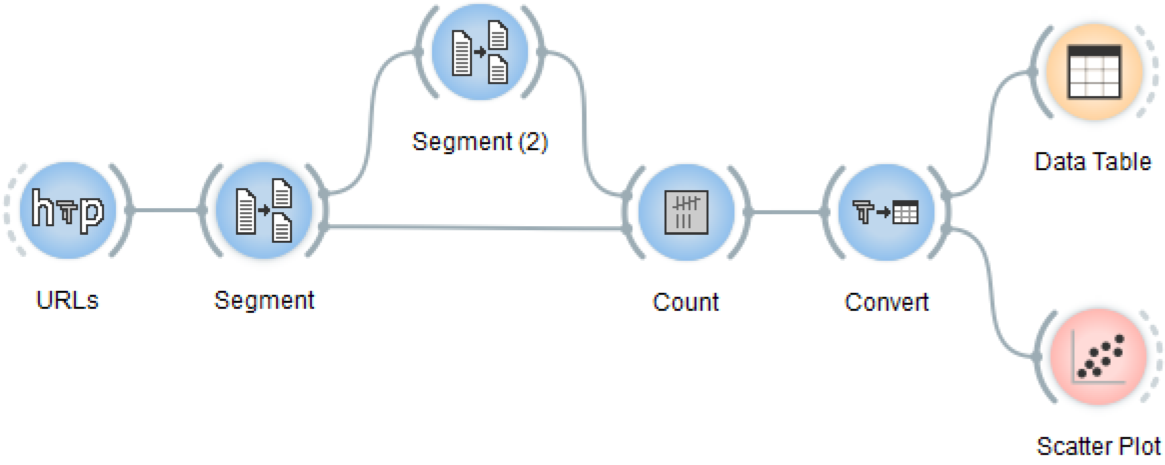


Fig. 72: Figure 1: Orange Textable workflow for visualizing term frequency in *Moby Dick*.

Clicking on the **Scatter Plot** instance, one can easily see the relative frequency of the term *whale(s)* in each chapter of Melville's novel (see *figure 2* below). By toggling the **Y-axis Attribute** dropdown box, one can select *Ahab* and visualize the frequency of *Ahab* in the novel.

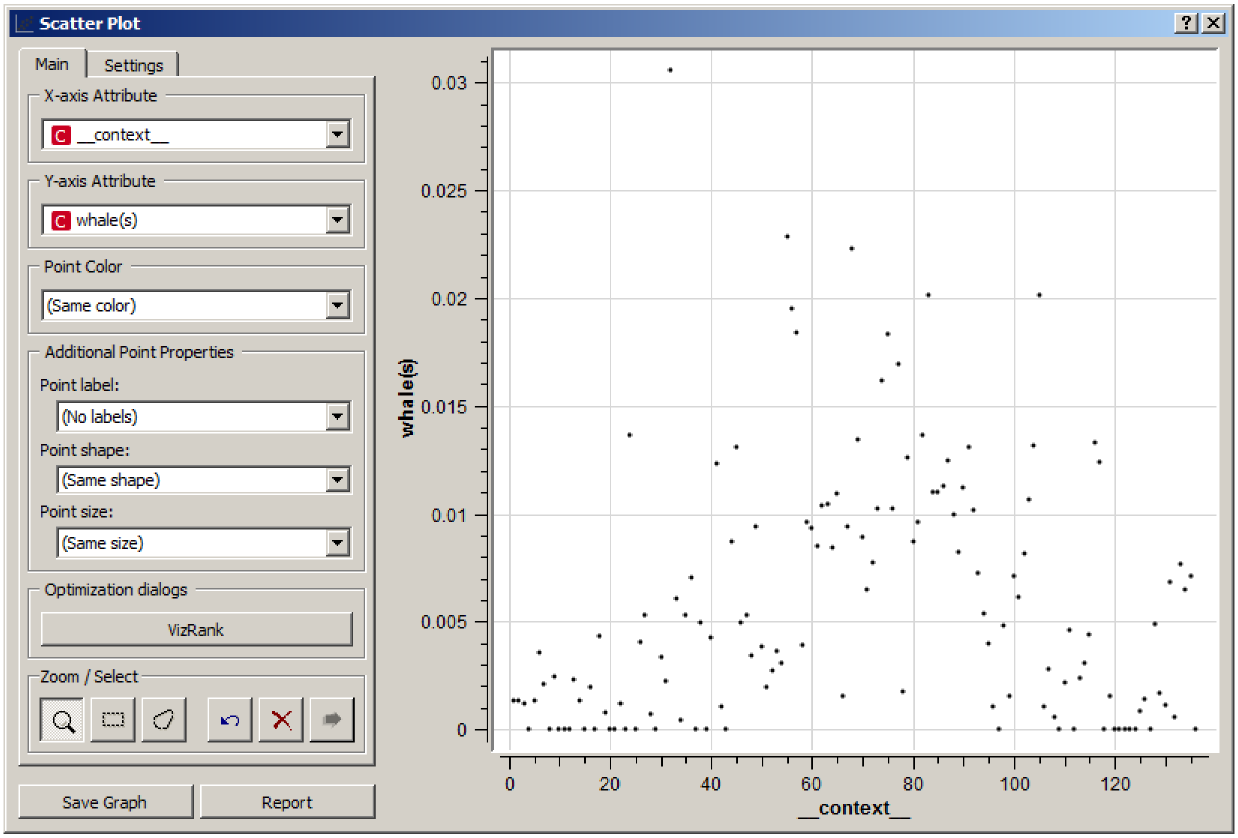
Although one might have supposed that the distribution of Captain Ahab would closely resemble that of whales within the novel, the plots above tell a different story. While Ahab is most present in early and then later chapters, whales are most present in the novel's middle chapters, creating something of an inverse relationship between the two. For the literary critic, this relationship offers new evidence with which to evaluate the strategy and structure of Melville's novel.

1.5.2 Stylometric analysis of Shakespeare's Titus Andronicus

(This use case was designed with the help of Douglas Duhaime and the following text was slightly adapted from a description kindly contributed by him.)

This is a case study in "stylometry", or the quantitative analysis of a writer's style. The data to be analyzed is William Shakespeare's play *Titus Andronicus*, which scholars have long believed William Shakespeare did not write alone.

[1] The schema can be downloaded from `here`.



X-axis Attribute

C __context__

Y-axis Attribute

C whale(s)

Point Color

(Same color)

Additional Point Properties

Point label: (No labels)

Point shape: (Same shape)

Point size: (Same size)

Since the publication of John Robertson's study *Did Shakespeare Write Titus Andronicus*, many have believed that particular scenes within the text have been written by other playwrights of the time: many believe that Act 1 Scene 1, for instance, was written by Shakespeare's contemporary George Peele.

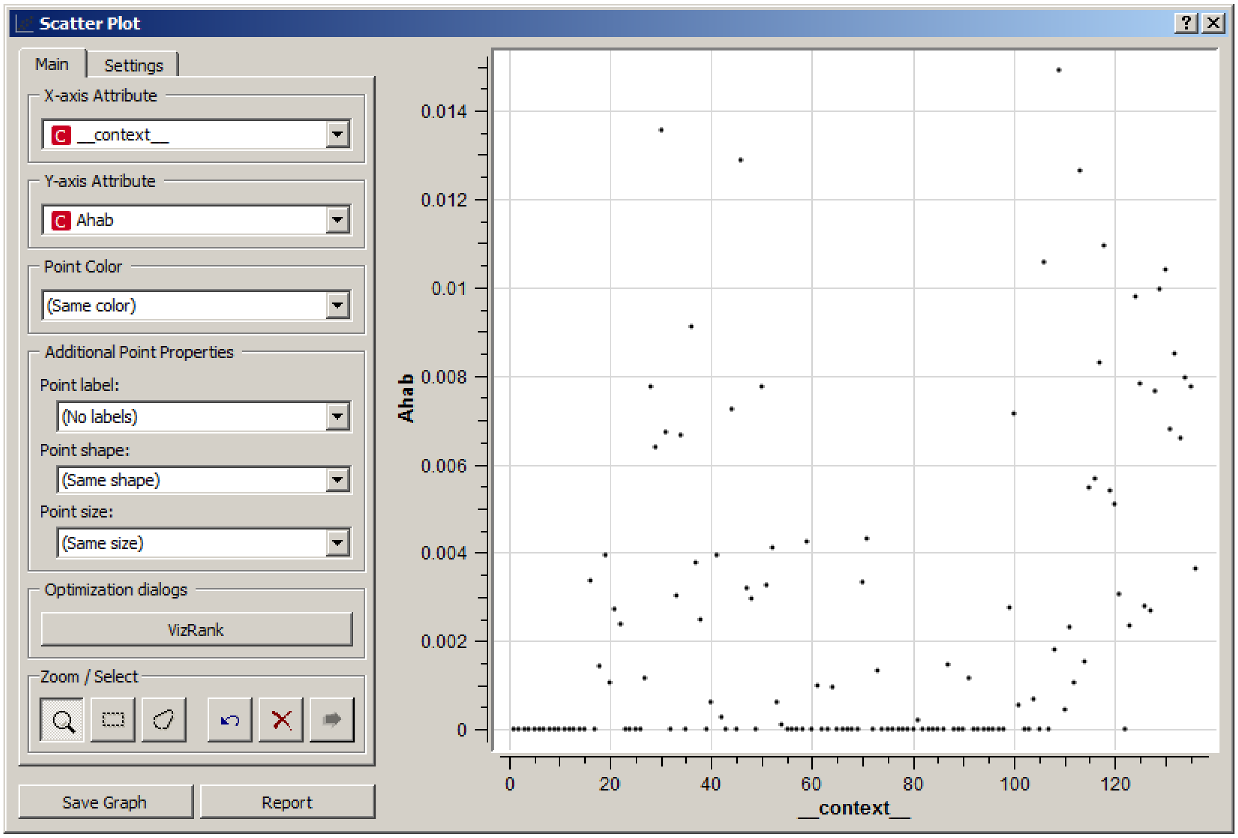
In order to test this hypothesis, the following Orange Textable workflow measures the degree to which the language in each scene within *Titus Andronicus* resembles the language within each other scene (*figure 1* below).[1] By changing the **Mode** parameter within the **Intersect** instance, one can elect to focus only on content words or stopwords, and by changing the **Distance Metrics** parameter within the **Example Distance** isntance, one can change the similarity metric for the language comparison. Finally, by clicking on the **Distance Map** icon within this workflow, one can see at a glance how distinct the vocabulary within each scene is.

Comparing the stopwords within each scene using a normalized Euclidean distance metric, one finds that Act 1 Scene 1 is indeed a significant outlier within *Titus Andronicus*. The scene remains an outlier when one performs TF-IDF normalization on the term-document matrix (within the **Convert** instance), and when one uses a normalized Manhattan distance metric. Iterating through each of the various distance metrics, and toggling between different normalization metrics, Act 1 Scene 1 remains the most consistent outlier. This adds further evidence to the argument that the scene's stylistic fingerprint departs from that of that of the rest of the play.

1.6 Reference

This part of the documentation explains the effect of every control of each Orange Textable widget. Widgets making up Orange Textable are grouped into 4 main categories based on the type of functionality they offer. A section of this part of the documentation covers each such category. The last section documents the details of JSON formats that can be used for the configuration of some of Orange Textable's widgets.

[1] The schema can be downloaded from `here`.



Scatter Plot

Main | Settings

X-axis Attribute: __context__

Y-axis Attribute: Ahab

Point Color: (Same color)

Additional Point Properties

Point label: (No labels)

Point shape: (Same shape)

Point size: (Same size)

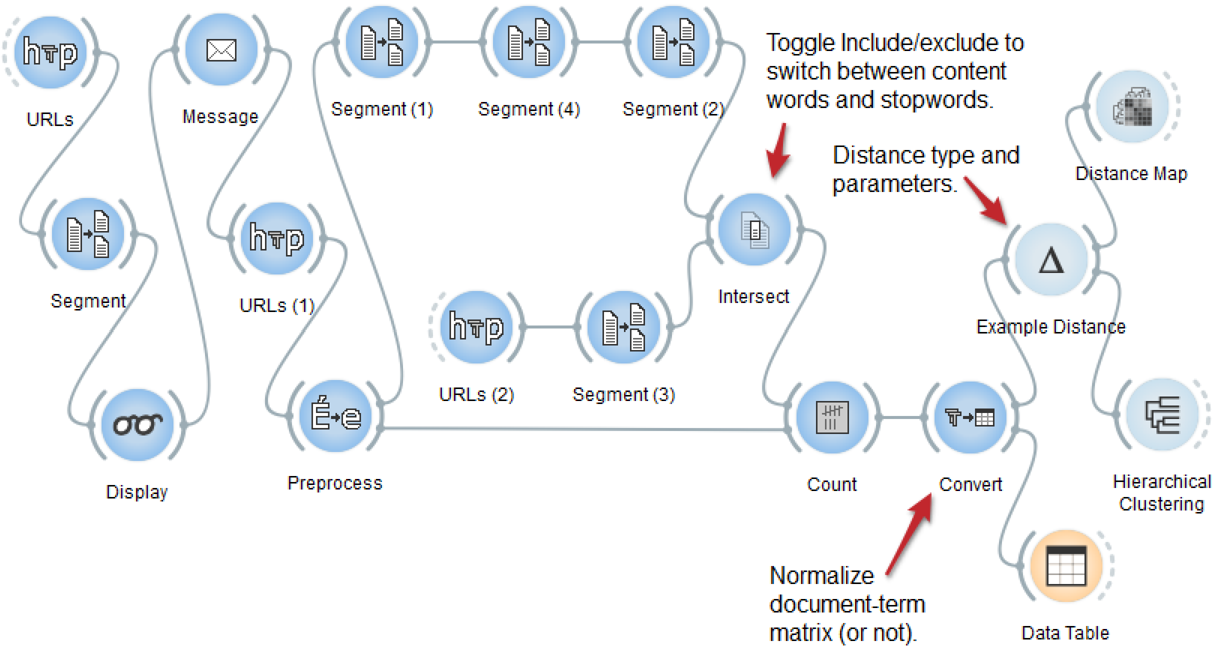
Optimization dialogs

VizRank

Zoom / Select

Save Graph | Report

Fig. 73: Figure 2: Negative correlation between the relative frequency of terms *whale(s)* (top) and *Ahab* (bottom) in Melville's novel.



Toggle Include/exclude to switch between content words and stopwords.

Distance type and parameters.

Normalize document-term matrix (or not).

Fig. 74: Figure 1: Orange Textable workflow for the stylometric analysis of *Titus Andronicus**.

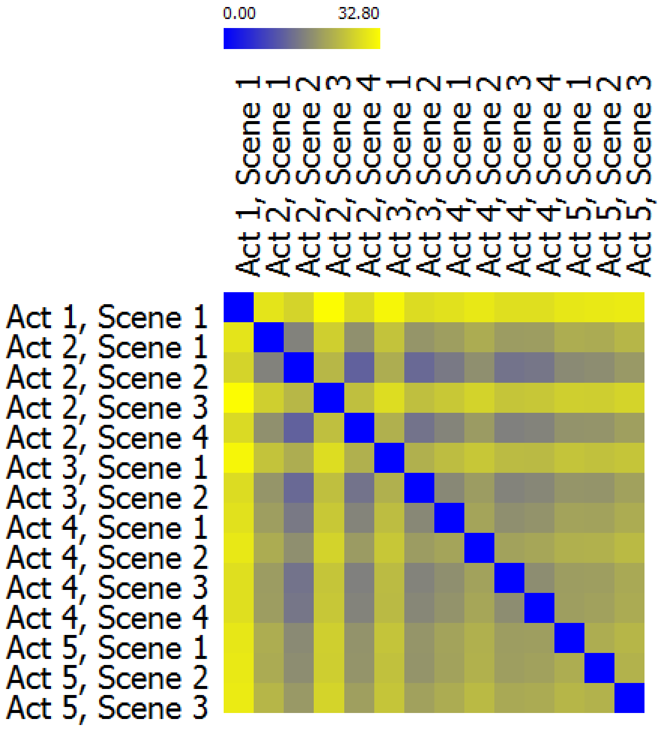


Fig. 75: Figure 2: Act 1 Scene 1 is a consistent stylistic outlier in Shakespeare's play.

1.6.1 Text import widgets

The common purpose of widgets of this category is to import text data in Orange Canvas, either from the keyboard (*Text Field*), from files (*Text Files*), or from the Internet (*URLs*). They all emit *Segmentation* data.

Text Field

Import text data from keyboard input.

Signals

Inputs:

- `Text data`

  Segmentation containing text to be edited

Outputs:

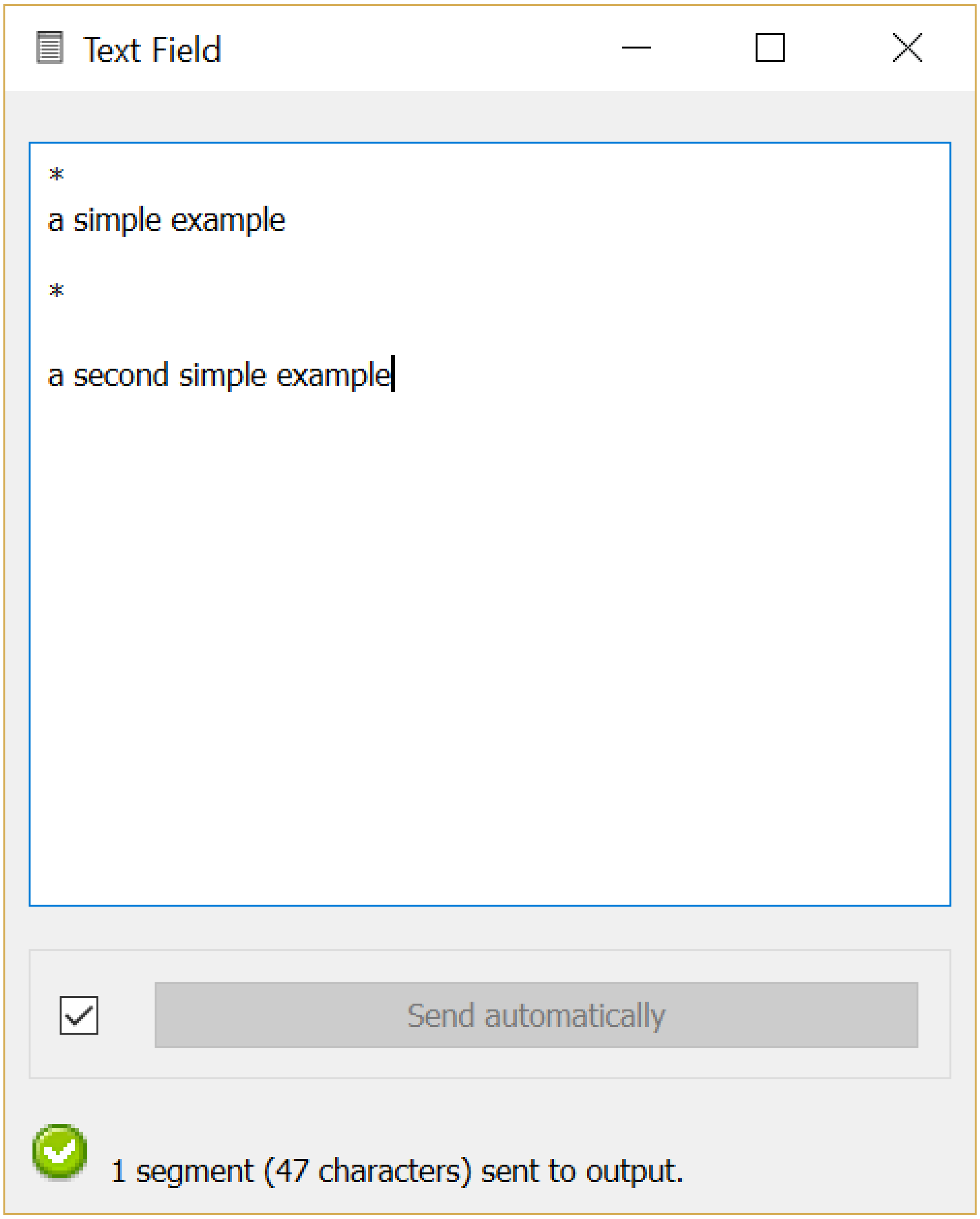
- `Text data`

  Segmentation covering the input text

Description

This widget allows the user to import keyboard collected data. It emits a segmentation containing a single unannoted segment covering the whole string. Secondarily, **Text Field** can be used to manually edit a previously imported string.

The interface of the widget is divided in two zones (see *figure 1* below). The upper part is a text field editable by the user. The standard editing functions (copy, paste, cancel, etc.) are accessible through a right-click on the field.



```
*
a simple example
*

a second simple example
```

☑ Send automatically

1 segment (47 characters) sent to output.

Fig. 76: Figure 1: Interface of the *Text field* widget.

The **Field** section allows the user to copy or manually edit texts. The text can be segmented using a character.

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface (editing of the text or label modification).

It should be noted that the text field's content is normalized in three ways:

- it is systematically converted to Unicode
- it is subjected to the canonical Unicode decomposition-recomposition technique (Unicode sequences such as `LATIN SMALL LETTER C (U+0063)` + `COMBINING CEDILLA (U+0327)` are systematically replaced by the combined equivalent, e.g. `LATIN SMALL LETTER C WITH CEDILLA (U+00C7)`)
- various forms of line endings (in particular `\r\n` and `\r`) are converted to a single form (namely `\n`)

When an instance of **Text Field** receives a segmentation on an incoming connection, the contents of all incoming segments are concatenated (without adding any delimiters) and the resulting string replaces the current textual content of the widget (if any). This allows the user to manually edit text that has been previously imported in Orange Textable. Some points are worth noting:

- This operation creates a distinct string from the one that has been previously imported: it really amounts to *copying* the original string and making the copy available for manual edition. As such, it is prone to a very specific and possibly disconcerting type of error, which can be best understood by studying the example given in the documentation of *Preprocess* (section *Caveat*), where what is said about *Preprocess* also applies to **Text Field**.
- Modifications brought from within the interface of **Text Field** to a string imported in this way will be lost if the **Text Field** instance receives a new input on its incoming connection. In particular, this will happen if the schema is saved and later re-opened. To avoid any loss of data, the safest way to operate is to remove the incoming connection as soon as it has been created and the string has been copied in the **Text Field** instance's interface; indeed, removing the incoming connection will not remove the imported string from the instance's interface, where it can then be edited without risking to overwrite it.

Messages

Information

Data correctly sent to output: 1 segment (<n> characters). This confirms that the widget has operated properly.

No data sent to output yet: text field is empty. The widget instance is not able to emit data to output because no text has been entered in the text field.

Examples

- *Getting started: Keyboard input and segmentation display*
- *Cookbook: Import text from keyboard*

See also

- Getting started: Merging segmentations together
- Getting started: Annotating by merging
- *Getting started: Converting XML markup to annotations*
- *Reference: Preprocess (section "Caveat")*

Text Files

Import data from raw text files.

Signals

Inputs:

- `Message`

  JSON Message controlling the list of imported text files

Outputs:

- `Text data`

  Segmentation covering the content of imported text files

Description

This widget is designed to import the contents of one or several text files in Orange Canvas. It outputs a segmentation containing a (potentially annotated) segment for each imported file. The imported textual content is normalized in several ways:

- it is systematically converted to Unicode (from the encoding defined by the user)
- it is subjected to the canonical Unicode decomposition-recomposition technique (Unicode sequences such as `LATIN SMALL LETTER C (U+0063)` + `COMBINING CEDILLA (U+0327)` are systematically replaced by the combined equivalent, e.g. `LATIN SMALL LETTER C WITH CEDILLA (U+00C7)`)
- it is stripped from the utf8 byte-order mark (if any)
- various forms of line endings (in particular `\r\n` and `\r`) are converted to a single form (namely `\n`)

The interface of **Text files** is available in two versions, according to whether or not the **Advanced Settings** checkbox is selected.

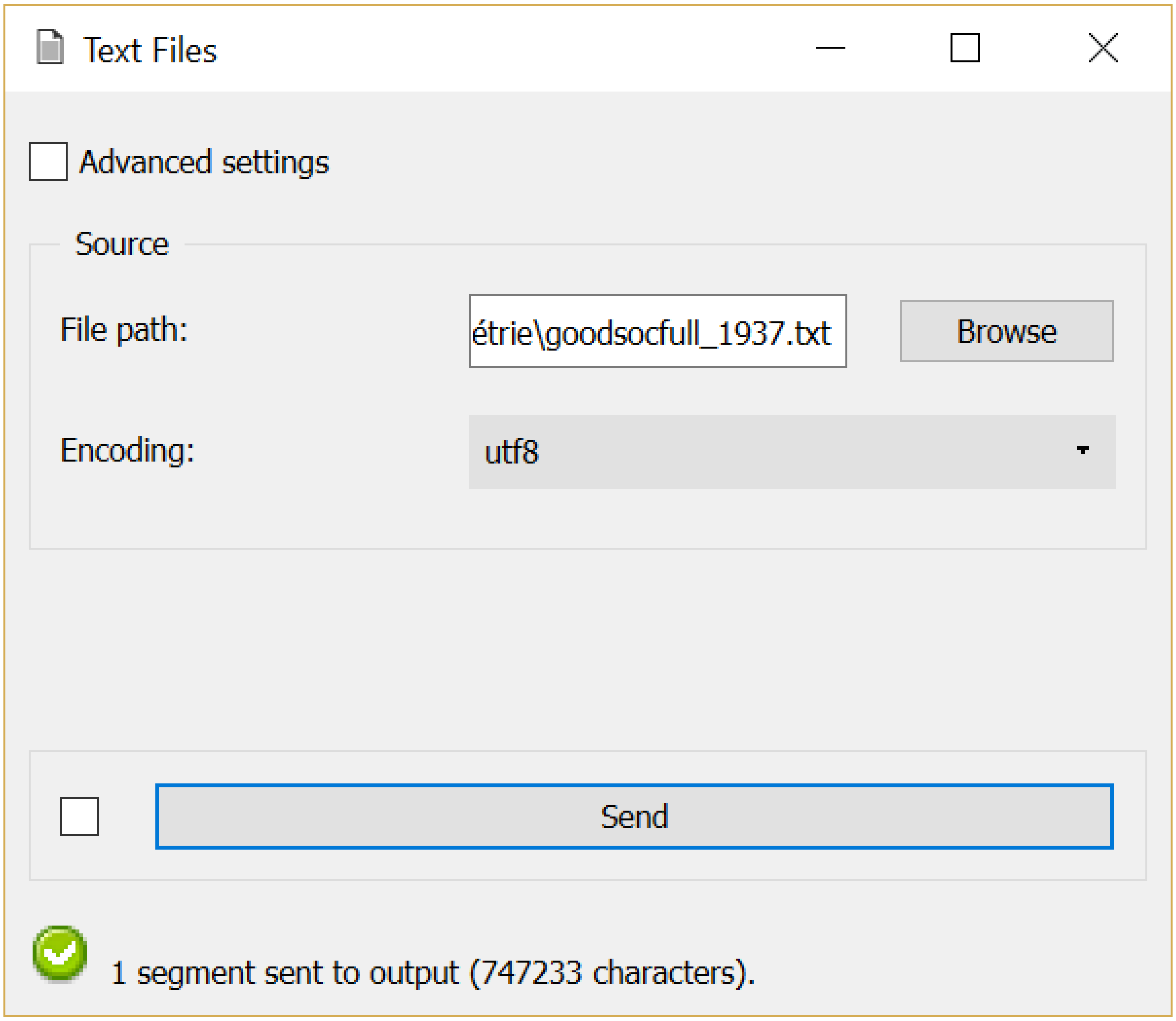
Basic interface

In its basic version (see *figure 1* below), the **Text Files** widget is limited to the import of a single file. The interface contains a **Source** section enabling the user to select the input file. The **Browse** button opens a file selection dialog; the selected file then appears in the **File path** text field (it can also be directly inputted with the keyboard). The **Encoding** drop-down menu enables the user to specify the encoding of the file.

The user can define the label of the output segmentation (**Options**) by checking the **Advanced settings**.

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface.

The text below the **Send** button indicates the number of characters in the single segment contained in the output segmentation, or the reasons why no segmentation is emitted (no input data, encoding issue, etc.).



□ Advanced settings

File path: étrie\goodsocfull_1937.txt

Encoding: utf8

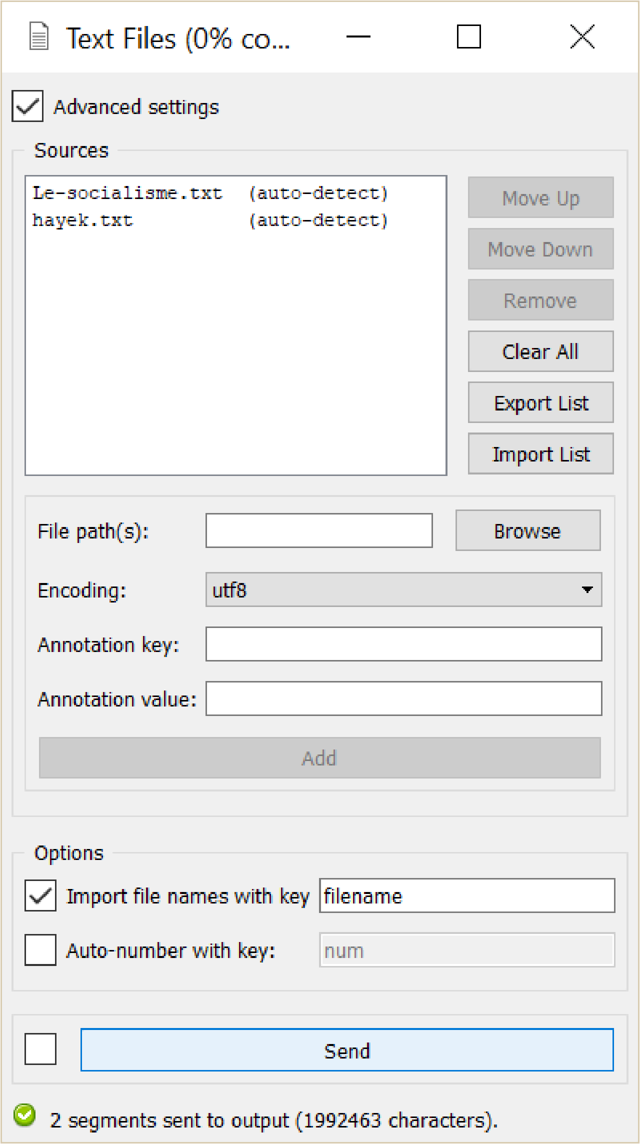
□

1 segment sent to output (747233 characters).

Fig. 77: Figure 1: **Text files** widget (basic interface).

Advanced interface

The advanced version of **Text Files** allows the user to import several files in a determined order; each file can moreover be associated to a distinct encoding and specific annotations. The emitted segmentation contains a segment for each imported file.



Text Files (0% co...

☑ Advanced settings

Sources

```
Le-socialisme.txt  (auto-detect)
hayek.txt          (auto-detect)
```

File path(s):

Encoding: utf8

Annotation key:

Annotation value:

Options

☑ Import file names with key filename

□ Auto-number with key: num

□ Send

2 segments sent to output (1992463 characters).

Fig. 78: Figure 2: **Text files** widget (advanced interface).

The advanced interface (see *figure 2* above) presents similarities with that of the *URLs*, *Recode*, and *Segment* widgets. The **Sources** section allows the user to select the input file(s) as well as their encoding, to determine the order in which they appear in the output segmentation, and optionally to assign an annotation. The list of imported files appears at the top of the window; the columns of this list indicate (a) the name of each file, (b) the corresponding annotation (if any), and (c) the encoding with which each is associated.

In *figure 2*, we can see that two files are imported and that each is provided with an annotation whose key is *author*. The first file associates value *Dickens* with this key and is encoded in utf-8; the second one has value *Fitzgerald* and is encoded in iso-8859-1.

The first buttons on the right of the imported files' list enable the user to modify the order in which they appear in the output segmentation (**Move Up** and **Move Down**), to delete a file from the list (**Remove**) or to completely empty it (**Clear All**). Except for **Clear All**, all these buttons require the user to previously select an entry from the list. **Import List** enables the user to import a file list in JSON format (see *JSON im-/export format*, *File list*) and to add it to the previously selected sources. In the opposite **Export List** enables the user to export the source list in a JSON file.

The remainder of the **Sources** section allows the user to add new files to the list. The easiest way to do so is to first click on the **Browse** button, which opens a file selection dialog. After having selected one or more files in this dialog and validated the choice by clicking on **Open**, the files appear in the **File paths** field and can be added to the list by clicking on the **Add** button. It is also possible to type the complete paths of the files directly in the text field, separating the paths corresponding to the successive files with the string ” ” (space + slash + space).

Before adding one or more files to the list by clicking on **Add**, it is possible to select their encoding (**Encoding**), and to assign an annotation by specifying its key in the **Annotation key** field and the corresponding value in the **Annotation value** field. These three parameters (encoding, key, value) will be applied to each file appearing in the **File paths** field at the moment of their addition to the list with **Add**.

The **Options** section allows the user to specify the label affected to the output segmentation. The **Import filenames with key** checkbox enables the program to create for each imported file an annotation whose value is the file name (as displayed in the list) and whose key is specified by the user in the text field on the right of the checkbox. Similarly the button **Auto-number with key** enables the program to automatically number the imported files and to associate the number to the annotation key specified in the text field on the right.

In *figure 2*, it was thus decided to assign the label *novels* to the output segmentation, and to associate the name of each file to the annotation key *filename*. On the other hand, the auto-numbering option has not been enabled.

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface.

The text below the **Send** button indicates the length of the output segmentation in characters, or the reasons why no segmentation is emitted (no selected file, encoding issue, etc.). In the example, the two segments corresponding to the imported files thus total up to 1'262'145 characters.

Remote control

Text Files is one the widgets that can be controlled by means of the *Message* widget. Indeed, it can receive in input a message consisting of a file list in JSON format (see *JSON im-/export format*, *File list*), in which case the list of files specified in this message replaces previously imported sources (if any). Note that removing the incoming connection from the **Message** instance will not, by itself, remove the list of files imported in this way from the **Text Files** instance's interface; conversely, this list of files can be modified using buttons **Move up/down**, **Remove**, etc. even if the incoming connection from the **Message** instance has not been removed. Finally, note that if a **Text Files** instance has the basic version of its interface activated when an incoming connection is created from an instance of *Message*, it automatically switches to the advanced interface.

Messages

Information

Data correctly sent to output: <n> segments (<m> characters). This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no file selected. The widget instance is not able to emit data to output because no input file has been selected.

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* and *Errors* below).

Warnings

No label was provided. A label must be entered in the **Output segmentation label** field in order for computation and data emission to proceed.

No annotation key was provided for auto-numbering. The **Auto-number with key** checkbox has been selected and an annotation key must be specified in the text field on the right in order for computation and data emission to proceed.

JSON message on input connection doesn't have the right keys and/or values. The widget instance has received a JSON message on its `Message` input channel and the keys and/or values specified in this message do not match those that are expected for this particular widget type (see *JSON im-/export format*, *File list*).

JSON parsing error. The widget instance has received data on its `Message` input channel and the data couldn't be correctly parsed. Please use a JSON validator to check the data's well-formedness.

Errors

Couldn't open file **or** ***Couldn't open file '<filepath>'.*** A file couldn't be opened and read, typically because the specified path is wrong.

Encoding error **or** ***Encoding error: file '<filepath>'.*** A file couldn't be read with the specified encoding (it must be in another encoding).

Examples

- *Cookbook: Import text from file*

See also

- *Reference: JSON im-/export format*, *File list*
- *Reference: Message widget*

URLs

Fetch text data from internet locations.

Signals

Inputs:

- `Message`

  JSON Message controlling the list of imported URLs

Outputs:

- `Text data`

  Segmentation covering the content of imported URLs

Description

This widget is designed to import the contents of one or several internet locations (URLs) in Orange Canvas. It outputs a segmentation containing a potentially annotated) segment for the content of each selected URL. The imported textual content is normalized in several ways:

- it is systematically converted to Unicode (from the encoding defined by the user)
- it is subjected to the canonical Unicode decomposition-recomposition technique (Unicode sequences such as `LATIN SMALL LETTER C (U+0063)` + `COMBINING CEDILLA (U+0327)` are systematically replaced by the combined equivalent, e.g. `LATIN SMALL LETTER C WITH CEDILLA (U+00C7)`)
- it is stripped from the utf8 byte-order mark (if any)
- various forms of line endings (in particular `\r\n` and `\r`) are converted to a single form (namely `\n`)

The interface of **URLs** is available in two versions, according to whether or not the **Advanced Settings** checkbox is selected.

Basic interface

In its basic version (see *figure 1* below), the **URLs** widget is limited to the import of a single URL's content. The interface contains a **Source** section enabling the user to type the input URL and to select the encoding of its content.

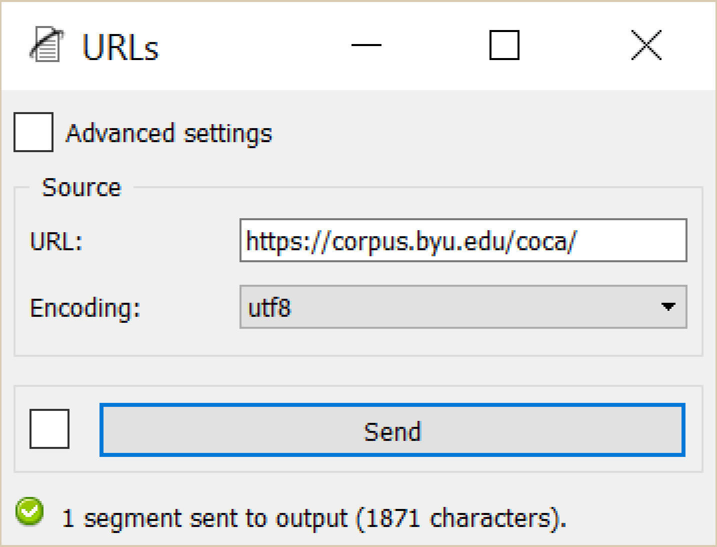
The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface.

Below the **Send** button, the user finds the number of characters in the single segment contained in the output segmentation, or the reasons why no segmentation is emitted (inability to retrieve the data, encoding issue, etc.).

Advanced interface

The advanced version of **URLs** allows the user to import the content of several URLs in a determined order; each URL can moreover be associated to a distinct encoding and specific annotations. The emitted segmentation contains a segment for the content of each imported URL.

The advanced interface (see *figure 2* above) presents similarities with that of the *Text Files*, *Recode*, and *Segment* widgets. The **Sources** section allows the user to specify the imported URL(s) as well as their content's encoding, to determine the order in which they appear in the output segmentation, and optionally to assign an annotation. The list of imported URLs appears at the top of the window; the columns of this list indicate (a) the URL, (b) the corresponding annotation (if any), and (c) the encoding with which the content of each is associated.



URL: https://corpus.byu.edu/coca/

Encoding: utf8

1 segment sent to output (1871 characters).

Fig. 79: Figure 1: **URLs** widget (basic interface).

In *figure 2*, we can see that two URLs are imported (only the end of each URL is visible on the figure) and that each is provided with an annotation whose key is *author*. The first URL associates value *Dickens* with this key and is encoded in utf-8; the second one has value *Fitzgerald* and is encoded in iso-8859-1.

The first buttons on the right of the imported URLs' list enable the user to modify the order in which they appear in the output segmentation (**Move Up** and **Move Down**), to delete an URL from the list (**Remove**) or to completely empty it (**Clear All**). Except for **Clear All**, all these buttons require the user to previously select an entry from the list. **Import List** enables the user to import a list of URLs in JSON format (see *JSON im-/export format*, *URL list*) and to add it to the previously selected sources. In the opposite **Export List** enables the user to export the source list in a JSON file.

The remainder of the **Sources** section allows the user to add new URLs to the list. these must first be inputted in the field with the same name before they can be added to the list by clicking on the **Add** button. In order for several URLs to be simultaneously added, they must be separated by the string " / " (space + slash + space).

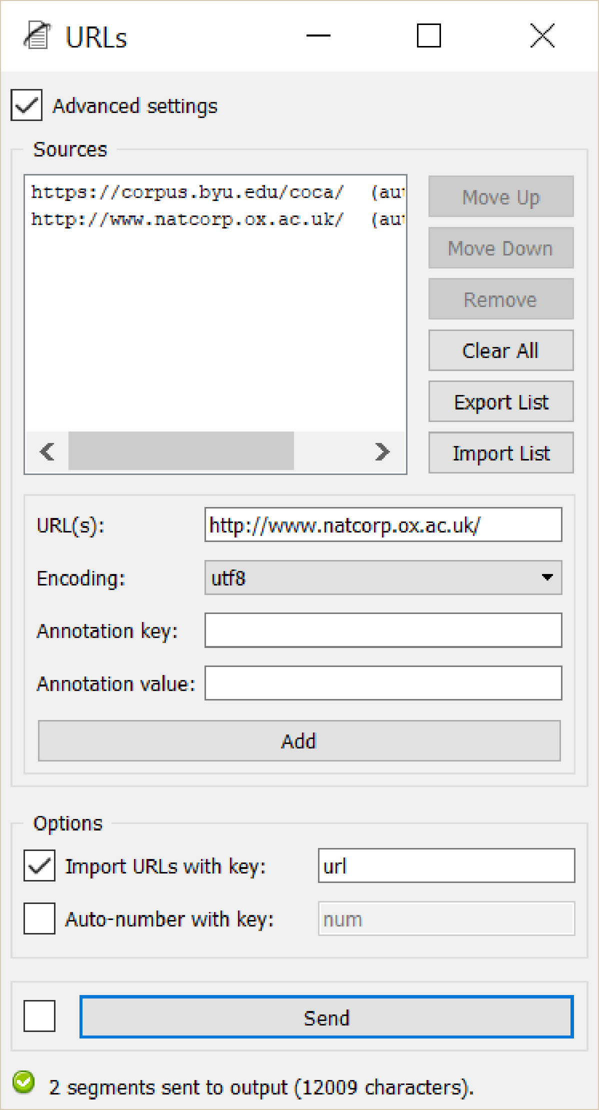
Before adding one or more URLs to the list by clicking on **Add**, it is possible to select their encoding (**Encoding**), and to assign an annotation by specifying its key in the **Annotation key** field and the corresponding value in the **Annotation value** field. These three parameters (encoding, key, value) will be applied to each URL appearing in the **URLs** field at the moment of their addition to the list with **Add**.

The **Import URLs with key** checkbox enables the program to create for each imported URL an annotation whose value is the URL (as displayed in the list) and whose key is specified by the user in the text field on the right of the checkbox. Similarly the button **Auto-number with key** enables the program to automatically number the imported URLs and to associate the number to the annotation key specified in the text field on the right.

In *figure 2*, it was thus decided to assign the label *novels* to the output segmentation, and to associate the name of each URL to the annotation key *url*. On the other hand, the auto-numbering option has not been enabled.

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface.

Below the **Send** button, the user finds the length of the output segmentation in characters, or the reasons why no segmentation is emitted (inability to retrieve the data, encoding issue, etc.). In the example, the two segments corresponding to the imported URLs' content thus total up to 1'300'344 characters.



URLs

☑ Advanced settings

Sources

```
https://corpus.byu.edu/coca/   (au
http://www.natcorp.ox.ac.uk/   (au
```

URL(s): http://www.natcorp.ox.ac.uk/

Encoding: utf8

Annotation key:

Annotation value:

Options

☑ Import URLs with key: url

□ Auto-number with key: num

□

2 segments sent to output (12009 characters).

Fig. 80: Figure 2: **URLs** widget (advanced interface).

Remote control

URLs is one the widgets that can be controlled by means of the *Message* widget. Indeed, it can receive in input a message consisting of a URL list in JSON format (see *JSON im-/export format*, *URL list*), in which case the list of URLs specified in this message replaces previously imported sources (if any). Note that removing the incoming connection from the **Message** instance will not, by itself, remove the list of URLs imported in this way from the **URLs** instance's interface; conversely, this list of files can be modified using buttons **Move up/down**, **Remove**, etc. even if the incoming connection from the **Message** instance has not been removed. Finally, note that if an **URLs** instance has the basic version of its interface activated when an incoming connection is created from an instance of *Message*, it automatically switches to the advanced interface.

Messages

Information

Data correctly sent to output: <n> segments (<m> characters). This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no URL selected. The widget instance is not able to emit data to output because no input URL has been selected.

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* and *Errors* below).

Warnings

No label was provided. A label must be entered in the **Output segmentation label** field in order for computation and data emission to proceed.

No annotation key was provided for auto-numbering. The **Auto-number with key** checkbox has been selected and an annotation key must be specified in the text field on the right in order for computation and data emission to proceed.

JSON message on input connection doesn't have the right keys and/or values. The widget instance has received a JSON message on its `Message` input channel and the keys and/or values specified in this message do not match those that are expected for this particular widget type (see *JSON im-/export format*, *File list*).

JSON parsing error. The widget instance has received data on its `Message` input channel and the data couldn't be correctly parsed. Please use a JSON validator to check the data's well-formedness.

Errors

Couldn't retrieve <URL>. An URL couldn't be retrieved and read, possibly because it is incorrect, or because the internet connexion has not been working properly.

Encoding error* or *Encoding error: <URL>. An URL couldn't be read with the specified encoding (it must be in another encoding).

Examples

- *Cookbook: Import text from internet location*

See also

- *Reference: JSON im-/export format*, *URL list*
- *Reference: Message widget*

1.6.2 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 (or export) the details of a given *Segmentation* object (content and address of segments, as well as their possible annotations).

Preprocess

Basic text preprocessing.

Signals

Inputs:

- `Segmentation`

  Segmentation covering the text that should be preprocessed

Outputs:

- `Text data`

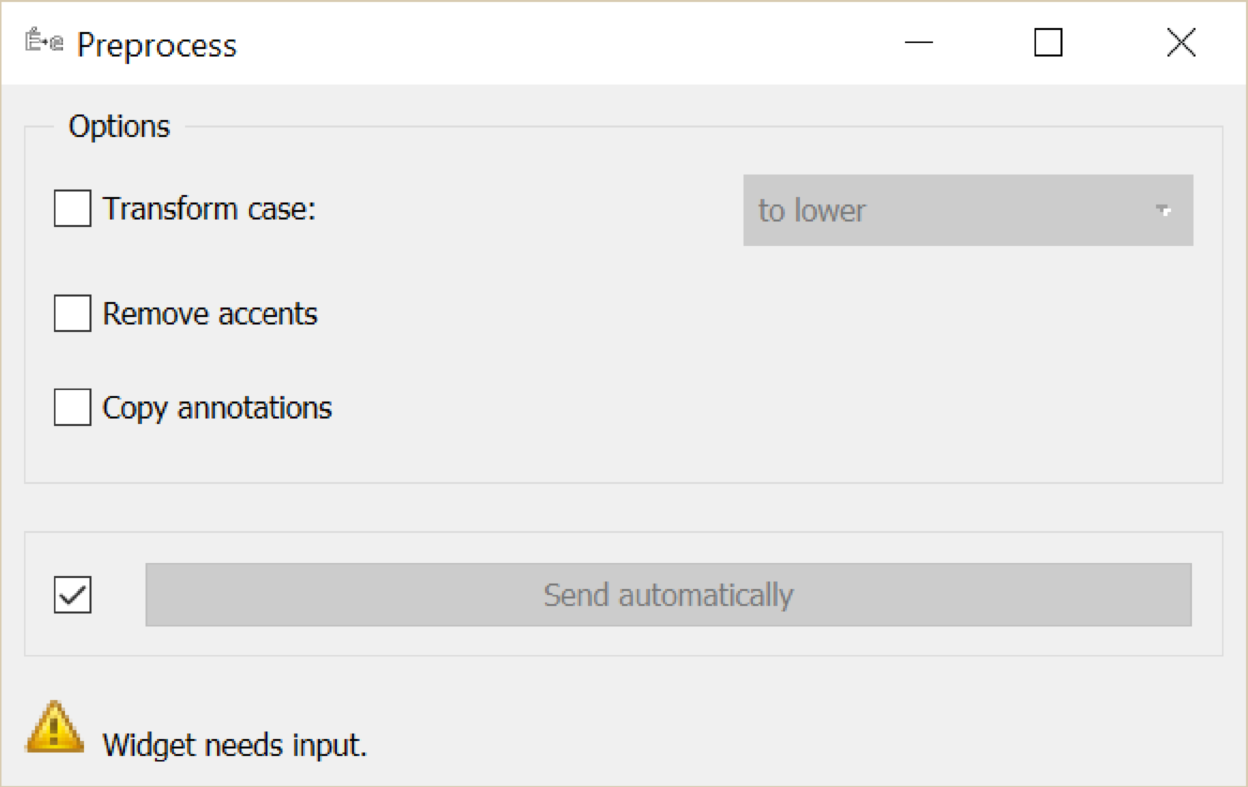
  Segmentation covering the modified text

Description

This widget inputs a segmentation, creates a modified copy of the content of the segmentation, and outputs a new segmentation corresponding to the modified data. The possible modifications are on the case (lower case/upper case) and the replacing of accentuated characters by their non-accentuated equivalents.

Note that **Preprocess** creates a copy of each modified segment, which increases the program's memory footprint; moreover this widget can only work on segmentations without any overlap, which means no part of the text is covered by more than one segment.

the **Transform case** checkbox triggers the systematic modification of the case: select **to lower** to convert every character to lower case and **to upper** to convert them to upper case. The **Remove accents** checkbox controls the replacement of accentuated character by their non-accentuated equivalents (é -> e, ç -> c, etc.).



Preprocess

Options

□ Transform case: to lower

□ Remove accents

□ Copy annotations

☑ Send automatically

Widget needs input.

Fig. 81: Figure 1: Interface of the **Preprocess** widget.

The **Copy annotations** button copies all the annotations of the input segmentation to the output segmentation; it is only accessible when the **Advanced settings** checkbox is selected (otherwise the annotations are by default copied).

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

Below the **Send** button, the user finds the number of segments present in the output segmentation, or the reasons why no segmentation is emitted (no input data, overlaps in the input segmentation, etc.).

Caveat

As one of the rare widgets of Textable that do create new *strings* and not only new *segmentations*, **Preprocess** is prone to a very specific and possibly disconcerting type of error, which can be best understood by studying an example.

Suppose that you wish to count word frequency in the content of two *Text Field* instances–a scenario similar to that illustrated in section *Counting in specific contexts*. You could use *Merge* to combine the *Text Field* instances' output in a single segmentation (see *figure 2* below), then segment the latter into words with *Segment*. You would eventually feed both the segmentation emitted by *Segment* (specifying units) and by *Merge* (specifying contexts) to an instance of *Count* for building the frequency table.

Moreover, suppose that you want to convert the input texts to lower case before counting word frequency. An intuitive way of performing this is by inserting a **Preprocess** instance between *Merge* and *Segment* as on *figure 3* below. However, because **Preprocess** creates a *new* string for each input segment and emits a segmentation that refers to these new strings, this raises a rather insidious issue.

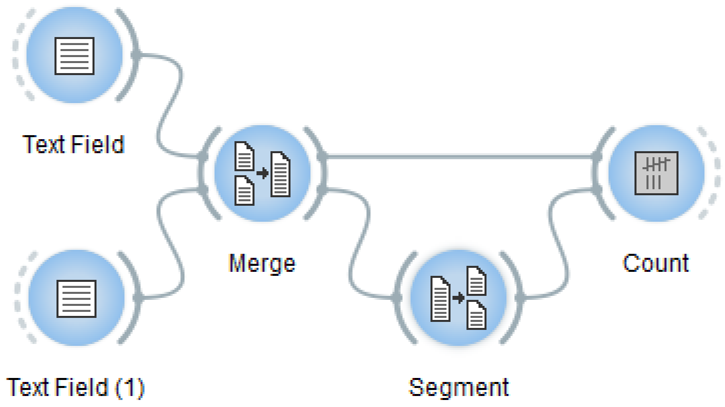


Fig. 82: Figure 2: Counting words in the content of two *Text Field* instances.

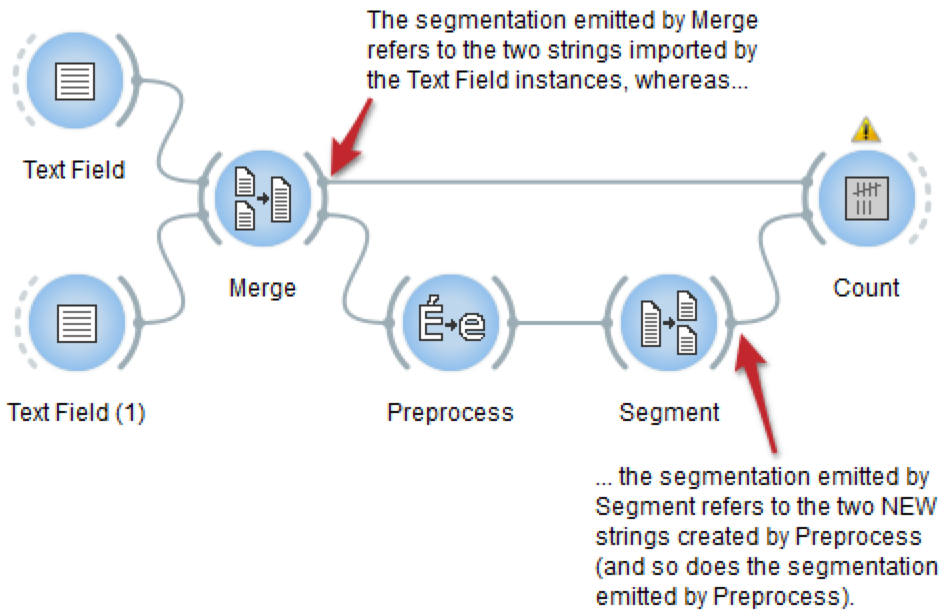

The segmentation emitted by Merge refers to the two strings imported by the Text Field instances, whereas...
Text Field
Merge
Count
Text Field (1)
Preprocess
Segment
... the segmentation emitted by Segment refers to the two NEW strings created by Preprocess (and so does the segmentation emitted by Preprocess).

Fig. 83: Figure 3: WRONG way of inserting a **Preprocess** instance in the schema.

To no effect, *Count* will attempt to find occurrences of the units specified by the segmentation received from *Segment* in the contexts specified by the segmentation received from *Merge*; since those actually belong to distinct strings, none of these units occurs in any of these contexts and the frequency table will remain hopelessly empty (as indicated by the warning symbol on top of the *Count* widget instance).

Luckily, a small wiring modification suffices to entirely solve the problem: the connection between *Merge* and *Count* should simply be replaced by a *direct* connection between **Preprocess** and *Count*, as on *figure 4* below. This way, units and contexts refer to the same set of strings and occurrences of the ones can be properly counted in the others.

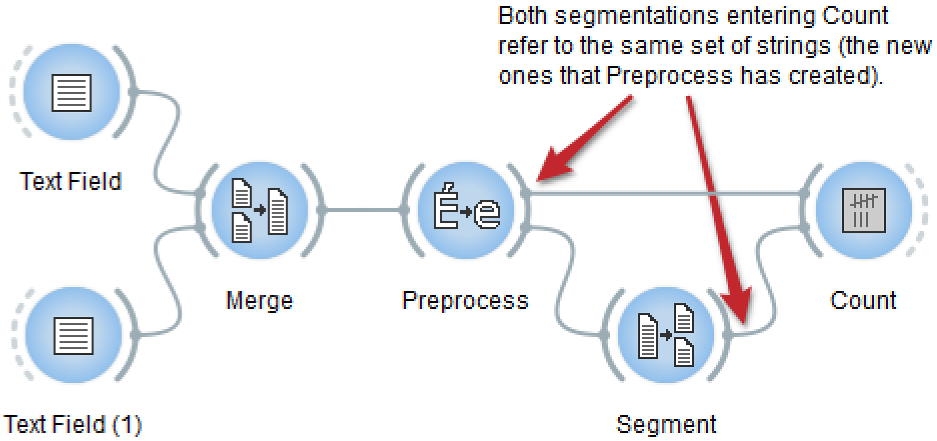

Both segmentations entering Count refer to the same set of strings (the new ones that Preprocess has created).
Text Field
Merge
Preprocess
Count
Text Field (1)
Segment

Fig. 84: Figure 4: RIGHT way of inserting **Preprocess**.

Messages

Information

Data correctly sent to output: <n> segments. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* below).

Warnings

No label was provided. A label must be entered in the **Output segmentation label** field in order for computation and data emission to proceed.

Input segmentation is overlapping. At least two of the input segments cover the same substring, which this widget cannot handle. Make sure every input segment covers a distinct substring.

Examples

- Getting started: Merging segmentations together
- Getting started: Annotating by merging
- *Cookbook: Merge several texts*

See also

- Getting started: Tagging table rows with annotations

Examples

- *Cookbook: Convert text to lower or upper case*
- *Cookbook: Remove accents from text*

See also

- *Getting started: Counting in specific contexts*
- *Reference: Text Field widget*
- *Reference: Merge widget*
- *Reference: Segment widget*
- *Reference: Count widget*

Recode

Custom text recoding using regular expressions.

Signals

Inputs:

- `Segmentation`

  Segmentation covering the text that should be recoded

- `Message`

  JSON Message controlling the list of substitutions

Outputs:

- `Recoded text data`

  Segmentation covering the recoded text

Description

This widget inputs a segmentation, creates a modified copy of its content, and outputs a new segmentation corresponding to the modified data. The modifications applied are defined by *substitutions*, namely pairs composed of a regular expression (designed to identify portions of text that should be modified) and a replacement string.

It is possible to "capture" text portions using parentheses appearing in the regular expressions, in order to insert them in the replacement strings, where sequences `&1`, `&2`, etc. correspond to the successive pairs of parentheses (numbered on the basis of the position of the opening parenthesis).

Note that **Recode** creates a copy of each modified segment, which increases the program's memory footprint; moreover this widget can only work on segmentations without any overlap, which means no part of the text is covered by more than one segment.

The interface of **Recode** is available in two versions, according to whether or not the **Advanced Settings** checkbox is selected.

Basic interface

The basic version of the widget is limited to the application of a single substitution. Section **Substitution** (see *figure 1* below) allows the user to specify the regular expression (**Regex**) and the corresponding replacement string (**Replacement string**). If the replacement string is left empty, the text parts identified by the regular expression will simply be deleted; it is the case in the example of *figure 1*, which leads to the deletion of XML/HTML tags.[1]

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

Below the **Send** button, the user finds all the indications regarding the current status of the widget instance (see *Messages* below, section *Information*).

Advanced interface

In its advanced version, the **Recode** widget allows the user to define several substitutions and to determine the order in which they should successively be applied to each segment of the input segmentation.

The advanced interface (see *figure 2* above) presents similarities with that of the *Text Files*, *URLs*, and *Segment* widgets. The **Substitutions** section allows the user to define the substitutions applied to each successive input segment and to determine their application order. In the list displayed at the top of the window, each line specifies a substitution, and the columns indicate for each substitution (a) the corresponding regular expression, (b) the (possibly empty) replacement string, and (c) the options associated with the regular expression.[2]

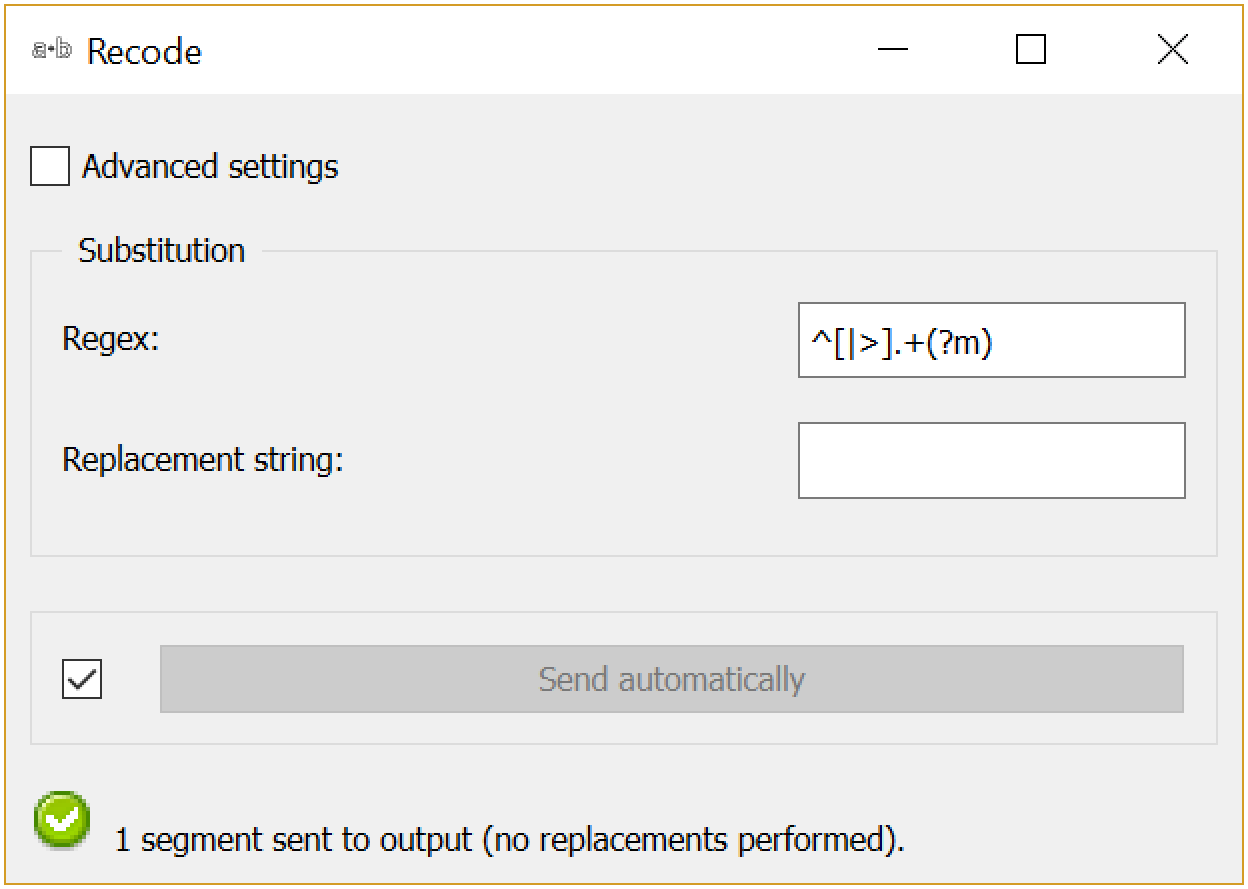
On *figure 2* above, we can see that three substitutions have been specified. The first deletes XML/HTML tags (it replaces them with the empty string). The second replaces occurrences of British English forms (*behaviour*, *colour*, and *neighbour*, possibly capitalized, since the *Ignore case* option is selected) with their American English variants (*behavior*, *color*, and *neighbor*), while the last replaces sequences like *a X of mine* with *my X*; thus they illustrate the possibility to "capture" text portions through parentheses appearing in the regular expression.

To take a concrete example, the successive application of these three substitutions to string

```
<example>I've just met a neighbour of mine.</example>
```

[1] For more details concerning the regular expression syntax, see the Python documentation. Note that option `-u` (*Unicode dependent*) is activated by default.

[2] For more details on the effect of options `-i`, `-u`, `-m`, and `-s`, see the Python documentation.



□ Advanced settings

Substitution

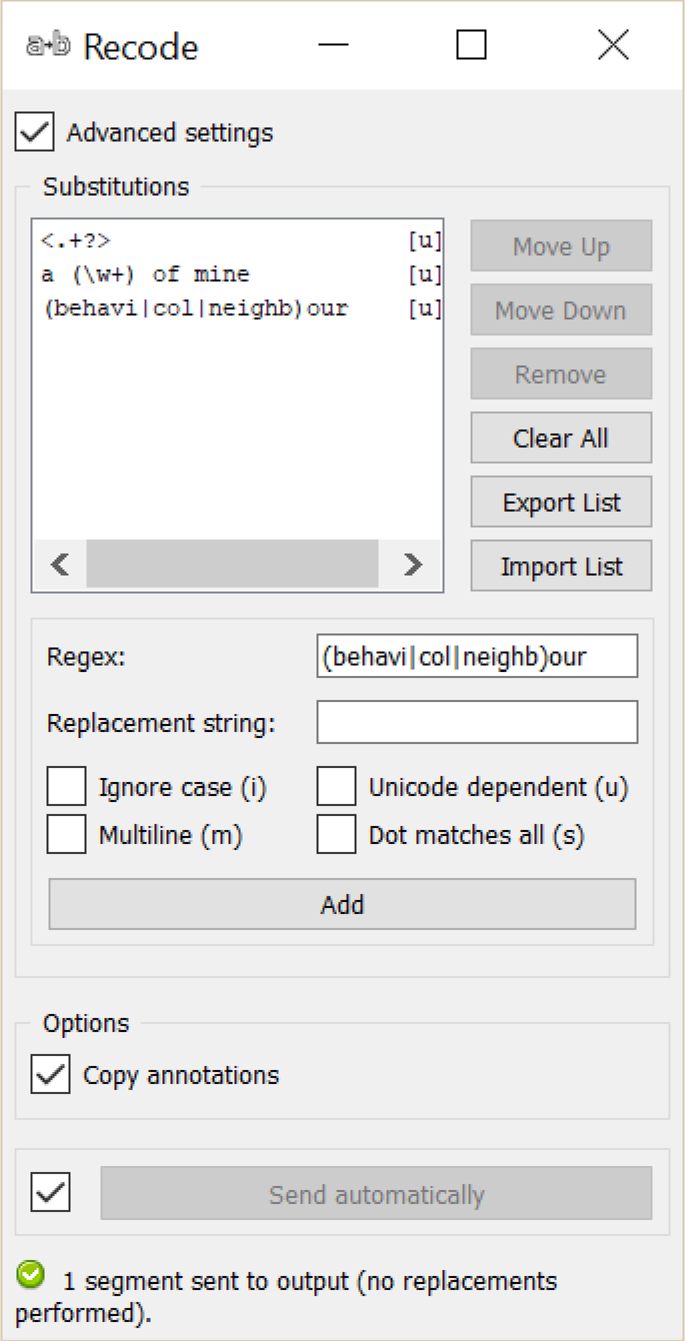
Regex: ^[|>].+(?m)

Replacement string:

☑ Send automatically

1 segment sent to output (no replacements performed).

Fig. 85: Figure 1: **Recode** widget (basic interface).



☑ Advanced settings

```
<.+?>                          [u]
a (\w+) of mine                [u]
(behavi|col|neighb)our         [u]
```

Regex: (behavi|col|neighb)our

Replacement string:

□ Ignore case (i) □ Unicode dependent (u)

□ Multiline (m) □ Dot matches all (s)

☑ Copy annotations

☑ Send automatically

1 segment sent to output (no replacements performed).

Fig. 86: Figure 2: **Recode** widget (basic interface).

will produce in turns the modified versions

```
I've just met a neighbour of mine.
```

```
I've just met a neighbor of mine.
```

```
I've just met my neighbor.
```

The first buttons on the right of the substitution list allow the user to modify the order in which they are successively applied to each segment of the input segmentation (**Move Up** and **Move Down**), to delete a substitution from the list (**Remove**) or to empty it entirely (**Clear All**). Except for **Clear All**, all of these buttons require the selection of an entry in the list beforehand. **Import List** enables the user to import a list of substitutions in JSON format (see *JSON im-/export format*, *Substitution list*) and to add them to those already selected. **Export List** enables the user on the contrary to export the list of substitutions in a JSON format file.

The remaining part of the **Substitutions** section allows the user to add new substitutions to the list. To define a new substitution, one must specify the regular expression (**Regex**) and the corresponding replacement string (**Replacement string**); the latter can be left empty, in which case the text portions identified by the regular expression will simply be deleted. The **Ignore case (i)**, **Unicode dependent (u)**, **Multiline (m)** and **Dot matches all (s)** checkboxes control the application of the corresponding options to the regular expression. Adding the new substitution to the list is achieved by clicking on the **Add** button.

The **Options** section allows the user to define the output segmentation label. The **Copy annotations** checkbox copies every annotation of the input segmentation to the output segmentation.

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

Below the **Send** button, the user finds all the indications regarding the current status of the widget instance (see *Messages* below, section *Information*).

Remote control

Recode is one the widgets that can be controlled by means of the *Message* widget. Indeed, it can receive in input a message consisting of a substitution list in JSON format (see *JSON im-/export format*, *Substitution list*), in which case the list of substitutions specified in this message replaces those previously specified (if any). Note that removing the incoming connection from the **Message** instance will not, by itself, remove the list of substitutions imported in this way from the **Recode** instance's interface; conversely, this list of files can be modified using buttons **Move up/down**, **Remove**, etc. even if the incoming connection from the **Message** instance has not been removed. Finally, note that if a **Recode** instance has the basic version of its interface activated when an incoming connection is created from an instance of *Message*, it automatically switches to the advanced interface.

Caveat

As one of the rare widgets of Textable that do create new *strings* and not only new *segmentations*, **Recode** is prone to a very specific and possibly disconcerting type of error, which can be best understood by studying the example given in the documentation of *Preprocess* (section *Caveat*), where all that is said about *Preprocess* also applies to **Recode**.

Messages

Information

Data correctly sent to output: <n> segments. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* and *Errors* below).

Warnings

No label was provided. A label must be entered in the **Output segmentation label** field in order for computation and data emission to proceed.

Input segmentation is overlapping. The instance's input segmentation contains overlapping segments, which preempts the application of recoding operations.

JSON message on input connection doesn't have the right keys and/or values. The widget instance has received a JSON message on its `Message` input channel and the keys and/or values specified in this message do not match those that are expected for this particular widget type (see *JSON im-/export format*, *Substitution list*).

JSON parsing error. The widget instance has received data on its `Message` input channel and the data couldn't be correctly parsed. Please use a JSON validator to check the data's well-formedness.

Errors

Regex error: <error_message>. The regular expression entered in the **Regex** field is invalid.

Regex error: <error_message> (substitution #<n>). The *n*-th regular expression in the **Substitutions** list is invalid.

Reference to unmatched group in replacement string. A replacement string specified by the user contains a reference to a numbered variable (`&1`, `&2`, `...`) which turns out to not always have a matching element.

Examples

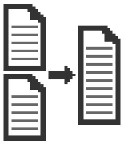
- *Cookbook: Replace all occurrences of a string/pattern*

See also

- *Reference: JSON im-/export format*, *Substitution list*
- *Reference: Message widget*
- *Reference: Preprocess (section "Caveat")*
- *Getting started: A note on regular expressions*

Footnotes

Merge



Merge two or more segmentations.

Signals

Inputs:

- `Segmentation` (multiple)

  Any number of segmentations that should be merged together

Outputs:

- `Merged data`

  Merged segmentation

Description

This widget takes several input segmentations, successively copies each segment of each input segmentation to form a new segmentation, and sends this segmentation to its output connections.

The **Options** section allows the user to import and label segments. The **Import labels with key** checkbox enables the user to create for each input segmentation an annotation whose value is the segmentation label (as displayed in the list) and whose key is specified by the user in the text field on the right of the checkbox. Similarly, the **Auto-number with key** checkbox enables the program to automatically number the output segments and to associate the number to the annotation key specified in the text field on the right. The **Copy annotations** checkbox copies every input segmentation annotation to the output segmentation.

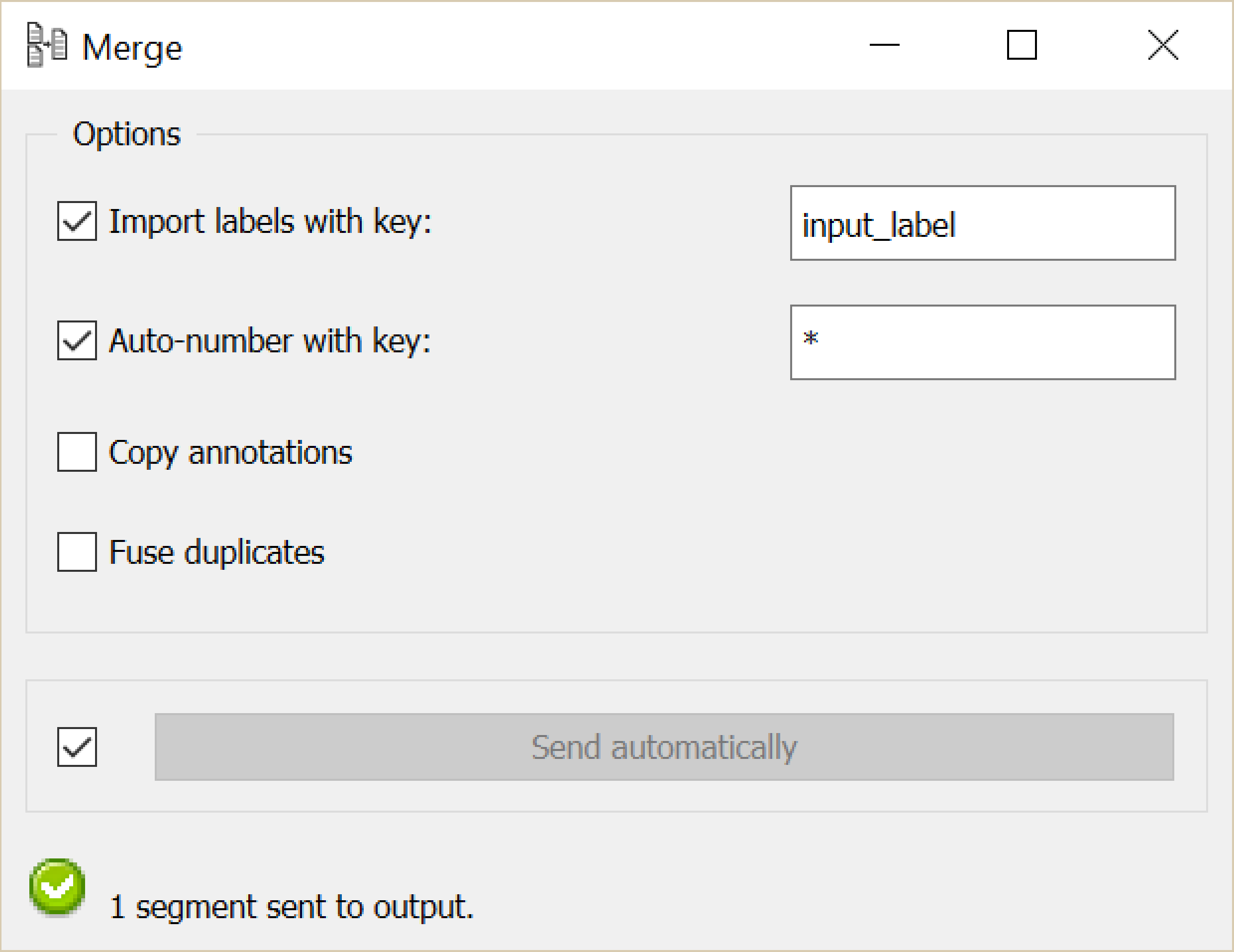
[1] The **Fuse duplicate segments** checkbox enables the program to fuse into a single segment several distinct segments whose addresses are the same; the annotations associated to the fused segments are all copied in the single resulting segment.[2]

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

Below the **Send** button, the user finds the number of segments in the output segmentation, or the reasons why no segmentation is emitted (no input data, no label specified for the output segmentation, etc.).

[1] Note that if sorting is enabled, it may well result in segments being ordered in a different way than specified by the user in the **Ordering** section.

[2] In the case where the fused segments have distinct values for the same annotation key, only the value of the last segment (in the order of the output segmentation before fusion) will be retained.



Merge

Options

☑ Import labels with key: input_label

☑ Auto-number with key: *

☐ Copy annotations

☐ Fuse duplicates

☑ Send automatically

1 segment sent to output.

Fig. 87: Figure 1: **Merge** widget (advanced interface).

Messages

Information

Data correctly sent to output: <n> segments. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* below).

Warnings

No label was provided. A label must be entered in the **Output segmentation label** field in order for computation and data emission to proceed.

No annotation key was provided for imported labels. The **Import labels with key** checkbox has been selected and an annotation key must be specified in the text field on the right in order for computation and data emission to proceed.

No annotation key was provided for auto-numbering. The **Auto-number with key** checkbox has been selected and an annotation key must be specified in the text field on the right in order for computation and data emission to proceed.

Examples

- Getting started: Merging segmentations together
- Getting started: Annotating by merging
- *Cookbook: Merge several texts*

See also

- Getting started: Tagging table rows with annotations

Footnotes

Segment

Subdivide a segmentation using regular expressions.

Signals

Inputs:

- `Segmentation`

  Segmentation that should be further segmented

- `Message`

  JSON Message controlling the list of regular expressions

Outputs:

- `Segmented data`

  Segmentation containing the newly created segments

Description

This widget inputs a segmentation and creates a new segmentation by subdividing each original segment into a series of new segments. By default, it works on the basis of a description of the form of the new segments (by means of regular expressions); alternatively, it can also operate based on a description of the separators that appear in-between the segments. It also allows the user to create annotations for the output segments.

In the same way as for the *Recode* widget, it is possible to “capture” text portions with square brackets used in a regular expression, notably to copy them in the annotation key and/or in the associated value; for this we use the notations `&1`, `&2`, etc. corresponding to the pairs of successive brackets (numbered on the basis of the position of opening parentheses) of the regular expression.[1]

The interface of **Segment** is available in two versions, according to whether or not the **Advanced Settings** checkbox is selected.

Basic interface

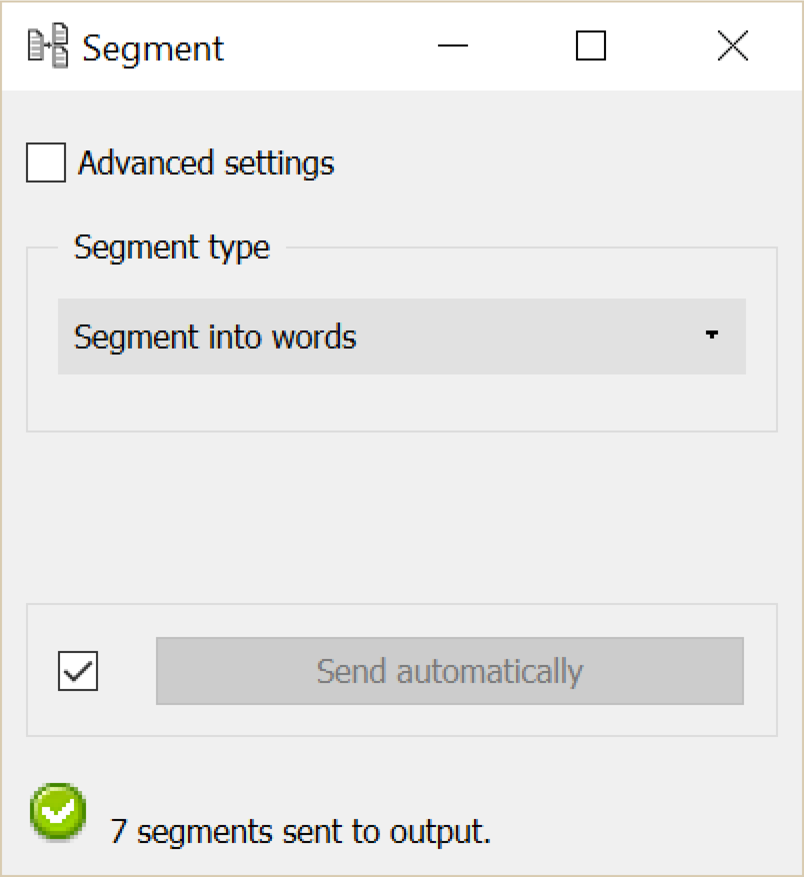
The basic version of the widget permits to choose between four types of Text Data segmentation output. The *Segment into Letters* option segments text data into letters; the *Segment into Words* option segments text data into words (which is mandatory in order to count segments, see cookbook); the *Segment into lines* option segments text data into lines. Eventually, *Use a regular expression* opens a short Regex section (see *figure 1* below). This Regex can be a particular string of characters (a word) or a regular expression. For instance,‘‘w+‘‘ creates a segment for each word of each input segment (to be precise: each continuous sequence of alphanumerical characters and underscores).[2]

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

Below the **Send** button the user finds all the indications regarding the current status of the widget instance (see *Messages* below, section *Information*).

[1] This possibility does not apply when the widget is configured to identify the separators rather than the segments themselves (**Mode: Split**, see *Advanced interface*).

[2] It should be noted that the `-u` (*Unicode dependent*) option is activated by default (see Python documentation).



□ Advanced settings

Segment type

Segment into words

☑ Send automatically

7 segments sent to output.

Fig. 88: Figure 1: **Segment** widget (basic interface).

Advanced interface

In its advanced version, the widget enables the user to define several regular expressions and to determine the order in which they should successively be applied to each segment of the input segmentation. It also allows the user to specify if a given regular expression describes the form of the targeted segments (**Tokenize** mode) or rather the form of the separators in-between these segments (**Split** mode).[3]

The advanced interface (see *figure 2* above) presents similarities with that of the *Text Files*, *URLs*, and *Recode* widgets. The **Regexes** section allows the user to define the regular expressions applied successively to each segment of the input segmentation and to determine their application order. In the list which appears on top of the window, the columns indicate (a) the mode associated to this regular expression, namely *t* for *Tokenize* (default) or *s* for *Split*, (b) the actual expression, (c) the corresponding annotation (if any), and (d) the options associated to this expression.

On *figure 2* above, we can thus see that four regular expressions have been defined, each in the **Tokenize** mode; each identifies a type of character in the input segmentation and assigns to it an annotation whose key is type. The character classes identified by the four expressions are not mutually exclusive, however after having successively applied them, the widget automatically sorts the segments (exactly like the **Sort segments** option of the *Merge* widget) and fuses those whose addresses are identical, given that the **Fuse duplicates** option is selected, which triggers the fusion of segments with identical addresses (see below). In the end, each character thus belongs to a single segment, whose value for the annotation key *type* is the last one that was assigned to it according to the regular expressions application order.

The first of the four expressions (`.`) creates a segment for each character and assigns the annotation value *other* to it. The second (`\w`) creates a segment for each alphanumerical character, and assigns the annotation value *consonant* to it. The last two respectively identify vowels (`[aeiouy]`) and digits (`[0-9]`) and annotate them as such. To illustrate the mechanism explained in the previous paragraph, it can be noted that before segment sorting and duplicate fusion, each vowel of the input segmentation is associated with three segments whose values for the annotation key type are (in order) *other*, *consonant*, and *vowel*; after sorting and fusion, only the last of these values is retained.

The first buttons on the right of the list of regular expressions allow the user to modify the order in which they are successively applied to each segment of the input segmentation (**Move Up** and **Move Down**), to delete an expression from the list (**Remove**) or to empty it entirely (**Clear All**). Except for **Clear All**, all of these buttons require the selection of an entry in the list beforehand. **Import List** enables the user to import a list of regular expressions in JSON format (see *JSON im-/export format*, *Regular expression list*) and to add them to those already selected. **Export List** enables the user on the contrary to export the list of regular expressions in a JSON file.

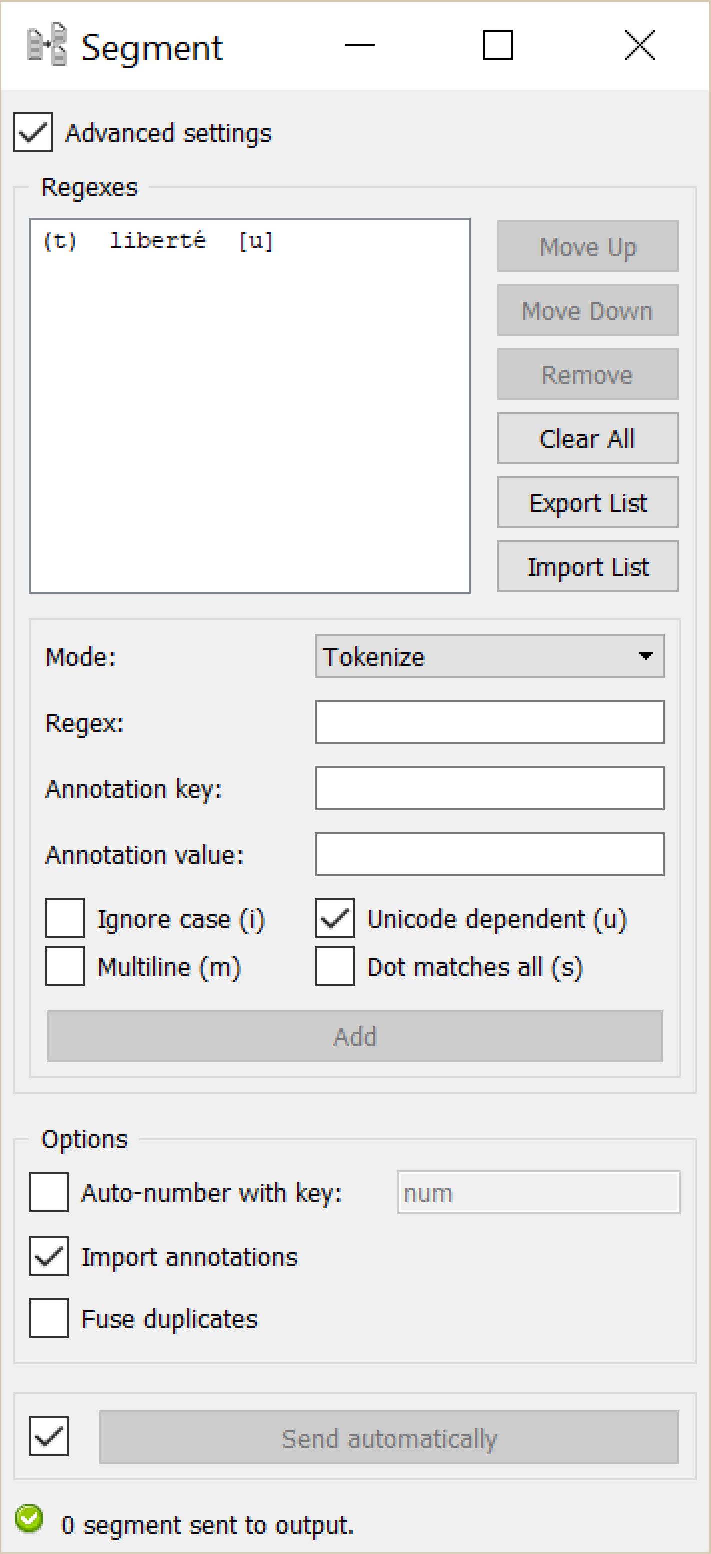
The remaining part of the **Regexes** section allows the user to add new regular expressions to the list. To do so, the regular expression should be specified (**Regex**) as well as, optionally, the annotation key and the corresponding value (**Annotation key** and **value**). The **Ignore case (i)**, **Unicode dependent (u)**, **Multiline (m)** and **Dot matches all (s)** checkboxes control the application of the corresponding options to the regular expressions. Adding the new regular expression to the list is executed by finally clicking on the **Add** button.

Through the **Options** section, the **Auto-number with key** checkbox enables the program to automatically number the output segments and to associate the number to the annotation key specified in the text field on the right. The **Import annotations** checkbox copies the annotations of each input segment to the corresponding output segments. The **Fuse duplicate segments** checkbox enables the program to fuse into a single segment several distinct segments whose addresses are identical; the annotations associated to the fused segments are all copied in the single resulting segment.[4]

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

[3] NB: in **Split** mode, empty segments that might occur between two consecutive occurrences of separators are automatically removed (this is because the data model adopted by Orange Canvas cannot represent empty segments).

[4] In the case where the fused segments have distinct values for the same annotation key, only the value of the last segment (in the order of the output segmentation before fusion) will be retained.



☑ Advanced settings

Regexes

```
(t)  liberté  [u]
```

Mode: Tokenize

Regex:

Annotation key:

Annotation value:

□ Ignore case (i) ☑ Unicode dependent (u)

□ Multiline (m) □ Dot matches all (s)

Options

□ Auto-number with key: num

☑ Import annotations

□ Fuse duplicates

☑ Send automatically

0 segment sent to output.

Fig. 89: Figure 2: **Segment** widget (advanced interface).

Below the **Send** button, the user finds indications regarding the current status of the widget instance (see *Messages* below, section *Information*).

Remote control

Segment is one the widgets that can be controlled by means of the *Message* widget. Indeed, it can receive in input a message consisting of a regular expression list in JSON format (see *JSON im-/export format*, *Regular expression list*), in which case the list of regular expressions specified in this message replaces those previously specified (if any). Note that removing the incoming connection from the **Message** instance will not, by itself, remove the list of regular expressions imported in this way from the **Segment** instance's interface; conversely, this list of files can be modified using buttons **Move up/down**, **Remove**, etc. even if the incoming connection from the **Message** instance has not been removed. Finally, note that if a **Segment** instance has the basic version of its interface activated when an incoming connection is created from an instance of *Message*, it automatically switches to the advanced interface.

Messages

Information

Data correctly sent to output: <n> segments. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* and *Errors* below).

Warnings

No regex defined. A regular expression must be entered in the **Regex** field in order for computation and data emission to proceed (in the advanced interface, the **Add** button must also be clicked).

No label was provided. A label must be entered in the **Output segmentation label** field in order for computation and data emission to proceed.

No annotation key was provided for auto-numbering. The **Auto-number with key** checkbox has been selected and an annotation key must be specified in the text field on the right in order for computation and data emission to proceed.

JSON message on input connection doesn't have the right keys and/or values. The widget instance has received a JSON message on its `Message` input channel and the keys and/or values specified in this message do not match those that are expected for this particular widget type (see *JSON im-/export format*, *Regular expression list*).

JSON parsing error. The widget instance has received data on its `Message` input channel and the data couldn't be correctly parsed. Please use a JSON validator to check the data's well-formedness.

Errors

Regex error: <error_message>. The regular expression entered in the **Regex** field is invalid.

Regex error: <error_message> (regex #<n>). The *n*-th regular expression in the **Regexes** list is invalid.

Reference to unmatched group in annotation key and/or value. In the advanced interface, a regular expression has been associated with an annotation key–value pair and in at least one of these terms reference is made to a numbered variable (`&1, &2, ...`) which turns out to not always have a matching element.

Examples

- *Getting started: Segmenting data into smaller units*
- *Cookbook: Segment text in smaller units*

See also

- *Reference: JSON im-/export format*, *Regular expression list*
- *Reference: Message widget*
- *Getting started: A note on regular expressions*

Footnotes

Select

Select a subset of segments in a segmentation.

Signals

Inputs:

- `Segmentation`

  Segmentation out of which a subset of segments should be selected

Outputs:

- `Selected data` (default)

  Segmentation containing the selected segments

- `Discarded data`

  Segmentation containing the discarded segments

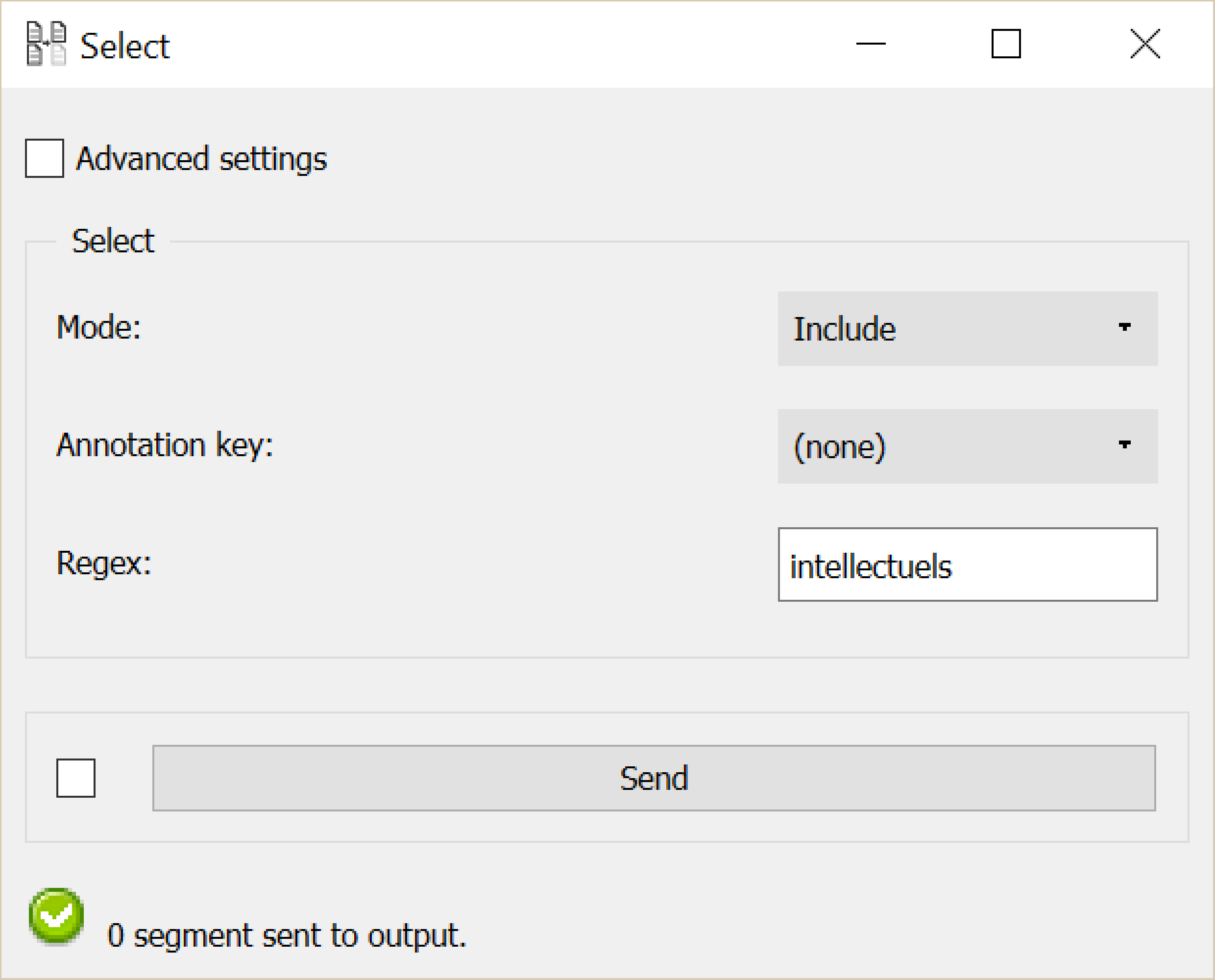
Description

This widget inputs a segmentation and creates a new segmentation including only some of the input segments. Segment selection can be based on their content, their annotations, or their frequency; it can also be random. No matter which method is used, the widget emits on a second output connection (not selected by default) a segmentation containing the segments that were *not* selected.

The interface of **Select** is available in two versions, according to whether or not the **Advanced Settings** checkbox is selected.

Basic interface

The basic version of the widget (see *figure 1* below) is limited to the selection of segments based on a regular expression (see *Method: Regex* in section *Advanced interface* below). The differences with the advanced interface are the following: (i) regular expression options are not accessible (-*u*, *Unicode dependent*, is nonetheless activated by default); (ii) auto-numbering is disabled; and (iii) annotations are copied by default.



Mode: Include

Annotation key: (none)

Regex: intellectuels

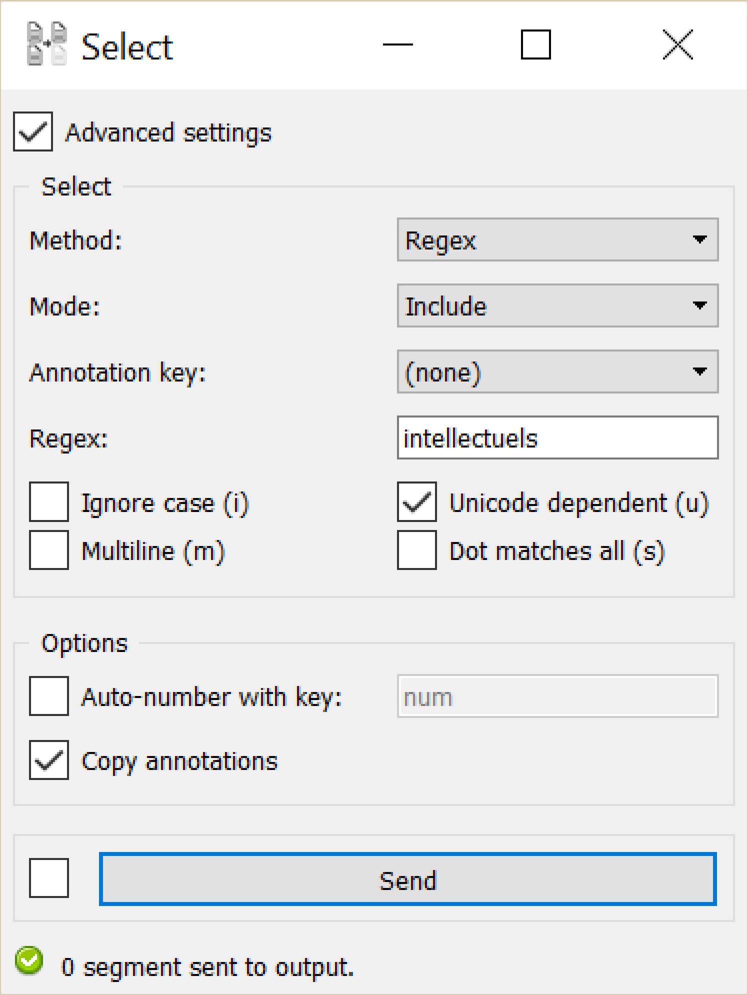
□ Send

0 segment sent to output.

Fig. 90: Figure 1: **Select** widget (basic interface).

Advanced interface

In its advanced version, the **Select** section of the widget interface comes in three versions depending on the value chosen in the **Method** drop-down menu (see figures *2* to *4* below).



Select

☑ Advanced settings

Select

Method: Regex

Mode: Include

Annotation key: (none)

Regex: intellectuels

□ Ignore case (i) ☑ Unicode dependent (u)

□ Multiline (m) □ Dot matches all (s)

Options

□ Auto-number with key: num

☑ Copy annotations

□ Send

0 segment sent to output.

Fig. 91: Figure 2: **Select** widget (advanced interface, **Regex** method).

Method: Regex

This method consists of selecting the segments of the input segmentation whose content or annotations are matched by a regular expression. The **Mode** drop-down menu (see *figure 2* above) allows the user to specify if the segments corresponding to the regular expression should be selected (**Include**) or not (**Exclude**), in which case the segments that do *not* correspond to the regular expression will be selected.

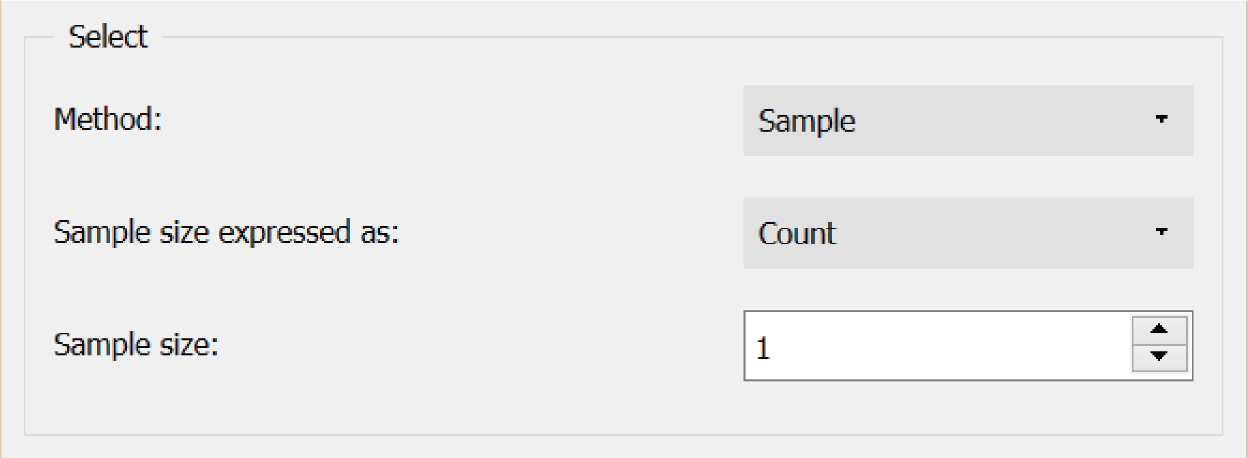
The **Annotation key** drop-down menu allows the user to choose an annotation key from the input segmentation; in that case, the segments whose annotation values for this key are matched by the regular expression will be selected (or not). If the value *(none)* is selected, the *content* of the segments will be matched against the regular expression.

The **Regex** field is designed to specify the regular expression used for segment selection, and the **Ignore case (i)**, **Unicode dependent (u)**, **Multiline (m)** and **Dot matches all (s)** checkboxes control the application of the corresponding options to this expression.

In the example of *figure 2* above, the widget is configured to include (**Mode: Include**) from the input segmentation the segments whose annotation value for key *category* (**Annotation key:** *category*) is either *noun* or *verb* (**Regex:** `^(noun|verb)$`).

Method: Sample

This method consists of selecting the segments of the input segmentation with a random sampling process, such that every input segment has the same probability of being selected or not.



Select

Method: Sample

Sample size expressed as: Count

Sample size: 1

Fig. 92: Figure 3: **Select** widget (advanced interface, **Sample** method).

The **Sample size expressed as** drop-down menu (see *figure 3* above) allows the user to choose the way in which to express the wanted size for the sample. If the value **Count** is selected, as on *figure 3*, the size of the sample will be expressed directly in the number of segments (**Sample size**). If the **Proportion** value is selected, the size will be expressed in percentage of input segments (**Sampling rate (%)**).

Method: Threshold

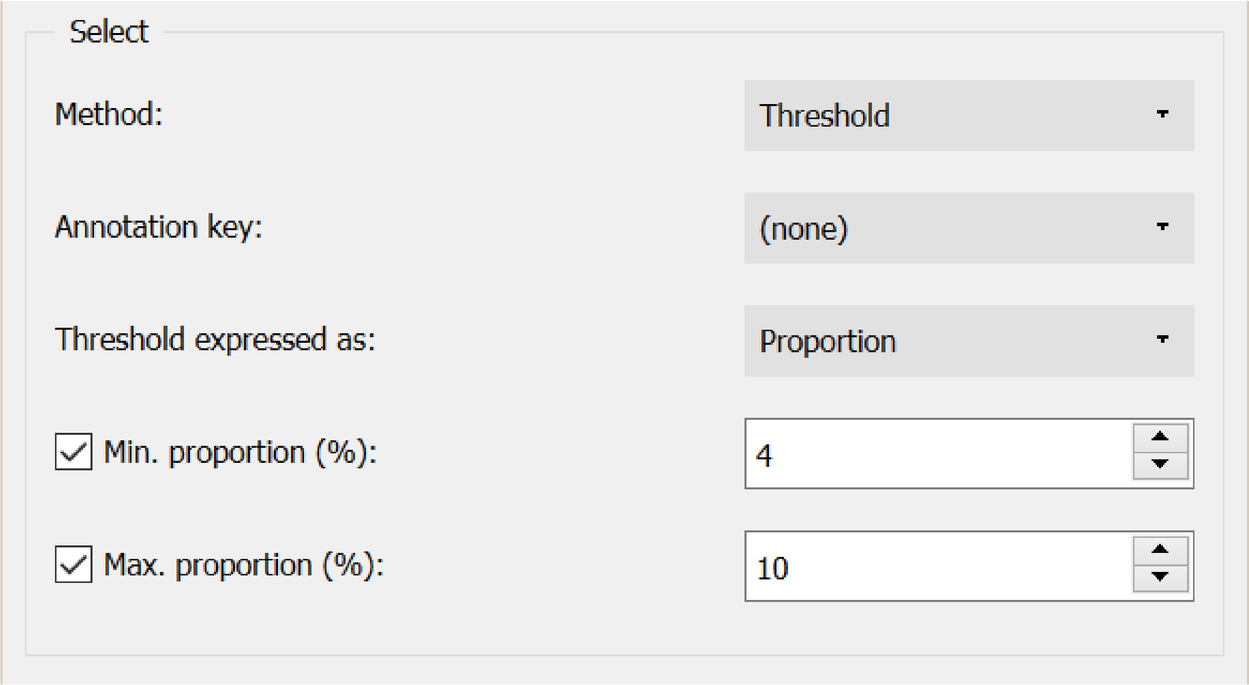
This method consists of retaining from the input segmentation only the segments whose content (or annotation value for a given key) has a frequency in the segmentation that is comprised between given bounds.

The **Annotation key** drop-down menu (see *figure 4* above) allows the user to select an annotation key from the input segmentation; if so, the frequency of the annotation values associated with this key will condition the inclusion of input segments. If the value *(none)* is selected, the frequency of the segment *content* will be decisive.

The **Threshold expressed as** drop-down menu allows the user to choose the way in which to express the minimal and maximal frequency limits. If the value **Count** is selected, the limits will be expressed in absolute frequencies (**Min./Max. count**). If the value **Proportion** is selected, as in *figure 4*, the limits will be expressed in percentages (**Min./Max. proportion (%)**). For both values (minimum and maximum), thresholding is applied only if the corresponding box is checked.

In the *figure 4* example, the widget is configured to retain only the segments whose annotation value for the key *category* (**Annotation key**) has a relative frequency (**Threshold expressed as: Proportion**) comprised between 5% (**Min. proportion (%)**) and 10% (**Max. proportion (%)**) in the input segmentation.

The elements of the **Options** section of the widget interface are common to the three selection methods presented above. The **Auto-number with key** checkbox enables the program to automatically number the segments of the output segmentation and to associate the number to the annotation key specified in the text field on the right. The **Copy annotations** checkbox copies every annotation of the input segmentation to the output segmentation.



Select

Method: Threshold

Annotation key: (none)

Threshold expressed as: Proportion

☑ Min. proportion (%): 4

☑ Max. proportion (%): 10

Fig. 93: Figure 4: **Select** widget (advanced interface, **Threshold** method).

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

Below the **Send** button, some indications are given about the number of segments in the output segmentation, or the reasons why no segmentation is emitted (no input data, no selected input segment, etc.).

Messages

Information

Data correctly sent to output: <n> segments. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* and *Errors* below).

Warnings

No regex defined. A regular expression must be entered in the **Regex** field in order for computation and data emission to proceed.

No label was provided. A label must be entered in the **Output segmentation label** field in order for computation and data emission to proceed.

No annotation key was provided for auto-numbering. The **Auto-number with key** checkbox has been selected and an annotation key must be specified in the text field on the right in order for computation and data emission to proceed.

Errors

Regex error: <error_message>. The regular expression entered in the **Regex** field is invalid.

Examples

- *Getting started: Partitioning segmentations*
- Getting started: Annotation-based selection
- *Cookbook: Include/exclude segments based on a pattern*
- *Cookbook: Filter segments based on their frequency*
- *Cookbook: Create a random selection or sample of segments*

Intersect

In-/exclude segments based on another segmentation.

Signals

Inputs:

- `Segmentation` (multiple)

  Segmentation out of which a subset of segments should be selected ("source" segmentation), or containing the segments that will be in-/excluded from the former ("filter" segmentation").

Outputs:

- `Selected data` (default)

  Segmentation containing the selected segments

- `Discarded data`

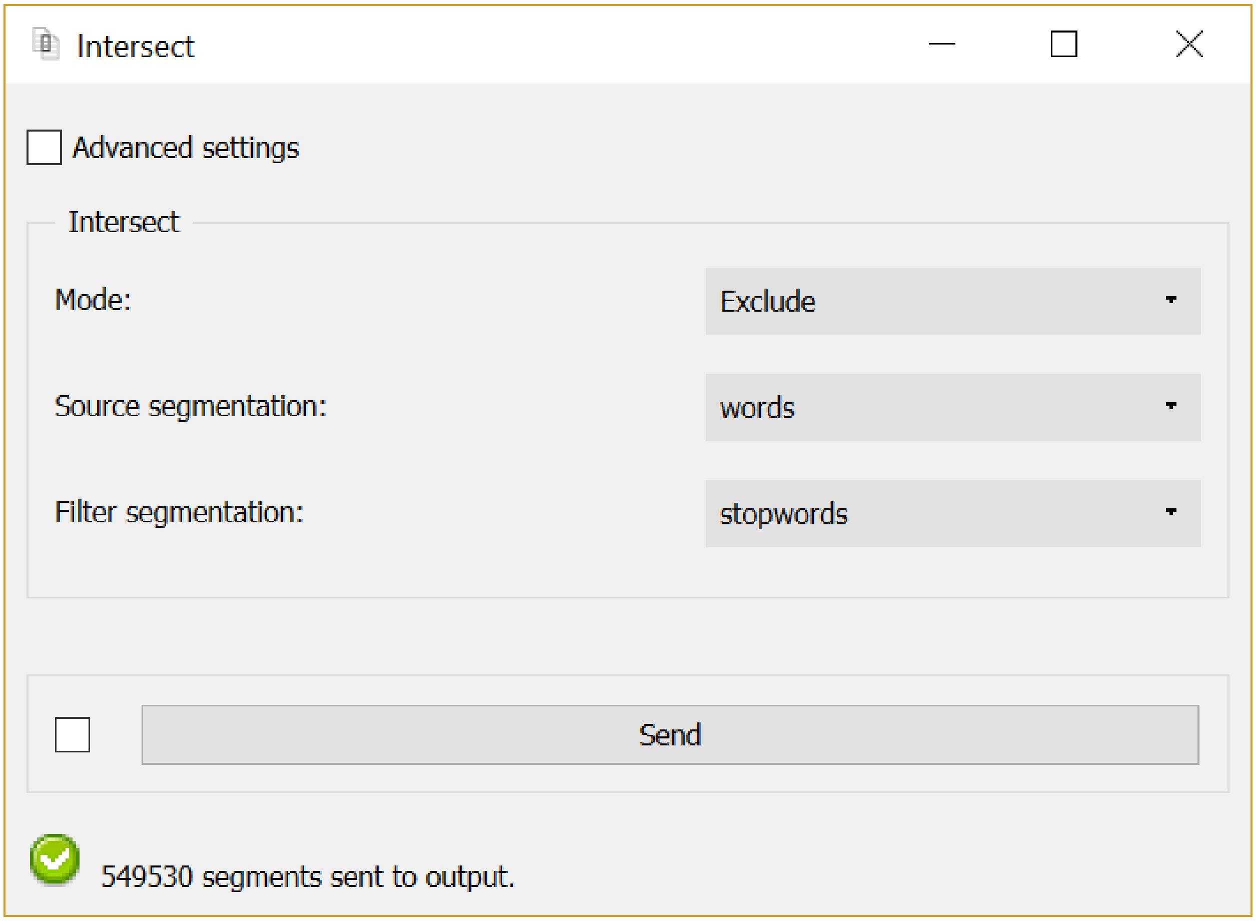
  Segmentation containing the discarded segments

Description

This widget inputs several segmentations and selects the segments of one of them ("source" segmentation) on the basis of the segments present in another ("filter" segmentation). It also emits on an output connection (not selected by default) a segmentation containing the segments that were *not* selected.

Basic interface

The **Intersect** section of the widget's basic interface (see *figure 1* above) allows the user to specify if the segments of the source segmentation that correspond to a type present in the filter segmentation should be included (**Mode: Include**) in the output segmentation or excluded (**Mode: Exclude**) from it. This section is also designed to select the source segmentation (**Source segmentation**) and the filter segmentation (**Filter segmentation**) among the input segmentations.[1]



☐ Advanced settings

Mode: Exclude

Source segmentation: words

Filter segmentation: stopwords

☐

549530 segments sent to output.

Fig. 94: Figure 1: **Intersect** widget (basic interface).

The **Source annotation** key drop-down menu allows the user to select an annotation key from the source segmentation; thus the segments whose annotation value for this key corresponds to a type present in the filter segmentation will be in-/excluded. If the value *(none)* is selected, the segment content will be decisive.

[1] It should be noted that the interface does not prevent the user from selecting the same segmentation as source and filter, which can only make sense if different values are selected in the **Source annotation key** and **Filter annotation key** menus (the latter being only available when the **Advanced settings** checkbox is selected).

Thus in *figure 1* above, the widget inputs two segmentations. The first (**Source segmentation**), whose label is *words*, is the result of the segmentation of a text in words, as performed with the *Segment* widget for instance. The second (**Filter segmentation**), whose label is *stopwords*, is the result of the segmentation in words of a list of so-called "stopwords" (articles, pronouns, prepositions, etc.)–typically deemed irrelevant for information retrieval.

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

Below the **Send** button, the number of segments in the output segmentation are indicated, or the reasons why no segmentation is emitted (no input data, no selected input segment, etc.).

Advanced interface

The main difference between the widget's basic and advanced interface is that in the latter, section **Intersect** includes a **Filter annotation key** drop-down menu and a **Source annotation key**.

If a given annotation key of the filter segmentation is selected in the drop-down menu of the **Filter annotation key**, the corresponding annotation value (rather than *content*) types will condition the in-/exclusion of the source segmentation segments. Since the **Source annotation key** drop-down menu is set on *(none)*, the content of input segments will determine the next steps (rather than the values of some annotation key). Concretely, the source segmentation segments (namely the words from the text) whose content matches that of a segment from the filter segmentation (namely a stopword) will be excluded (**Mode: Exclude**) from the output segmentation. By contrast, choosing the value **Include** would result in including as output only the stopwords from the text.

The advanced interface also offers two additional controls in section **Options**. The **Auto-number with key** checkbox enable the program to automatically number the segments from the output segmentation and to associate their number to the annotation key specified in the text field on the right. The **Copy annotations** checkbox copies every annotation from the input segmentation to the output segmentation.

Messages

Information

Data correctly sent to output: <n> segments. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* below).

Warnings

No label was provided. A label must be entered in the **Output segmentation label** field in order for computation and data emission to proceed.

No annotation key was provided for auto-numbering. The **Auto-number with key** checkbox has been selected and an annotation key must be specified in the text field on the right in order for computation and data emission to proceed.

Examples

- *Getting started: Using a segmentation to filter another*
- *Cookbook: Exclude segments based on a stoplist*

Footnotes

Extract XML

Create a new segmentation based on XML markup.

Signals

Inputs:

- `Segmentation`

  Segmentation covering XML data based on which a new segmentation will be created

Outputs:

- `Extracted data`

  Segmentation containing the segments corresponding to extracted XML elements

Description

This widget inputs a segmentation, searches in its content portions corresponding to a specific XML element type, and creates a segment for each occurrence of this element. It should be noted that if a given occurrence is distributed among several segments of the input segmentation, it will result in the creation of as many segments in the output segmentation.

Every attribute from extracted elements is automatically converted in annotation in the output segmentation. For example, extracting the element <*div*> in the following fragment:

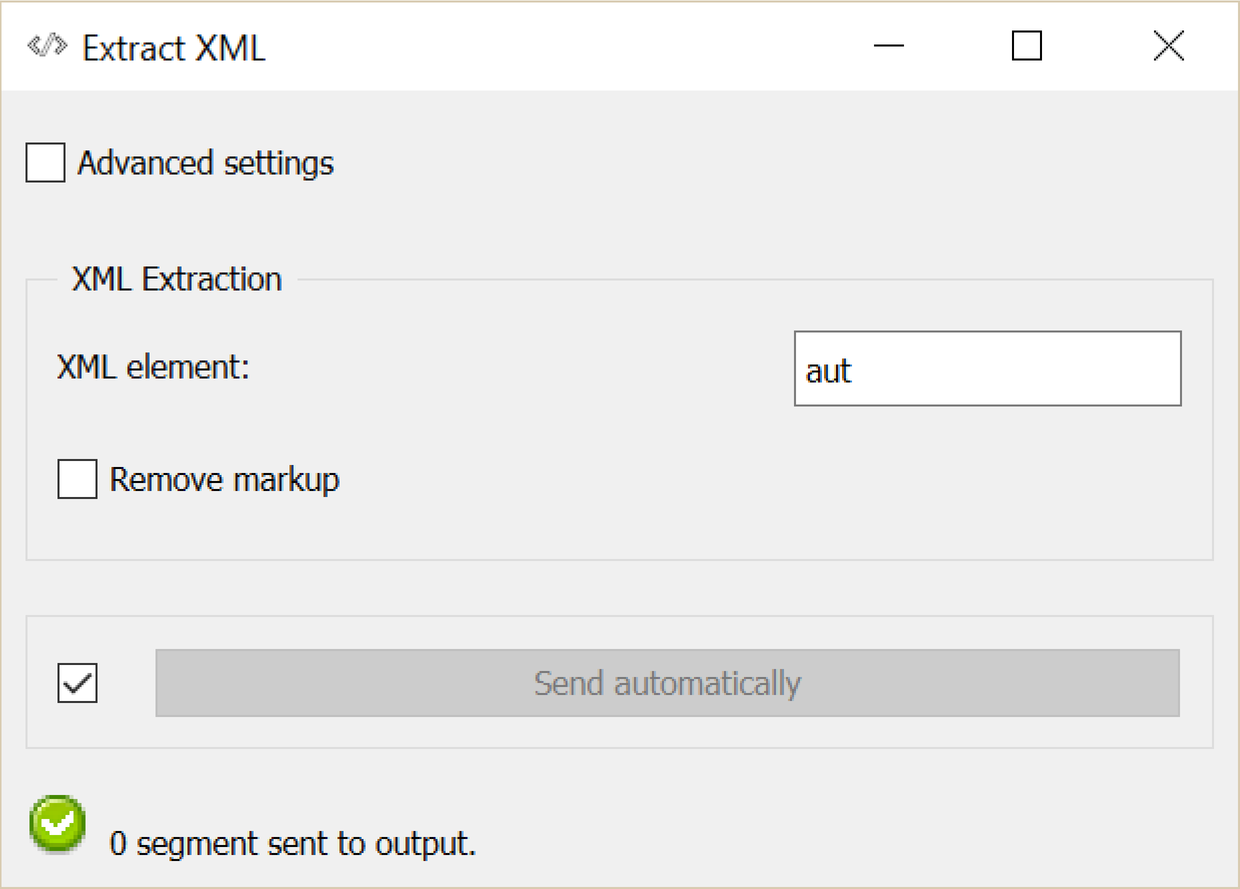
```
<div type="interjection">Cripes!</div>
```

will result in the creation of a segment whose content is *Cripes!* and whose annotation value for key *type* is *interjection*.

This widget offers the easiest and most flexible way to import into Orange Textable a segmentation and arbitrary annotations specified by the user for a given text. Let us however mention the following limitation: the widget automatically deletes all segments of zero length in the output segmentation. As a consequence, it is impossible to import empty XML elements (be they in the form <*element*></*element*> or <*element/*>).

Basic interface

In the basic widget interface (see *figure 1* below), the **XML Extraction** section allows the user to specify the XML element to extract (**XML element**). The widget indeed only allows the extraction of a single type of element at a time; however, it extracts every occurrence of this element, including those embedded in other occurrences of the same type.



Extract XML

□ Advanced settings

XML Extraction

XML element: aut

□ Remove markup

☑ Send automatically

0 segment sent to output.

Fig. 95: Figure 1: **Extract XML** widget (basic interface).

The **Remove markup** checkbox triggers the deletion of XML tags embedded within the extracted XML elements, if any. An important consequence of the use of this option is that the extracted elements will potentially be decomposed in several segments corresponding to portions of their content which are separated by the deleted XML tags (see *Advanced interface* for an example of this mechanism[1]).

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

Below, the **Send** button, the user finds indications such as the number of segments in the output segmentation, or the reasons why no segmentation is emitted (no input data, no output segment created, etc.).

[1] In comparison with the advance interface, it should also be noted that in the basic interface the options **Prioritize shallow attributes** and **Fuse duplicates** are disabled by default.

Advanced interface

The XML Extraction section of the widget interface (see *figure 2* below) allows the user to configure the XML element extraction. The field **XML element** allows the user to indicate the XML element type which should be sought. The **Import element with key** checkbox enables the program to assign to each output segment an annotation whose key is the text contained in the field immediately on the right and whose value is the name of the XML element extracted by the widget.

If the **Remove markup** checkbox is selected, XML tags embedded within the extracted XML elements will be excluded from the output segmentation. An important consequence of the use of this option is that the extracted elements will potentially be decomposed in several segments corresponding to portions of their content which are separated by the excluded XML tags. For example, given the following fragment:

```
<text>a <keyword>fragment</keyword> of XML data</text>
```

the extraction of element <*text*> will lead to the creation of three segments:

```
a
```

```
fragment
```

```
of XML data
```

If on the other hand the **Remove markup** option is not selected, a single segment will be created:

```
a <keyword>fragment</keyword> of XML data
```

The **Prioritize shallow attributes** checkbox determines the behavior of the widget in the very particular case where (a) elements of the extracted type are (exactly) embedded in one another, (b) they have different values for the same attribute, (c) the **Remove markup** option is selected and (d) the **Fuse duplicates** option (section **Options**) as well. This could be the case in the extraction of the <*div*> element in the following fragment for example:

```
<div type="A"><div type="B">two exactly embedded elements</div></div>
```

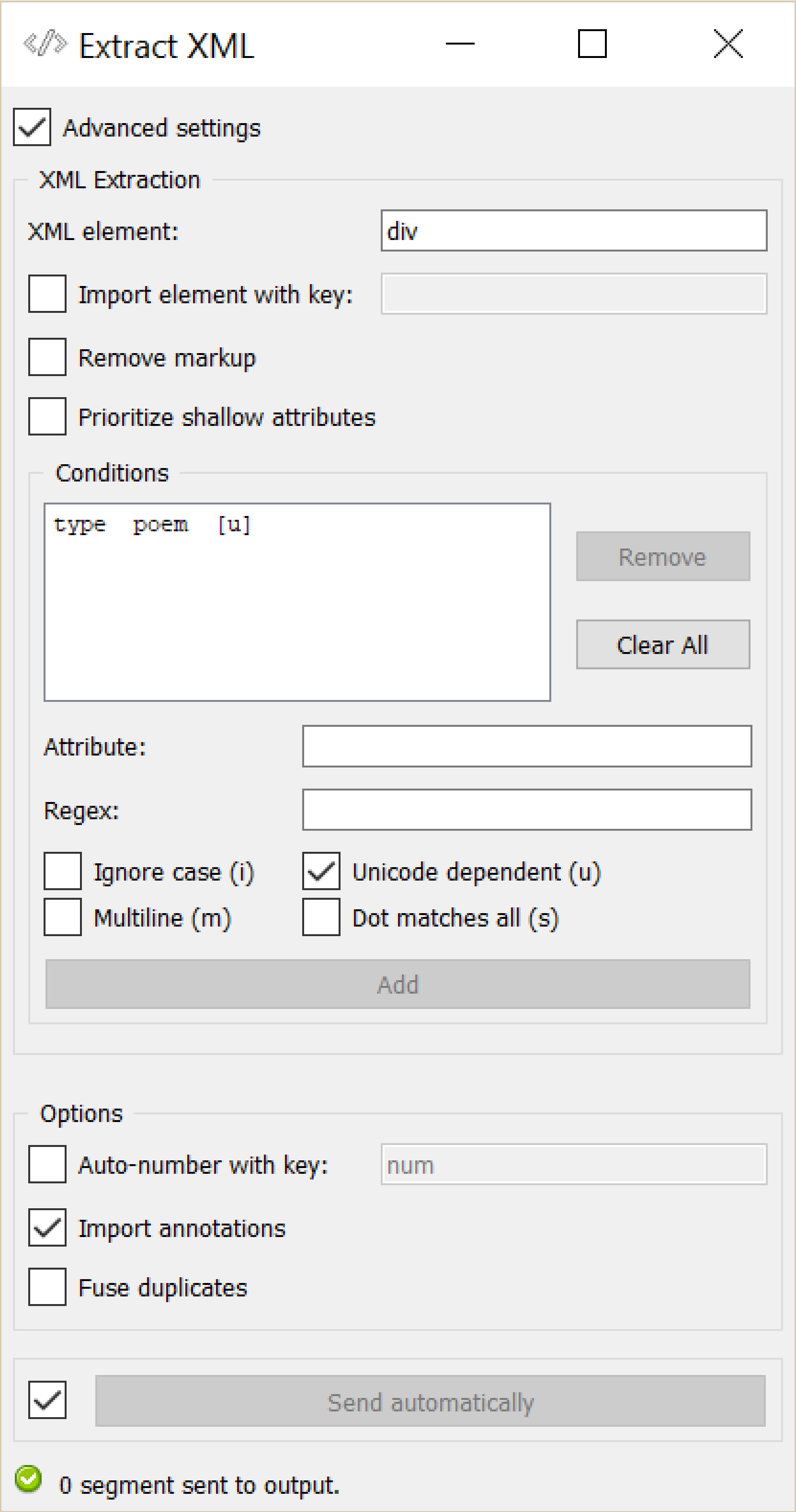
In such a case, the widget will first create two segments that have the exact same address (since the embedded XML tags are deleted with **Remove markup**), then by the effect of **Fuse duplicates**, it will seek to fuse them into one. It will only be able to keep one of the rival annotation values *A* and *B* for the annotation key *type*; by default, it will be the value associated to the element closest to the root in the XML tree, namely *A*. If on the other hand the **Prioritize shallow attributes** option is selected, the value of the element closest to the "surface" will be kept, in our example *B*.

The **Conditions** subsection included in the **XML Extraction** section allows the user to limit the extraction by specifying conditions bearing on attributes of the extracted elements. These conditions are expressed in the form of regular expressions that the given attribute values must match. In the list appearing at the top of this subsection, the columns indicate (a) the concerned attribute, (b) the corresponding regular expression, and (c) the options associated to this expression.[2]

In *figure 2* above), we have thus limited the extraction only to the <*div*> elements that have a type attribute whose value is *poem*. If several conditions were defined, they would all have to be fulfilled for an element to be extracted. The buttons on the right enable the user to delete the selected condition (**Remove**) or to empty the list completely (**Clear All**).

The remaining part of the **Conditions** subsection allows the user to add new conditions to the list. To do so, the attribute in question (**Attribute**) and the corresponding regular expression (**Regex**) must be specified. The **Ignore case (i)**, **Unicode dependent (u)**, **Multiline (m)** and **Dot matches all (s)** checkboxes manage the application of the

[2] See Python documentation.



Extract XML

☑ Advanced settings

XML Extraction

XML element: div

□ Import element with key:

□ Remove markup

□ Prioritize shallow attributes

Conditions

```
type   poem   [u]
```

Attribute:

Regex:

□ Ignore case (i)   ☑ Unicode dependent (u)

□ Multiline (m)   □ Dot matches all (s)

Options

□ Auto-number with key: num

☑ Import annotations

□ Fuse duplicates

☑ Send automatically

0 segment sent to output.

Fig. 96: Figure 2: **Extract XML** widget (advanced interface).

corresponding options to the regular expression. Adding the new condition to the list is finally carried out by clicking on the **Add** button.

Through the **Options** section the **Auto-number with key** checkbox enables the program to automatically number the segments of the output segmentation and to associate the number to the annotation key specified in the text field on the right. The **Import annotations** checkbox copies in each output segment every annotation associated to the corresponding segment of the input segmentation. The **Merge duplicate segments** checkbox enables the program to fuse distinct segments whose addresses are the same in a single segment; the annotations associated to the fused segments are copied in the single resulting segment.[3]

The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

Below the **Send** button, the user finds some indications such as the number of segments in the output segmentation, or the reasons why no segmentation is emitted (no input data, no output segment created, etc.).

Messages

Information

Data correctly sent to output: <n> segments. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* and *Errors* below).

Warnings

No XML element was specified. The name of an XML element must be entered in the **XML element** field in order for computation and data emission to proceed.

No label was provided. A label must be entered in the **Output segmentation label** field in order for computation and data emission to proceed.

No annotation key was provided for element import. In the advanced settings, the **Import element with key** checkbox has been selected and an annotation key must be specified in the text field on the right in order for computation and data emission to proceed.

No annotation key was provided for auto-numbering. The **Auto-number with key** checkbox has been selected and an annotation key must be specified in the text field on the right in order for computation and data emission to proceed.

[3] In the case where the fused segments have distinct values for the same annotation key, only the value of the last segment (in the order of the extracted segments before fusion) will be retained.

Errors

Regex error: <error_message> (condition #<n>). The regular expression in the *n*-th line of the **Conditions** list is invalid.

XML parsing error (missing closing tag / orphan closing tag). The input XML data couldn't be correctly parsed. Please use an XML validator to check the data's well-formedness.

Examples

- *Getting started: Converting XML markup to annotations*
- *Cookbook: Convert XML tags to Orange Textable annotations*

Footnotes

Display

Display or export the details of a segmentation.

Signals

Inputs:

- `Segmentation`

  Segmentation to be displayed or exported.

Outputs:

- `Bypassed segmentation` (default)

  Exact copy of the input segmentation

- `Displayed segmentation`

  Segmentation covering the entire string displayed in the widget's interface

Description

This widget inputs a segmentation and displays on screen the content and the annotation of the segments that compose it. The widget allows the user notably to export the information in a text file. Moreover, it forwards the segmentation without any modification on its output connections.[1]

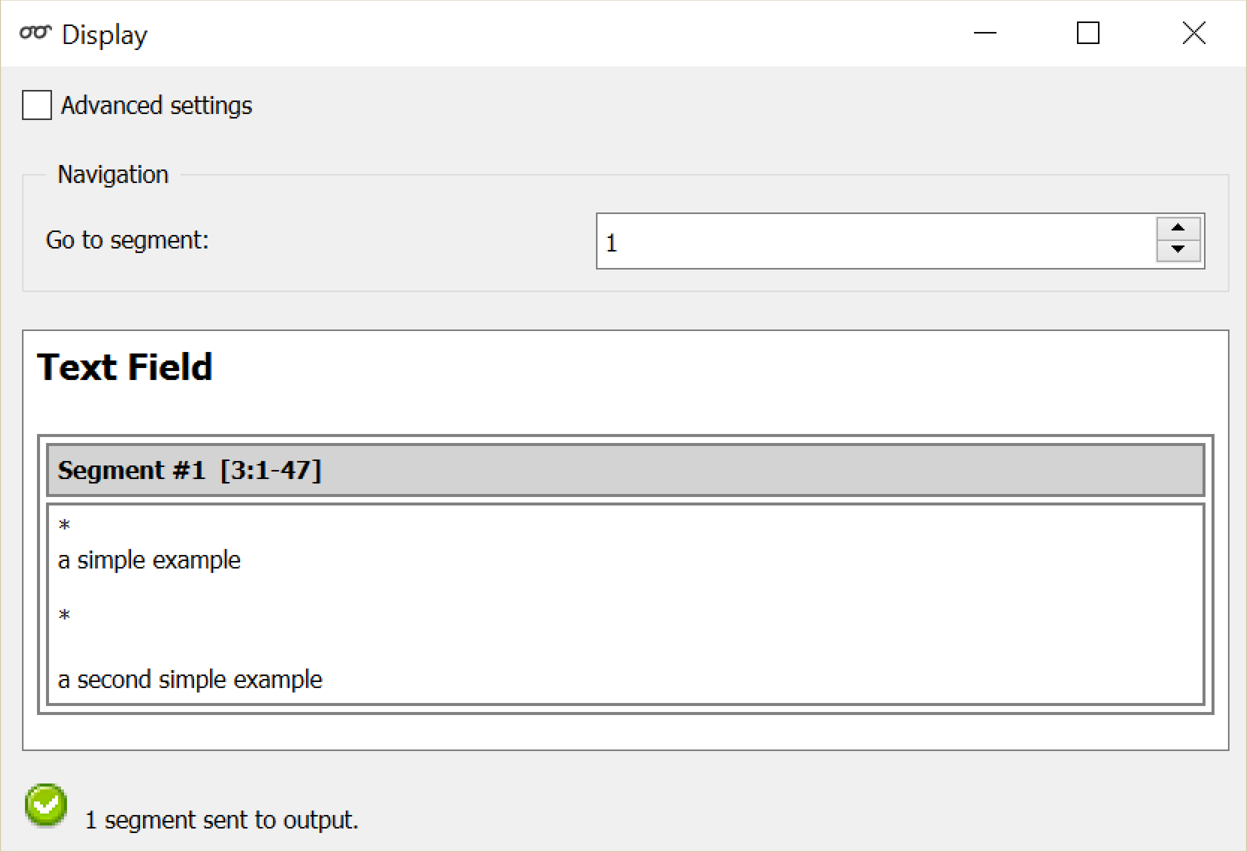
Display plays an essential role in schema construction: it is the best way to check that the configuration of the other segmentation processing widgets leads to the desired result in terms of segment and annotation creation or modification.

[1] The widget also sends, on a second channel not selected by default, a segmentation with a single segment containing the entire string as it is displayed in the widget's interface.

It should be noted that for long segmentations, the widget may appear stuck for a certain time after the progress bar has run – a problem related to the graphic interface library on which Orange Canvas relies. Unless memory overflow occurs, the problem normally solves itself after a few moments.

Basic interface

In its basic version, the widget formats the input segmentation in HTML and displays for each segment its number, its complete address (string index, start and end positions) as well as its annotations (see *figure 1*). The **Navigation** section enables the program to directly show a particular segment using **Go to segment**.



Text Field

Segment #1 [3:1-47]

*
a simple example

*

a second simple example

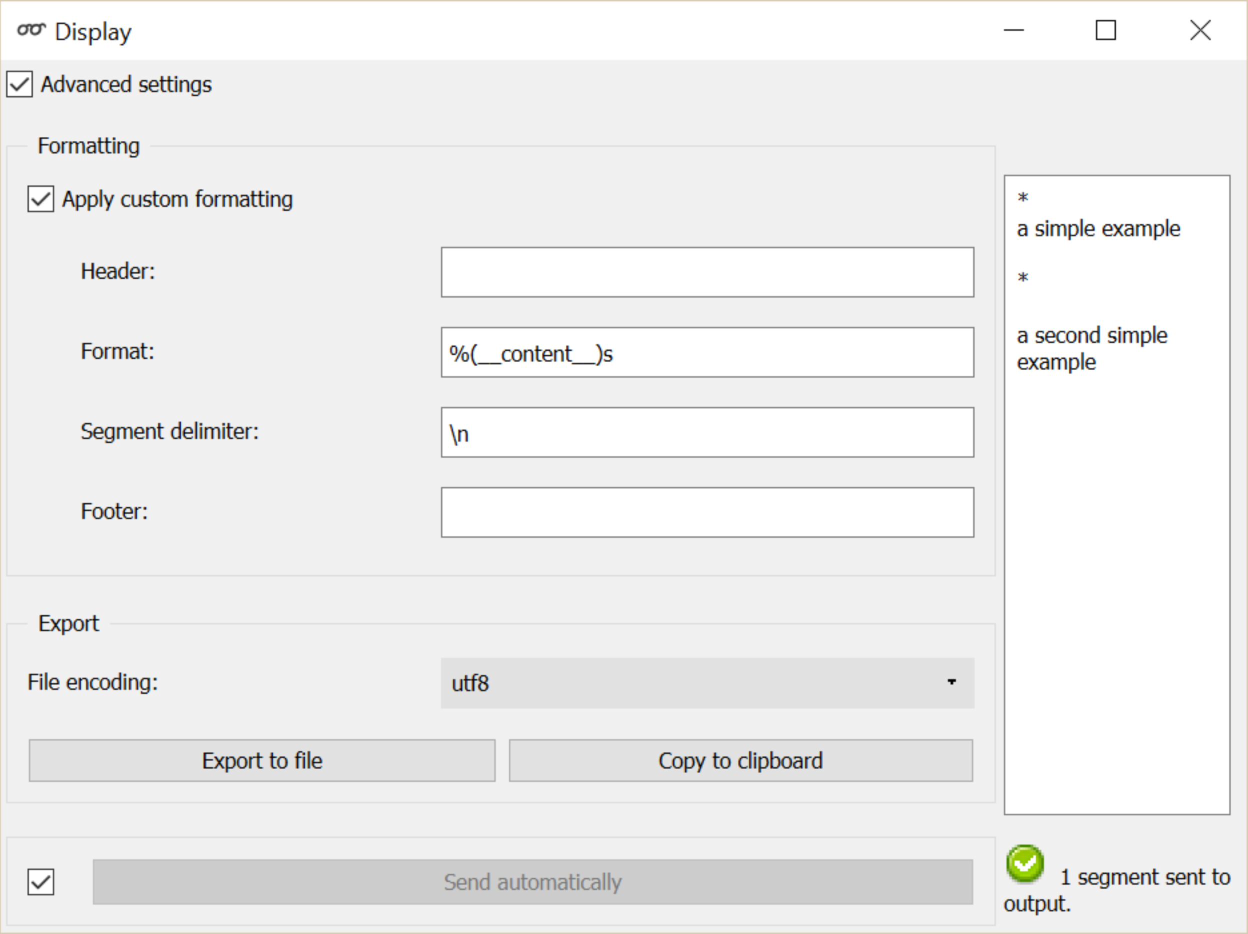
1 segment sent to output.

Fig. 97: Figure 1: **Display** widget (basic interface).

It can be noted that the basic interface of **Display** is more sober than those of the other widgets of Orange Textable: it does not include a **Send** button nor a **Send automatically** checkbox. What motivates this design is the will to emphasize the fundamental functionality of visualization of the input segmentation content and annotations – main reason for the use of **Display** in most cases. In this context, by default, data are automatically sent on output connections.

Advanced interface

The widget's advanced interface (see *figure 2*) restores informative indications such as the number of segments in the input segmentation or the reasons why no segmentation is emitted (for example no input data) below the Text data window. The **Send** button triggers the emission of a segmentation to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).



☑ Advanced settings

Formatting

☑ Apply custom formatting

Header:

Format: %(__content__)s

Segment delimiter: \n

Footer:

Export

File encoding: utf8

☑

```
*
a simple example
*

a second simple
example
```

1 segment sent to output.

Fig. 98: Figure 2: **Display** widget (advanced interface).

The **Apply custom formatting** button enables the program to produce a personalized rendering. In this mode, the formatting of each segment is determined by a string entered in the **Format** field. This string can contain text that will be reproduced as it is in the rendered output, as well as references to variables to insert in the output. These references take the following general form:

> *%(variable_name)format*

where *variable_name* designates the variable to insert and *format* the desired format for this variable. For a basic use, all you need is to know that the format code *s* designates a character string and *i* an integer.[2] If the name of the variable is one of the following predefined strings, it will be interpreted as indicated in the right column:[3]

variable name	meaning
__content__	segment content
__num__	segment number
__str_index__	string index
__str_index_raw__	string index counting from 0
__start__	initial position
__start_raw__	initial position counting from 0
__end__	final position

If on the contrary the name of the variable is not among those of the list, the program will interpret it as an annotation key and will attempt to display the corresponding value (or the string *__none__* if this key is not defined for the considered segment).

The string entered in the **Segment delimiter** field, if any, will be inserted between each segment of the formatted segmentation. Use the sequence *n* for a line break and *t* for tabulation.

The **Header** and **Footer** fields enable the user to specify strings that will be inserted respectively at the beginning and the end of the formatted segmentation.

To take a simple example, consider the following (extract of a) segmentation of the string *a simple example*[4]:

content	start	end	*letter category*
a	1	1	*vowel*
s	3	3	*consonant*
i	4	4	*vowel*
…	…	…	…
e	16	16	*vowel*

By entering:

- `<word>\n` in the **header** field,
- `<letter pos="%(__num__)i" type="%(letter category)s">%(__content__)s</letter>` in the **format** field,
- `\n` in the **segment delimiter** field, and
- `\n</word>` in the **footer** field,

we obtain the following formatting:

[2] For more details on the syntax of format codes, see Python documentation.

[3] In general, predefined strings in Orange Textable have in common that they begin and end by two *underscore* characters (_); it is greatly recommended to avoid this form for every name supplied by the user (in particular for the segmentation labels, as well as for the keys and annotation values).

[4] By convention, we do not indicate here the string index associated with each segment but only its start and end positions, along with the annotation values associated with it; moreover, for the sake of readability, we do indicate the content of each segment, though it is not formally part of the segmentation (but rather of the string to which the segmentation refers).

```
<word>
<letter pos="1" type="vowel">a</letter>
<letter pos="2" type="consonant">s</letter>
<letter pos="3" type="vowel">i</letter>
<letter pos="4" type="consonant">m</letter>
<letter pos="5" type="consonant">p</letter>
<letter pos="6" type="consonant">l</letter>
<letter pos="7" type="vowel">e</letter>
<letter pos="8" type="vowel">e</letter>
<letter pos="9" type="consonant">x</letter>
<letter pos="10" type="vowel">a</letter>
<letter pos="11" type="consonant">m</letter>
<letter pos="12" type="consonant">p</letter>
<letter pos="13" type="consonant">l</letter>
<letter pos="14" type="vowel">e</letter>
</word>
```

The **Export** section of the widget interface also allows the user to export the displayed segmentation (standard HTML or user-defined format) in a file. The encoding can be selected (**Encoding**) then click on **Export** to open a file selection dialog. By clicking the **Copy to clipboard** button, the user may also to copy the displayed segmentation in order to paste it in another application for instance; in this case, the utf-8 encoding is used by default.

When the option **Apply custom formatting** is not selected, the **Navigation** section is enabled and allows the user to view a particular segment through the **Go to segment** control.

Messages

Information

Data correctly sent to output: <n> segments. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to 'Displayed segmentation' channel, see 'Widget state' below. A problem with the 'Format' parameter prevents this widget instance from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see and *Errors* below).

Errors

Format mismatch error: a <variable_type> is required. In the advanced interface, the string entered in the **Format** field indicates that a variable of a certain type (e.g. float) is expected, but in at least one case, the corresponding value is of another type (e.g. string). The string type (e.g. `%(__content__)s`) is usually the safest bet.

Format mismatch error: not enough arguments for format string. In the advanced interface, the string entered in the **Format** field indicates that a variable is expected but in at least one case, there is no corresponding value. Make sure that no placeholder is used without an explicit name (always use e.g. `%(__content__)s`, and never `%s`).

Format error: missing variable type. In the advanced interface, a variable type indication is missing in the string entered in the **Format** field. Make sure that no placeholder is used without a variable type indication (always use e.g. `%(__content__)s`, and never `%(__content__))`.

Format error: missing name. In the advanced interface, a variable name is missing in the string entered in the **Format** field. Make sure that no placeholder is used without a variable name (always use e.g. `%(__content__)s`, and never `%()s`).

Examples

- *Getting started: Keyboard input and segmentation display*
- *Cookbook: Display text content*
- *Cookbook: Export text content (and/or change text encoding)*

Footnotes

1.6.3 Table construction widgets

Widgets of this category take *Segmentation* data in input and emit tabular data in the internal format of Orange Textable. 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*). Widget *Cooccurrence* makes it possible to measure the tendency of text units to occur in the same contexts, while *Context* serves to build concordances and collocation lists. Finally, *Category* exploits categorical information associated with segmentations.

Count

Count segment types.

Signals

Inputs:

- `Segmentation` (multiple)

  Segmentation whose segments constitute the units to be counted or the contexts in which the units will be counted

Outputs:

- `Pivot Crosstab`

  Table displaying the absolute frequency of units

Description

This widget inputs one or several segmentations, counts the frequency of segments defined by one of the segmentations (potentially within segments defined by another), and sends the result in the form of a *contingency table* (or *co-occurrence matrix* or also *term–document matrix*).

The contingency tables produced by this widget are of *PivotCrosstab* type, a subtype of the generic *Table* format (see *Convert* widget, section *Table formats*). In such a table, each column corresponds to a *unit* type, each line corresponds to a *context* type, and the cell at the intersection of a given column and line contains the count (or *absolute frequency*, or also number of occurrences) of this unit type in this context type.

To take a simple example, consider two segmentations of the string *a simple example*[1]:

A) label = *words*

content	start	end	*part of speech*	*word category*
a	1	1	*article*	*grammatical*
simple	3	8	*adjective*	*lexical*
example	10	16	*noun*	*lexical*

B) label = *letters* (extract)

content	start	end	*letter category*
a	1	1	*vowel*
s	3	3	*consonant*
i	4	4	*vowel*
...	...	...	...
e	16	16	*vowel*

Typically, we could define unit types based on the content of the segments of the *letters* segmentations, and context types based on the content of the segments of the *words* segmentations. Counting these unit types in these contexts types would thus produce the following contingency table[2]:

__context__	*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

Alternatively, we could rather count the *annotation values* (instead of the content) of the units and/or of the contexts. For example, by defining units on the basis of the annotations associated to the key *letter category* in the *letters* segmentation, and contexts on the basis of the annotations associated to the key *word category* in the *words* segmentation, we would obtain the following table:

__context__	*vowel*	*consonant*
grammatical	1	0
lexical	5	8

This way of selecting segmentations and annotation keys constitutes an extremely flexible mechanism which enables the user to easily produce a variety of contingency tables. Note that it is up to the user to provide a coherent definition of the units and contexts. In general, a given unit is considered to occur in a given context if, (a) the segment corresponding to the unit and the context are both be associated to the same string, (b) the initial position of the unit segment in the string is higher or equal to that of the context segment, and (c) conversely the final position of the unit is lower or equal to that of the context. In short, the unit must be *contained* within the context.

[1] By convention, we do not indicate here the string index associated with each segment but only its start and end positions, along with the various annotation values associated with it; moreover, for the sake of readability, we do indicate the content of each segment, though it is not formally part of the segmentation (but rather of the string to which the segmentation refers).

[2] The first column header, *__context__*, is a name predefined by Orange Textable.

A borderline case made possible by this *modus operandi* consists of defining units and contexts on the basis of the same segmentation. Indeed since every segment is contained in itself, nothing keeps us from using a single segmentation, *words* for example, and defining units with the key *part of speech* and contexts with the key *word category*:

__context__	*article*	*noun*	*verb*
grammatical	1	0	0
lexical	1	1	0

Orange Textable offers two other ways to define contexts while still using a single segmentation. The first relies on the notion of a "window" of *n* segments that we progressively "slide" from the beginning to the end of the segmentation. In our example, by applying this principle to the *letters* segmentation and by setting the window size to 11 segments, we thus define the following contexts:

1. *a simple exam*
2. *simple examp*
3. *imple exampl*
4. *mple example*

By otherwise defining the units based on the *letter category* annotations for example, we thus obtain the following counts (where the contexts are represented by their successive positions):

__context__	*vowel*	*consonant*
1	5	6
2	4	7
3	4	7
4	4	7

The last context specification mode that Count offers and which involves a single segmentation consists of defining the contexts as *n* segments immediately to the left and/or to the right of each segment. For example, based on the *letter category* annotations of segmentation *letters*, defining the contexts as the two segments immediately on the left and on the right of the segment results in the following contingency table (where the '+' symbol separates the successive segments of the context and the underscore symbol '_' separates the left and right parts of the context):

__context__	*consonant*	*vowel*
vowel+consonant_consonant	2	2
consonant+vowel_consonant	2	1
consonant+consonant_vowel	2	1
vowel+vowel_vowel	1	0

Such a table notably indicates that in a context composed, on the left, of a *vowel+consonant* sequence and, on the right, of a consonant (for example *ex_m* or *am_l*), we have twice observed a vowel and thrice a consonant. A particular case of this type of table is that of the *transition matrix* that defines a *Markov chain*, where we only consider the context on the left of the segments:

__context__	*vowel*	*consonant*
vowel	o	5
consonant	5	4

Let us also note that context specification, unlike unit specification, is optional. Indeed, it is always possible to globally count the frequency of segmentation units and thus produce a table that only contains a single row corresponding to the whole concerned segmentation (thus *letters*, in the following example):

__context__	a	s	i	m	p	l	e	x
__global__	2	1	1	2	2	2	3	1

Finally, in every scenario considered here, we could also take an interest for the frequency of the sequences from 2, 3, . . . , *n* segments (or *n–grams*) rather that to the frequency of isolated segments:

__context__	as	si	im	mp	pl	le	ex	xa	am
__global__	1	1	1	2	2	2	1	1	1

After having thus outlined the range of contingency table types that the **Count** widget can produce, we can take a look at its interface (see figures *1* to *4*). It contains two separate sections for unit definition (**Units**) and context definition (**Contexts**).

In the **Units** section, the **Segmentation** drop-down menu allows the user to select among the input segmentations the one whose segment types will be counted. The **Annotation key** menu displays the annotation keys associated to the chosen segmentation, if any; if one of the keys is selected, the corresponding annotation values will be counted; if on the other hand the value *(none)* is selected, the *content* of the segments will be counted. The **Sequence length** drop-down menu allows the user to indicate if isolated segments or segment *n–grams* should be counted; in this latter case, the (optional) string specified in the **Intra sequence delimiter** text field will be used to separate the content or the annotation value corresponding to each segment in the column headers.[3]

The **Contexts** section is available in several variants, depending on the selected value in the **Mode** drop-down menu. The latter allows the user to choose between the different ways of defining contexts described earlier. The **No context** mode (see *figure 1*) corresponds to the case where units are counted globally in the whole segmentation specified in the **Units** section (to which we will refer by the term *unit segmentation*).

The **Sliding window** mode (see *figure 2*) implements the notion of a "sliding window" introduced earlier. Typically it allows the user to observe the evolution of frequency throughout the unit segmentation. The only parameter is the window size (in number of segments), defined by the **Window size** cursor.

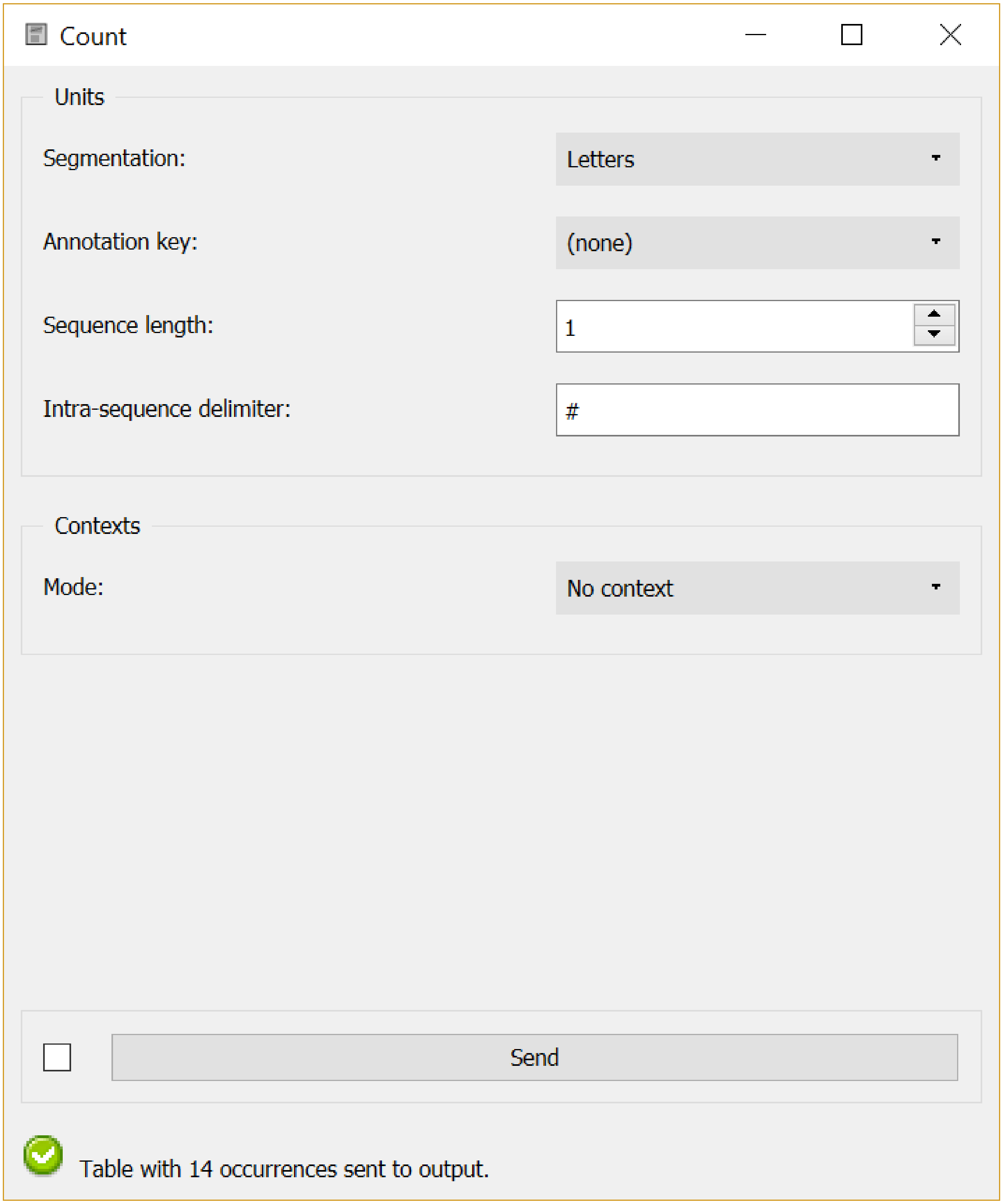
The **Left–right neighborhood** mode (see *figure 3*) allows the user to specify context types based on the *n* segments immediately to the left and/or right of each segment; this mode notably allows the user to build a Markov chain transition matrix. The **Left context size** and **Right context size** parameters determine the number of segments taken into consideration in each part of the context. The **Unit position marker** text field allows the user to specify the (possibly empty) character chain to insert in-between the left and right parts of the context in the row headers. The checkbox (**Treat distinct strings as contiguous**) enables the user to to choose if separate strings should be treated as if they were actually contiguous, so that the end of each string is adjacent to the beginning of the next string.

Finally, the **Containing segmentation** mode (see *figure 4*) corresponds to the case where contexts are defined by the segment types that appear in a segmentation (which can be that of the units or another). This segmentation, that we will call *context segmentation* by analogy, is selected among the input segmentations by means of the **Segmentation** drop-down menu. The **Annotation key** menu displays the annotation keys associated with the context segmentation, if any; if one of the keys is selected, the corresponding annotation value types will constitute the row headers; if however the value *(none)* is selected, the *content* of the segments will be exploited. The **Merge** contexts checkbox enables the program to globally count the units in the whole context segmentation.

Below the **Send button**, the user finds indications such as the sum of frequencies in the output table, or the reasons why not table is emitted (no input data or total frequency is zero).

The **Compute** button triggers the emission of a table in the internal format of Orange Textable, to the output connection(s). When it is selected, the **Compute automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

[3] The same character string will be inserted between the successive segments that build up the left and/or right context if the **Left–right neighborhood** mode is selected.



Segmentation: Letters

Annotation key: (none)

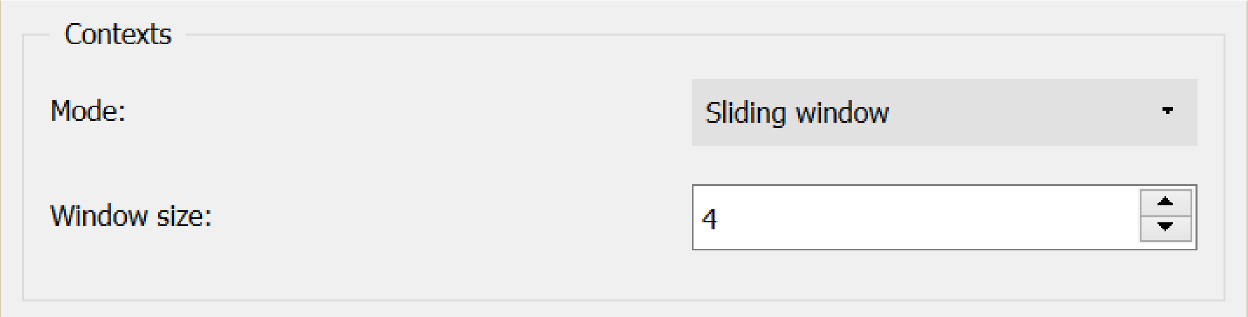
Sequence length: 1

Intra-sequence delimiter: #

Mode: No context

Table with 14 occurrences sent to output.

Fig. 99: Figure 1: **Count** widget (**No context** mode).

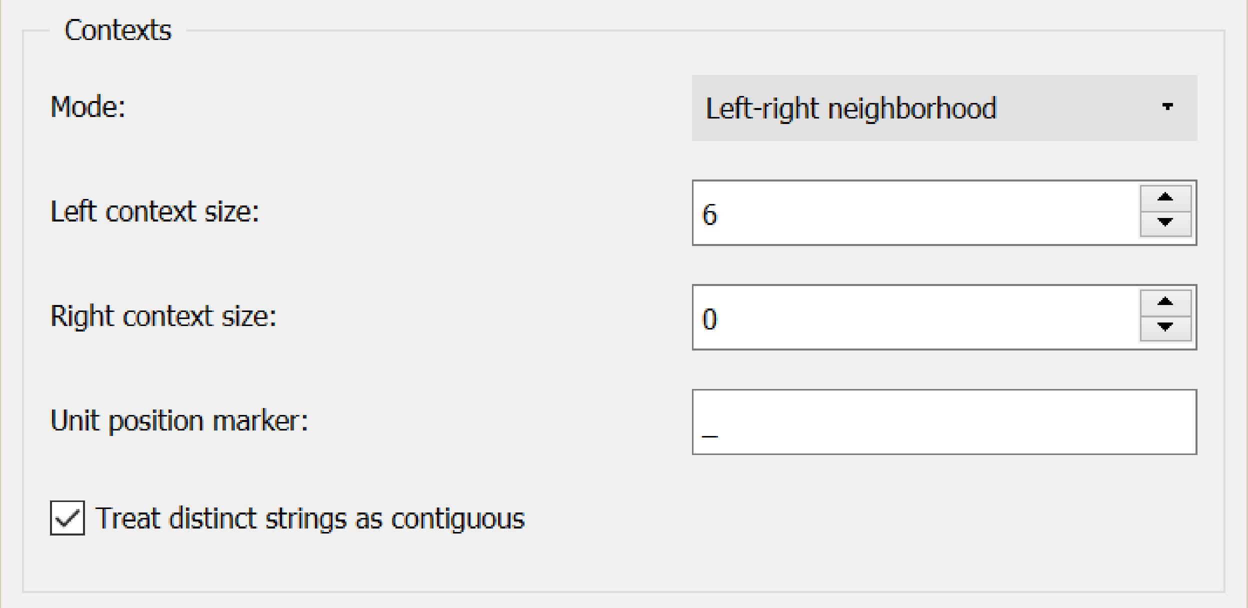


Contexts

Mode: Sliding window

Window size: 4

Fig. 100: Figure 2: **Count** widget (**Sliding window** mode).



Contexts

Mode: Left-right neighborhood

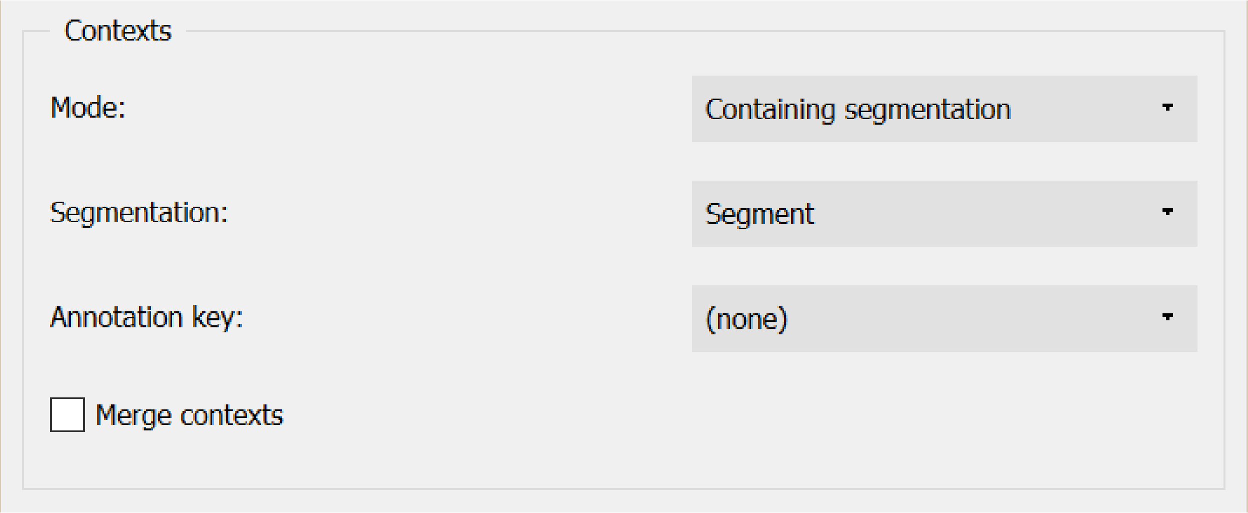
Left context size: 6

Right context size: 0

Unit position marker: _

☑ Treat distinct strings as contiguous

Fig. 101: Figure 3: **Count** widget (**Left–right neighborhood** mode).



Contexts

Mode: Containing segmentation

Segmentation: Segment

Annotation key: (none)

□ Merge contexts

Fig. 102: Figure 4: **Count** widget (**Containing segmentation** mode).

Messages

Information

Data correctly sent to output: total count is <n>. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Compute' when ready. Settings and/or input have changed but the **Compute automatically** checkbox has not been selected, so the user is prompted to click the **Compute** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* below).

Warnings

Resulting table is empty. No table has been emitted because the widget instance couldn't find a single element in its input segmentation(s). A likely cause for this problem (when using the **Containing segmentation** mode) is that the unit and context segmentations do not refer to the same strings, so that the units are in effect *not* contained in the contexts. This is typically a consequence of the improper use of widgets *Preprocess* and/or *Recode* (see *Caveat*).

Examples

- *Getting started: Counting segment types*
- *Getting started: Counting in specific contexts*
- *Cookbook: Count unit frequency*
- *Cookbook: Count occurrences of smaller units in larger segments*
- *Cookbook: Count transition frequency between adjacent units*
- *Cookbook: Examine the evolution of unit frequency along the text*

See also

- *Reference: Convert widget (section "Table formats")*

Footnotes

Length

Compute the (average) length of segments.

Signals

Inputs:

- `Segmentation` (multiple)

  Segmentation whose segments constitute the units of length measurement, the contexts whose lengths will be measured, or the units over which length will be averaged

Outputs:

- `Textable table`

  Table in the internal format of Orange Textable

Description

This widget inputs one or several segmentation, measures the length of one (eventually within the segments defined by another segmentation), and sends the results in table format. It also allows the user to calculate the average length of segments of a segmentation based on the units defined by another segmentation.

The tables produced by the **Length** widget have at least 2 columns, and at most 4. The first column contains the headers corresponding to the contexts – which are essentially defined in the same way as in the *Count* widget. The second column gives the length indications (in which case the header is *__length__*) or the average length (header *__length_average__*). In the latter case, the third column may then contain the corresponding standard deviations if their display is required by the user (header *__length_std_deviation__*), and the last column will indicate the number of elements on which the average calculation is done (header *__length_count__*).

To take a simple example, consider two segmentations of the string *a simple example*[1]:

A) label = *words*

content	start	end	*part of speech*	*word category*
a	1	1	*article*	*grammatical*
simple	3	8	*adjective*	*lexical*
example	10	16	*noun*	*lexical*

B) label = *letters* (extract)

content	start	end	*letter category*
a	1	1	*vowel*
s	3	3	*consonant*
i	4	4	*vowel*
…	…	…	…
e	16	16	*vowel*

Essentially here two basic configurations are considered. The first is when we are simply interested in the length of a given segmentation, for example *letters*:

__context__	*__length__*
__global__	14

[1] By convention, we do not indicate here the string index associated with each segment but only its start and end positions, along with the various annotation values associated with it; moreover, for the sake of readability, we do indicate the content of each segment, though it is not formally part of the segmentation (but rather of the string to which the segmentation refers).

In what follows, we will designate with the terms *units of measurement* the segments whose count is interpreted as a length measure, namely in this example the segments of the segmentation *letters*.

The second basic configuration is when we wish to know the *average* length of the segments of a segmentation, for example *words*, in terms of measure units belonging to another segmentation (here *letters*):

__context__	__length_average__	__length_std_deviation__	__length_count__
__global__	4.66666650772	2.62466931343	3

In this case, we will name *averaging units* the segments whose lengths are measured and averaged. Note that the average length calculation presupposes that at least one measure unit is *contained* within the averaging unit, in the sense that the following three conditions are met: (a) the segment corresponding to the unit and the context are both be associated to the same string, (b) the initial position of the unit segment in the string is higher or equal to that of the context segment, and (c) conversely the final position of the unit is lower or equal to that of the context.

These two elementary configurations (length measurement and average length calculation) can then be combined with two ways of specifying contexts – i.e. two ways of defining table rows. The first mode consists of defining the contexts based on the content or the annotations of a given segmentation; for example, here is the length of the *words* segments (contexts) in terms of those of *letters* (units of measurement):

__context__	__length__
a	1
simple	6
example	7

It should be noted that the segment *types* define the row headers, as illustrated in the following example, where the same segmentations are used but the contexts are defined by the annotation values associated with the key *word type*:

__context__	__length__
grammatical	1
lexical	13

The average length calculation is also applicable when the contexts are defined on the basis of a segmentation. In this case, we will generally use three different segmentations to define the units of measurement, the averaging units, and the contexts; for example, it could be to calculate the average length of words (in number of letters) in different texts. To stay in the frame of our example based on only two segmentations, we can exploit the fact that all segments are contained in themselves and calculate the average length of words (in number of letters) depending on the *word types* annotations (in other words we here use a single segmentation to determine the contexts and the averaging units):

__context__	__length_average__	__length_std_deviation__	__length_count__
grammatical	1	0	1
lexical	6.5	0.5	2

The second context specification mode lies on the concept of a “window” of *n* segments that we progressively slide from the beginning to the end of the segmentation. For example, by setting the window size to 2 segments, we can examine the average length of words (in number of letters) in successive bigrams of the *words* segmentation (identified by their position):

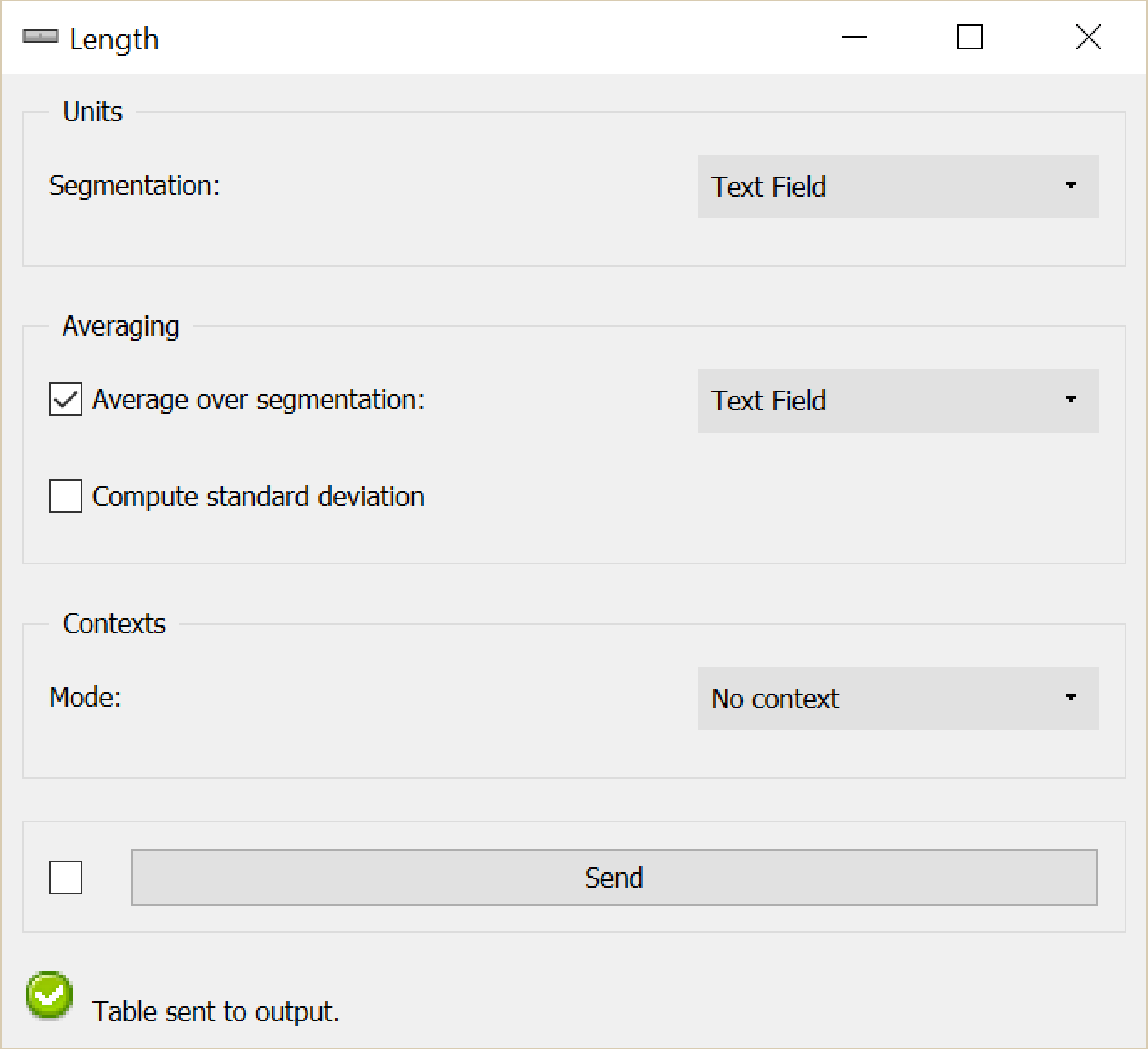
__context__	__length_average__	__length_std_deviation__	__length_count__
1	3.5	2.5	2
2	6.5	0.5	2

By construction, each cell of the column *__length_count__* will then contain the same value, or the window size. Based on this observation, it is rather easy to convince oneself that this latter context specification mode only makes sense when we are interested in the evolution of an *average* length throughout a segmentation.

We now move on to the presentation of the widget interface (see *figure 1*). It contains three separate sections for the specification of the units of measurement (**Units**), of the averaging units (**Averaging**), and of the contexts (**Contexts**).

The **Units** section only contains a single drop-down menu (**Segmentation**) used to select among the input segmentation the one whose segments will provide the units of measurement.

In the **Averaging** section, the **Average over segmentation** checkbox triggers the calculation of the average length. The drop-down menu on the right allows the user to select the segmentation whose segments will constitute the averaging units. The **Compute standard deviation** checkbox allows the user to calculate, other than the average length, its standard deviation. It should be noted that for large segmentations, this option is likely to spectacularly extend the calculation time.



Segmentation: Text Field

☑ Average over segmentation: Text Field

□ Compute standard deviation

Mode: No context

□ Send

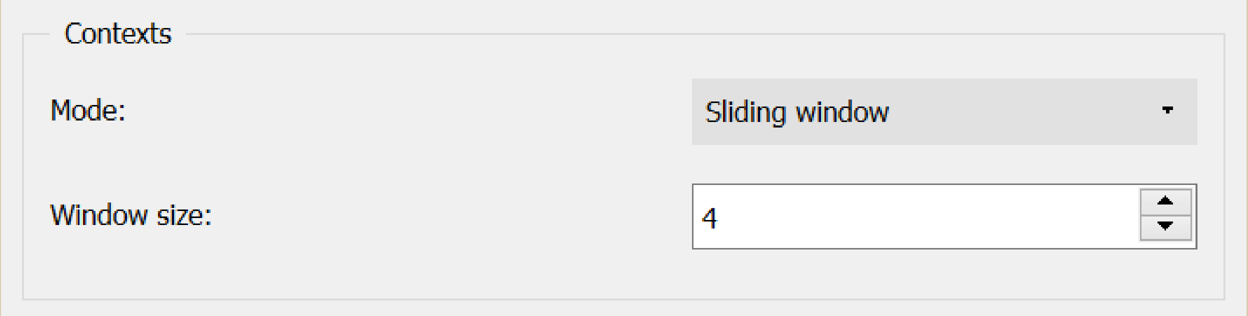
Table sent to output.

Fig. 103: Figure 1: **Length** widget (**No context** mode).

The **Contexts** section is available in several variants depending on the value selected in the **Mode** drop-down menu.

This latter option allows the user to choose among the context specification modes described above. The **No context** mode corresponds to the case where the length measurement or the average length calculation are globally applied to the entire segmentation that defines the units of measurement (specified in the **Units** section).

The **Sliding window** mode (*figure 2*) implements the notion of a "sliding window" introduced above. It allows the user to observe the evolution of the average length throughout the averaging unit segmentation. The only parameter is the size of the window (in number of segments), set by means of the **Window size** cursor.

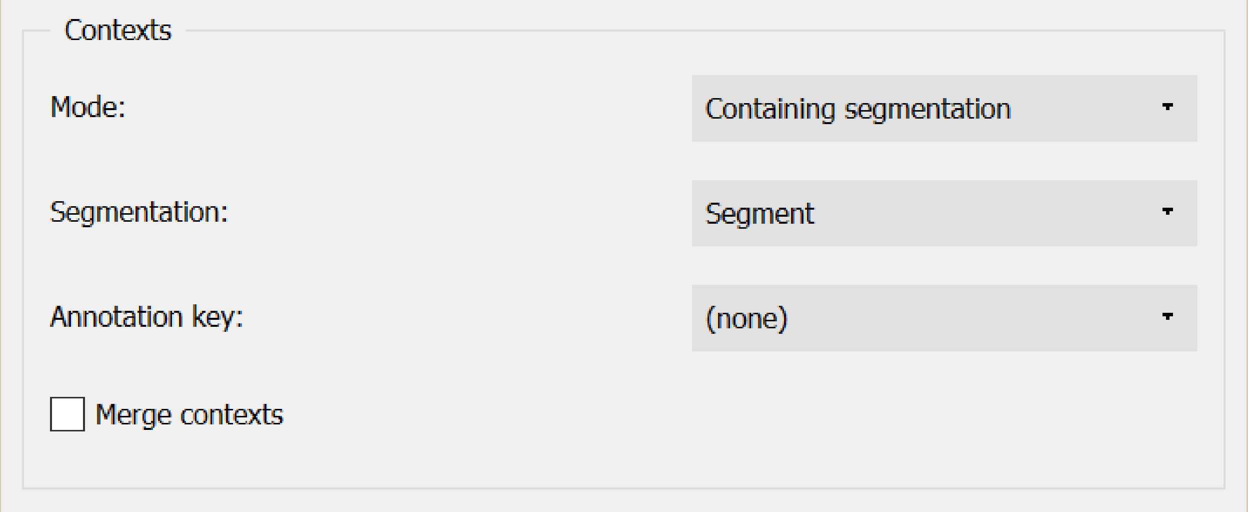


Contexts

Mode: Sliding window

Window size: 4

Fig. 104: Figure 2: **Length** widget (**Sliding window** mode).



Contexts

Mode: Containing segmentation

Segmentation: Segment

Annotation key: (none)

□ Merge contexts

Fig. 105: Figure 3: **Length** widget (**Containing segmentation** mode).

Finally, the **Containing segmentation** mode (see *figure 3*) corresponds to the case where the contexts are defined by the segment types appearing in a segmentation (that will most often be distinct from the segmentation providing the units of measurement and the averaging units). This segmentation is selected among the input segmentation by means of the **Segmentation** drop-down menu. The **Annotation key** menu shows the possible annotation keys associated to the selected segmentation; if one of these keys is selected, the corresponding types of annotation values will constitute the row headers; if on the other hand the value *(none)* is selected, the *content* of the segments will be used. The **Merge contexts** checkbox allows the user to measure the length or to calculate the average length globally in the entire segmentation that defined the contexts.

The **Send** button triggers the emission of a table in the internal format of Orange Textable to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

The informations below the **Send** button indicate if a table has been correctly emitted, or the reasons why no table is emitted (no input data).

Messages

Information

Data correctly sent to output. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* below).

Warnings

Resulting table is empty. No table has been emitted because the widget instance couldn't find a single element in its input segmentation(s). A likely cause for this problem (when using the **Containing segmentation** mode) is that the unit and context segmentations do not refer to the same strings, so that the units are in effect *not* contained in the contexts. This is typically a consequence of the improper use of widgets *Preprocess* and/or *Recode* (see *Caveat*).

Footnotes

Variety

Measure the variety of segments.

Signals

Inputs:

- `Segmentation`

  Segmentation whose segments constitute the units of variety measurement, or the contexts in which variety will be measured

Outputs:

- `Textable table`

  Table in the internal format of Orange Textable

Description

This widget inputs one or several segmentations, measures the variety of the segments of one of the segmentations (eventually within the segments defined by another segmentation), and sends the result in table format; it also allows the user to calculate the average variety by category (based on the annotation values of the segments). In order to make these two measures less dependent on the length of segmentations, it is possible to calculate their average value on a number of subsamples of fixed size.

The tables produced by the **Variety** widget have at least 2 columns, and at most 4. The first column contains the headers corresponding to the contexts – which are essentially defined in the same way as in the *Count* and *Length* widgets. The second column gives the variety measures and its header is __variety__, unless resampling has been applied (in which case the header will be __variety_average__). In the latter case, the third column will contain the corresponding standard deviation (header __variety_std_deviation__) and the last column the number of subsamples (header __variety_count__).

To take a simple example, consider two segmentations of the string *a simple example*[1]:

A) label = *words*

content	start	end	*part of speech*	*word category*
a	1	1	*article*	*grammatical*
simple	3	8	*adjective*	*lexical*
example	10	16	*noun*	*lexical*

B) label = *letters* (extract)

content	start	end	*letter category*
a	1	1	*vowel*
s	3	3	*consonant*
i	4	4	*vowel*
…	…	…	…
e	16	16	*vowel*

The most elementary measure made by the widget is that of the number of types or *variety*. For example, for the segmentation *letters*, by defining the units based on the content of the segments:

__context__	__variety__
__global__	8

Naturally, it is possible to define types based on the values associated to an annotation key, for example *letter category*:

__context__	__variety__
__global__	2

It is also possible to *weigh* the variety according to the frequency of types. To do this, we can calculate the *perplexity* of the segment distribution, that is to say the exponential of the entropy on this distribution. This measure is equal to the variety only if the segment types have a uniform frequency; it decreases and tends towards 0 as the segment distribution departs from uniformity and gradually becomes deterministic. As an example, here is the perplexity for *letter category*:

[1] By convention, we do not indicate here the string index associated with each segment but only its start and end positions, along with the various annotation values associated with it; moreover, for the sake of readability, we do indicate the content of each segment, though it is not formally part of the segmentation (but rather of the string to which the segmentation refers).

__context__	*__variety__*
__global__	1.97962633005

The difference observed between the variety with or without weighing (1.96 vs 2) shows the deviation from uniformity in the distribution of letter categories in this example.

Rather than looking at the variety (weighed or not) of the segment types *in general*, we can look at their average variety within a *category*. For example, we can ask what is the average variety of letters *depending on the letter category*:

__context__	*__variety__*
__global__	4.0

On average, in our example, a type of letter (*consonant or vowel*) is thus represented by 4.0 distinct letters – as long as we give the same weight to each category. The alternative consists of weighing the categories according to their frequency, which would result in our case in giving more weight to the variety of consonants (whose frequency is 9) than to that of the vowels (whose frequency is 6) in our average calculation:

__context__	*__variety__*
__global__	4.14285714286

From the increase observed compared to the case where the categories are not weighed, we can deduce that the number of distinct consonants is higher than that of the vowels.

To sum up, weighing (or not) the frequencies of units is the basis of the distinction between variety and perplexity; moreover, in the case where we calculate the average variety/perplexity per category, it is possible to weigh (or not) by the frequency of categories.

The different variety measures presented above can then be combined with the same *context* (i.e. table rows) specification modes as in the *Length* widget: the first mode consists in defining the contexts based on the content or the annotations of a given segmentation; the second lies on the concept of a “window” of *n* segments that we progressively “slide” from the beginning to the end of the segmentation.

All variety measures (weighed or not, simple or by category) are sensitive to the sample size, which in our case means the segmentation length. As such, they are in principle not directly comparable among/between of different lengths. Consider for example the (unweighted) variety of *letters* (units) in *words* (contexts):

__context__	*__variety__*
a	1
simple	6
example	6

To reduce the effect of this dependence to the segmentation length, it is possible to adopt the following strategy: draw a set number of subsamples in each segmentation to compare and report the average variety by subsample. For example, by setting the size of the subsamples to 2 segments, and by drawing 100 subsamples for each word, we obtain the following results:[2]

__context__	*__variety_average__*	*__variety_std_deviation__*	*__variety_count__*
a	—	—	—
simple	1.59	0.491833305094	100
example	1.52	0.499599839872	100

[2] The example has an instructive purpose; in practice we will typically use a clearly higher subsample size, for example 50 segments or more.

Here, we can see that the variety average in *simple* is very slightly higher than in *example* because *simple* is a shorter word and has no repeating letters. Moreover, since the article *a* is only one letter, our operation cannot build subsamples of 2 letters to compute and report their average variety, hence the missing values for variety average, standard deviation and count.

We now move on to the presentation of the widget interface (see *figure 1*). It has four separate sections, for unit specification (**Units**), category specification (**Categories**), context specification (**Contexts**), and resampling parameters (**Resampling**).

In the **Units** section, the **Segmentation** drop-down menu allows the user to select among the input segmentations the one whose segments will be the basis of the variety calculation. The **Annotation key** menu shows the possible annotation keys associated to the chosen segmentation; if one of these keys is selected, the corresponding annotation values will be used; if on the other hand the value *(none)* is selected, the content of the segments will be used. The **Sequence length** drop-down menu allows the user to indicate if the widget should consider the isolated segments or the *n–grams*. Finally, the **Weigh by frequency** checkbox allows the user to enable the weighing of the units by their frequency (thus the perplexity measure rather than the variety). Checking the **Dynamically adjust subsample size** box permits a more robust variety estimation. This calculation uses the RMSP subsample size adjustment method described in Xanthos and Guex 2015.

In the **Categories** section, the **Measure diversity per category** checkbox triggers the calculation of the average diversity by category. The **Annotation key** drop-down menu allows the user to select the annotation key whose values will be used for the category definitions. The **Weigh by frequency** checkbox allows the user to enable the weighing by the category frequency.

The **Contexts** section is available in several variants depending on the value selected in the **Mode** drop-down menu. The latter allows the user to choose among the context specification modes described above. The **No context** mode corresponds to the case where the variety measure is applied globally to the entire unit segmentation.

The **Sliding window** mode (see *figure 2*) implements the notion of a "sliding window" introduced earlier. It allows the user to observe the evolution of variety throughout the segmentation. The only parameter is the window size (in number of segments), set by means of the **Window size** cursor.

Finally, the **Containing segmentation** mode (see *figure 3*) corresponds to the case where the contexts are defined by the segment types appearing in a given segmentation. This segmentation is selected among the input segmentations by means of the **Segmentation** drop-down menu. The **Annotation key** menu shows the possible annotation keys associated to the selected segmentation; if one of these keys is selected, the corresponding annotation values will constitute the row headers; if on the other hand the value *(none)* is selected, the *content* of the segments will be used. The **Merge contexts** checkbox allows the user to measure the variety globally in the entire segmentation that defines the contexts.

In the **Resampling** section, the **Apply resampling** checkbox allows the user to enable the calculation of the average diversity in subsamples of fixed size. The number of segments by subsample is determined by the **Subsample size** cursor, and the number of subsamples with **Number of subsamples**.

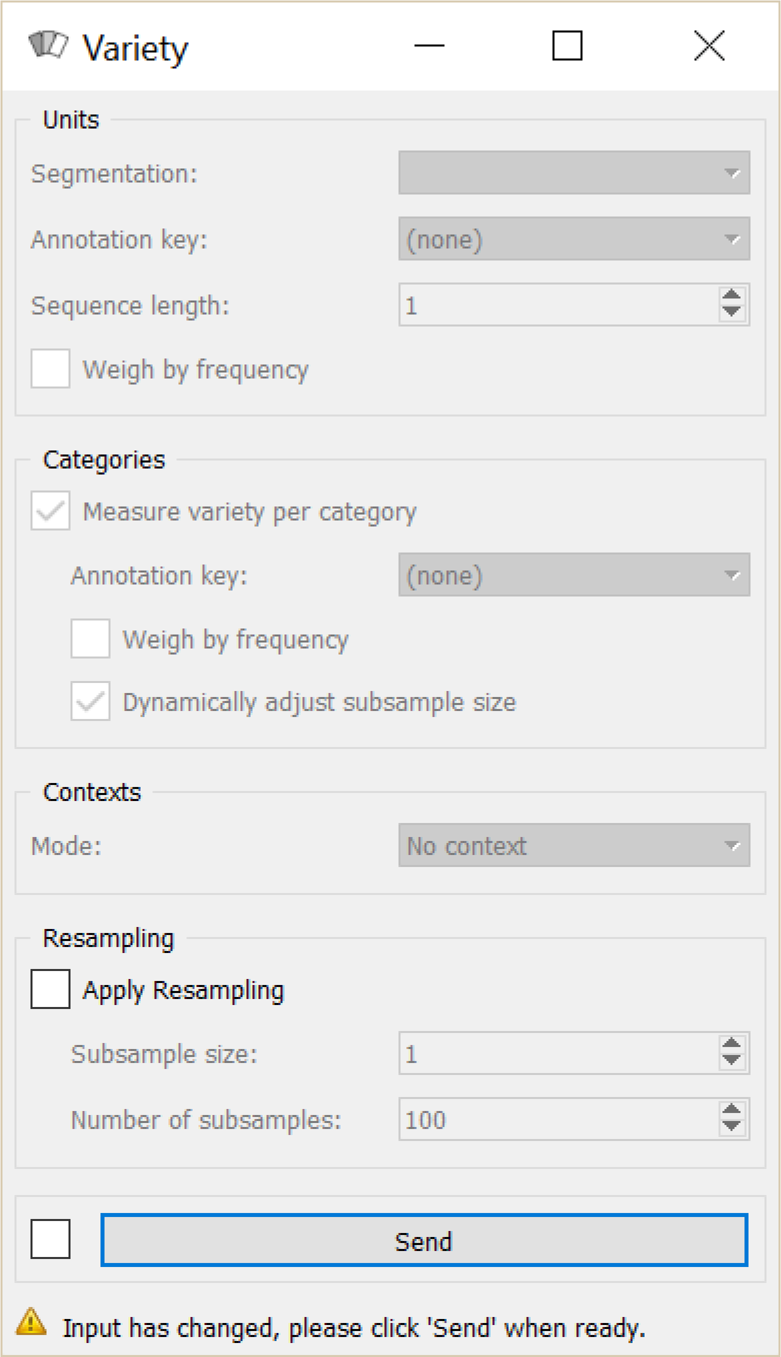
The **Send** button triggers the emission of a table in the internal format of Orange Textable, to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

The informations given under the **Send** button indicate if a table has been correctly emitted, or the reasons why no table is emitted (no input data, typically).

Messages

Information

Data correctly sent to output. This confirms that the widget has operated properly.



Units

Segmentation:

Annotation key: (none)

Sequence length: 1

□ Weigh by frequency

Categories

☑ Measure variety per category

Annotation key: (none)

□ Weigh by frequency

☑ Dynamically adjust subsample size

Contexts

Mode: No context

Resampling

□ Apply Resampling

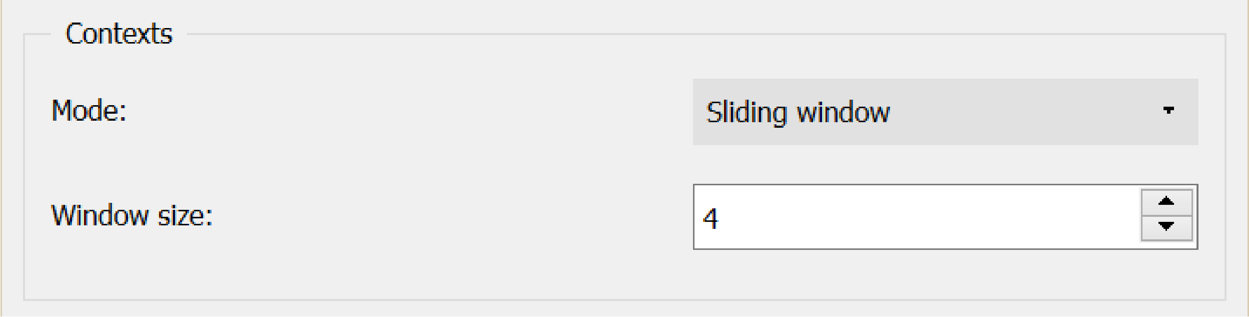
Subsample size: 1

Number of subsamples: 100

□ Send

⚠ Input has changed, please click 'Send' when ready.

Fig. 106: Figure 1: **Variety** widget.

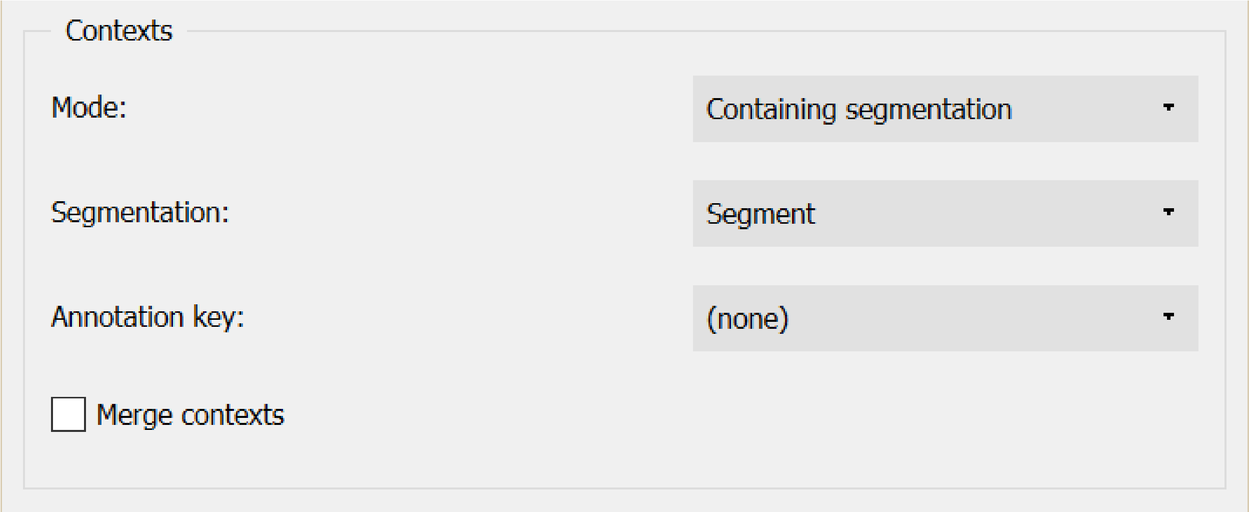


Contexts

Mode: Sliding window

Window size: 4

Fig. 107: Figure 2: **Variety** widget (**Sliding window** mode).



Contexts

Mode: Containing segmentation

Segmentation: Segment

Annotation key: (none)

□ Merge contexts

Fig. 108: Figure 3: **Variety** widget (**Containing segmentation** mode).

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* below).

Warnings

Resulting table is empty. No table has been emitted because the widget instance couldn't find a single element in its input segmentation(s). A likely cause for this problem (when using the **Containing segmentation** mode) is that the unit and context segmentations do not refer to the same strings, so that the units are in effect *not* contained in the contexts. This is typically a consequence of the improper use of widgets *Preprocess* and/or *Recode* (see *Caveat*).

Footnotes

Cooccurrence

Measure the cooccurrence of segments in documents.

Signals

Inputs:

- `Segmentation` (multiple)

  Segmentation whose segments constitute the units subject to measurement of their cooccurrence or the contexts in which unit cooccurrence will be measured

Outputs:

- `Pivot Crosstab`

  Table displaying the cooccurrence of units in the defined context

Description

This widget inputs one or several segmentations, measures the number of documents in which the input segments occur simultaneously, and sends the result in the form of a *cooccurrence matrix*[1].

[1] The definition of cooccurrence may vary depending on the discipline in which this notion is used. In text analytics, the cooccurrence is the number of the documents in which two textual units simultaneously occur. Here by convention, cooccurrence is the dot product of the transposed term-document matrix with itself, which is symmetric when considering only one unit type. As a result, and contrary to other definitions, the diagonal members of the matrix are not zero; rather, they indicate the document frequency of the corresponding textual unit (i.e. the number of

The cooccurrence matrix produced by this widget is of *IntPivotCrosstab* type, a subtype of the generic *Table* format (see *Convert* widget, section *Table formats*). Since this table is a cooccurrence matrix, both rows and columns correspond to *unit* types. The cell at the intersection of a given column and row represents the number of documents (*context* types) in which these two *unit* types occur simultaneously. As the measure of cooccurrence represents absolute frequency, the resulting table contains integer numbers, and as such it is of *IntPivotCrosstab* type, a subclass of *PivotCrosstab*.

To take a simple example, consider two segmentations of the string *a simple example*[2]:

A) label = *words*

content	start	end	*part of speech*	*word category*
a	1	1	*article*	*grammatical*
simple	3	8	*adjective*	*lexical*
example	10	16	*noun*	*lexical*

B) label = *letters* (extract)

content	start	end	*letter category*
a	1	1	*vowel*
s	3	3	*consonant*
i	4	4	*vowel*
...	...	...	...
e	16	16	*vowel*

Typically, we could define unit types based on the content of the segments of the *letters* segmentation.

As for the context types, there are two distinct forms of contexts for measuring the cooccurrence of the units: * **Sliding window* Containing segmentation**

Sliding window relies on the notion of a "window" of *n* segments that we progressively "slide" from the beginning to the end of the segmentation. In our example, by applying this principle to the *letters* segmentation and by setting the window size to 3 segments, we thus define the following contexts:

1. *a si*
2. *sim*
3. *imp*
4. *mpl*
5. *ple*
6. *le e*
7. *e ex*
8. *exa*
9. *xam*
10. *amp*
11. *mpl*
12. *ple*

context types in which it occurs).

[2] By convention, we do not indicate here the string index associated with each segment but only its start and end positions, along with the various annotation values associated with it; moreover, for the sake of readability, we do indicate the content of each segment, though it is not formally part of the segmentation (but rather of the string to which the segmentation refers).

Considering the letter segmentation as that of the unit types, we would obtain the following cooccurrence matrix[3]:

__unit__	a	s	i	m	p	l	e	x
a	4	1	1	2	1	0	1	2
s	1	2	2	1	0	0	0	0
i	1	2	3	2	1	0	0	0
m	2	1	2	6	4	2	0	1
p	1	0	1	4	6	4	2	0
l	0	0	0	2	4	5	3	0
e	1	0	0	0	2	3	5	2
x	2	0	0	1	0	0	2	3

Alternatively, we could consider the *annotation values* of the units instead of their content. For example, by defining units based on the annotations associated to the key *letter category* in the *letters* segmentation, and choosing the mode **Sliding window** for the context with the window size of 3 (see *figure 1*), we would obtain the following cooccurrence matrix:

__unit__	vowel	consonant
vowel	10	10
consonant	10	12

The mode **Containing segmentation** consists in measuring the cooccurrence of units in context defined by another segmentation. In the above example we consider *letter* as the segmentation for unit types and *word* as the segmentation for context types, and thus the following cooccurrence matrix will be obtained and is symmetric by definition:

__unit__	a	s	i	m	p	l	e	x
a	2	0	0	1	1	1	1	1
s	0	1	1	1	1	1	1	1
i	0	1	1	1	1	1	1	1
m	1	1	1	2	2	2	2	1
p	1	1	1	2	2	2	2	1
l	1	1	1	2	2	2	2	1
e	1	1	1	2	2	2	2	1
x	1	0	0	1	1	1	1	1

Each cell at the above table represents the number of words (segments of the context types) in which the unit in the column and the unit in the row are used simultaneously. For example, "2" in the fifth column and forth row, shows that there are two words in which *p* and *m* occur together.

In the **Containing segmentation** mode, it is also possible to measure the cooccurrence of units belonging to distinct segmentations. For instance this would enable us to know how many times a given vowel and a given consonant occur simultaneously in each word. By ticking the **Secondary units** checkbox in the interface of the widget, we will be able to define a segmentation for secondary unit types. In this case, the resulting cooccurrence matrix will no longer be symmetric. Therefore, in the above example, vowels as the primary units segmentation constitute the rows, and consonants as the secondary units segmentation constitute the columns of the resulting cooccurrence matrix (see *figure 2*):

__unit__	s	m	p	l	x
a	0	1	1	1	1
i	1	1	1	1	0
e	1	2	2	2	1

[3] The first column header, *__unit__*, is a name predefined by Orange Textable.

As mentioned in the **Sliding window** mode, it is always possible to measure the cooccurrence of the annotation values of the units (primary and secondary) and those of the contexts instead of the content of segments. In the case of the above example with the secondary units, the resulting crosstab consists of only one cell indicating the number of words in which every letter with *vowel* and every letter with *consonant* annotation value have occurred at the same time:

__unit__	*consonant*
vowel	2

Note that it is up to the user to provide a coherent definition of the units and contexts. In general, there are three conditions to be met in this respect: (a) the segment corresponding to the unit and the context are both associated to the same string, (b) the initial position of the unit segment in the string is higher or equal to that of the context segment, and (c) conversely the final position of the unit is lower or equal to that of the context. In short, the unit must be *contained* within the context.

It is also noteworthy that in order to measure the cooccurrence, it is by definition necessary to specify a context. The context is set to the **Sliding window** mode by default.

Finally, in every scenario considered here, we could also take an interest for the cooccurrence of sequences of 2, 3, . . . , *n* segments (or *n–grams*) rather than for the frequency of isolated segments. The cooccurrence matrix of bigrams in **Sliding window** (size 3) is illustrated below:

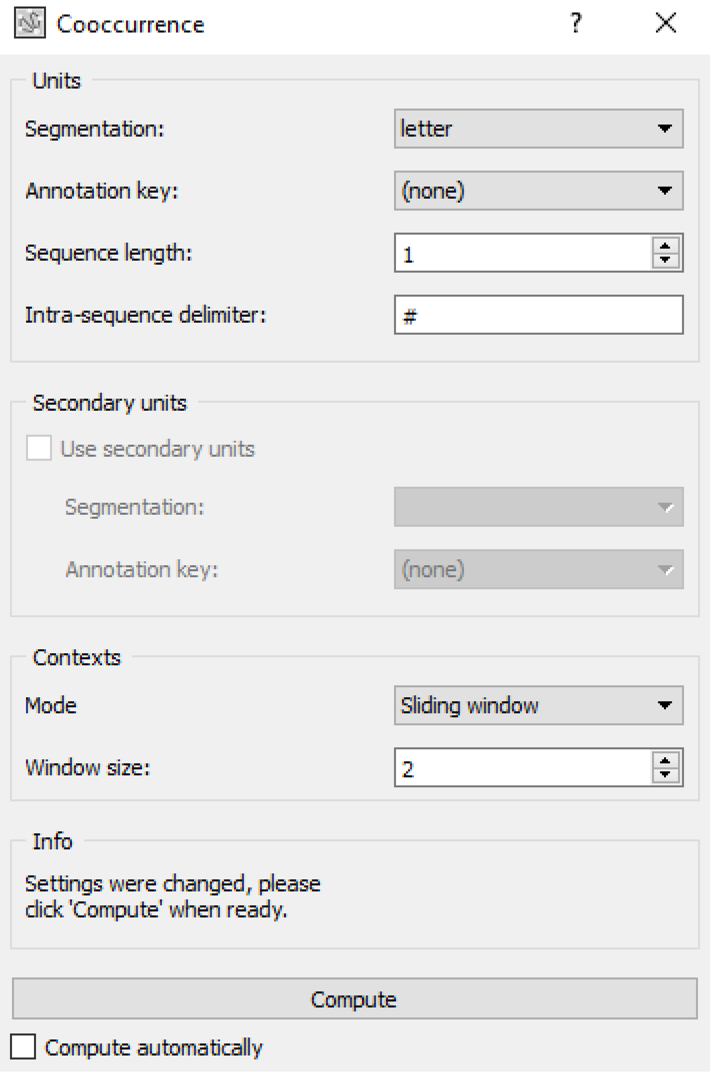
__unit__	*as*	*si*	*im*	*mp*	*pl*	*le*	*ee*	*ex*	*xa*	*am*
as	1	1	0	0	0	0	0	0	0	0
si	1	2	1	0	0	0	0	0	0	0
im	0	1	2	1	0	0	0	0	0	0
mp	0	0	1	4	2	0	0	0	0	0
pl	0	0	0	2	4	2	0	0	0	0
le	0	0	0	0	2	3	1	0	0	0
ee	0	0	0	0	0	1	2	1	0	0
ex	0	0	0	0	0	0	1	2	1	0
xa	0	0	0	0	0	0	0	1	2	1
am	0	0	0	1	0	0	0	0	1	2

Hereafter the interface of the widget will be introduced (see figures *1* to *4*). It contains three separate sections for unit definition (**Units** and **Secondary units**) and context definition (**Contexts**).

In the **Units** section, the **Segmentation** drop-down menu allows the user to select among the input segmentations, the one whose segment types will be subject to the cooccurrence measurement. The **Annotation key** menu displays the annotation keys associated to the chosen segmentation, if any; if one of the keys is selected, the corresponding annotation values will be considered; if on the other hand the value *(none)* is selected, the *content* of the segments will be taken into consideration. The **Sequence length** drop-down menu allows the user to indicate if isolated segments or segment *n–grams* should be considered; in the latter case, the (optional) string specified in the **Intra sequence delimiter** text field will be used to separate the content or the annotation value corresponding to each segment in the table headers.

The **Secondary units** section has almost the same characteristics as the **Units** section, except for the fact that there is no **Sequence length** menu. This section is by default disabled due to the default mode of the **Contexts** section being **Sliding window**, in which only one unit segmentation can be considered for the measure of cooccurrence (see *figure 1*). When changing the mode to *Containing segmentation*, the box becomes automatically enabled (see *figure 2*).

The **Contexts** section is available in two forms, depending on the selected value in the **Mode** drop-down menu. This allows the user to choose between the two possible ways of defining contexts described earlier. The **Sliding window** mode (see *figure 3*) implements the notion of a “sliding window” introduced earlier. Typically, it allows the user to observe the cooccurrence of the unit types with one another throughout the unit segmentation. The only parameter is the window size (in number of segments), defined by the **Window size** cursor, set to 2 by default.



Cooccurrence

Units

Segmentation: letter

Annotation key: (none)

Sequence length: 1

Intra-sequence delimiter: #

Secondary units

□ Use secondary units

Segmentation:

Annotation key: (none)

Contexts

Mode: Sliding window

Window size: 2

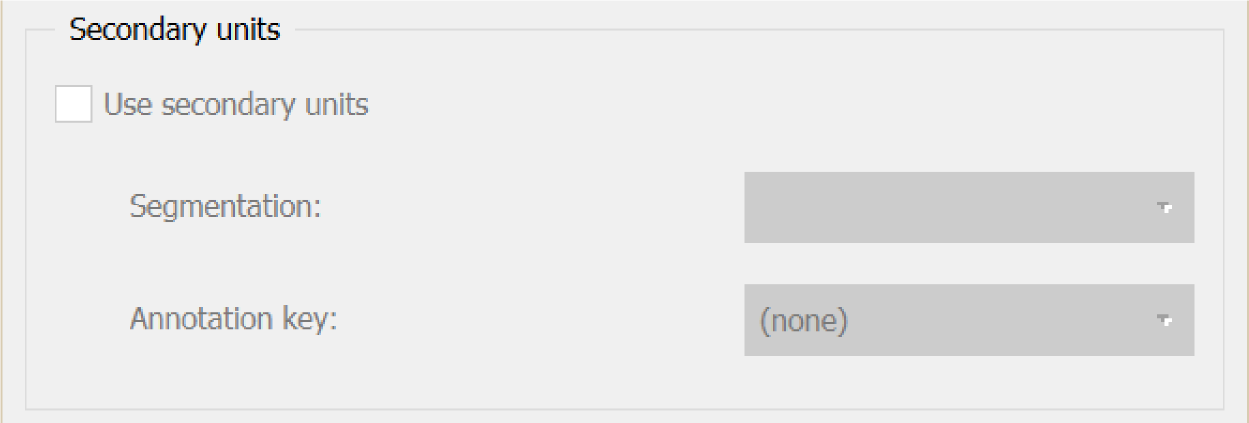
Info

Settings were changed, please
click 'Compute' when ready.

Compute

□ Compute automatically

Fig. 109: Figure 1: **Cooccurrence** widget (**Sliding window** mode as the default mode).



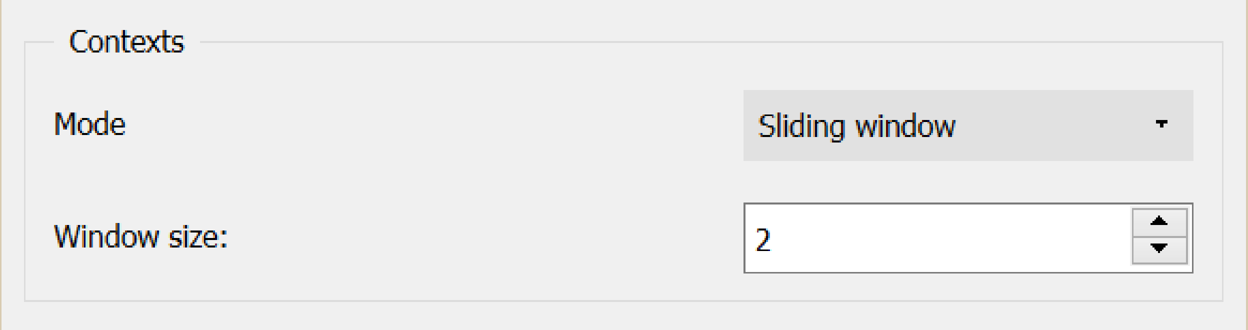
Secondary units

□ Use secondary units

Segmentation:

Annotation key: (none)

Fig. 110: Figure 2: **Secondary units** box of **Cooccurrence** widget.

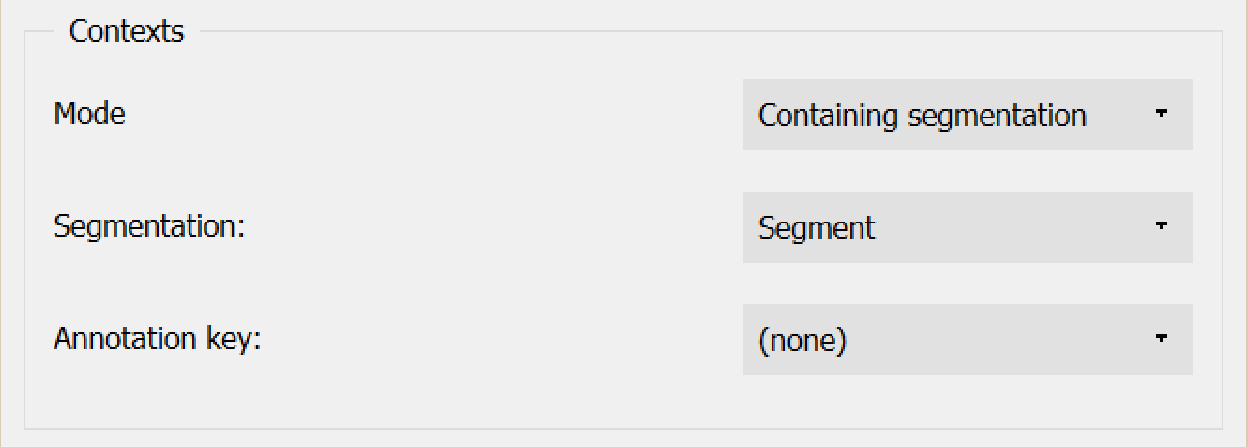


Contexts

Mode: Sliding window

Window size: 2

Fig. 111: Figure 3: **Cooccurrence** widget (**Sliding window** mode).



Contexts

Mode: Containing segmentation

Segmentation: Segment

Annotation key: (none)

Fig. 112: Figure 4: **Cooccurrence** widget (**Containing segmentation** mode).

Finally, the **Containing segmentation** mode (see *figure 4*) corresponds to the case where contexts are defined by the segment types that appear in another segmentation. This segmentation is selected among the input segmentations by means of the **Segmentation** drop-down menu. The **Annotation key** menu displays the annotation keys associated with the context segmentation, if any; if one of the keys is selected, the corresponding annotation value types will constitute the row headers; otherwise the value *(none)* is selected so that *content* of the segments will be exploited.

The **Send** button triggers the emission of a table in the internal format of Orange Textable, to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

The informations given below the **Send** button indicate whether or not the data is correctly sent to the output table. If not, the respective error message will be given.

Messages

Information

Data correctly sent to output. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* below).

Warnings

Resulting table is empty. No table has been emitted because the widget instance couldn't find a single element in its input segmentation(s). A likely cause for this problem (when using the **Containing segmentation** mode) is that the unit and context segmentations do not refer to the same strings, so that the units are in effect *not* contained in the contexts. This is typically a consequence of the improper use of widgets *Preprocess* and/or *Recode* (see *Caveat*).

See also

- *Reference: Convert widget (section "Table formats")*

Footnotes

Context

Explore the context of segments.

Signals

Inputs:

- `Segmentation` (multiple)

  Segmentation containing the "key segments" whose context will be examined or the segments which serve to define these contexts.

Outputs:

- `Textable table`

  Table displaying the concordance of key segments or their collocations.

Description

This widget inputs one or several segmentations and outputs concordances or collocation lists in table format, allowing the user to examine the contexts in which selected segments appear.

The functioning of this widget lies on the notions of units and contexts, as all table contruction widgets. The role of the unit segmentation is central; it defines the key segments whose contexts can be examined by means of the resulting concordances or lists of collocations.

To take a simple example, consider two segmentations of the string *a simple example*[1]:

A) label = *words*

content	start	end	*part of speech*	*word category*
a	1	1	*article*	*grammatical*
simple	3	8	*adjective*	*lexical*
example	10	16	*noun*	*lexical*

B) label = *letters* (extract)

content	start	end	*letter category*
a	1	1	*vowel*
s	3	3	*consonant*
i	4	4	*vowel*
…	…	…	…
e	16	16	*vowel*

The simplest case is when a single segmentation is considered; the only way to define contexts is thus in terms of a given number of neighboring segments. For example, given the single *letters* segmentation, we can build the following concordance:

[1] By convention, we do not indicate here the string index associated with each segment but only its start and end positions, along with the various annotation values associated with it; moreover, for the sake of readability, we do indicate the content of each segment, though it is not formally part of the segmentation (but rather of the string to which the segmentation refers).

__id__	__pos__	1L	__key_segment__	1R
1	1	—	*a*	*s*
2	2	*a*	*s*	*i*
3	3	*s*	*i*	*m*
4	4	*i*	*m*	*p*
5	5	*m*	*p*	*l*
6	6	*p*	*l*	*e*
7	7	*l*	*e*	*e*
8	8	*e*	*e*	*x*
9	9	*e*	*x*	*a*
10	10	*x*	*a*	*m*
11	11	*a*	*m*	*p*
12	12	*m*	*p*	*l*
13	13	*p*	*l*	*e*
14	14	*l*	*e*	—

In this table, the column *__id__* gives the index of each key segment (its position in the table). The column *__pos__* indicates the position of each key segment in the unit segmentation, and in this case this information duplicates the previous one (we will see below that it is not always the case). The key segment itself appears in the *__key_segment__* column, and its direct neighbors on the left and the right appear respectively in the columns *1L* and *1R*.

The number of neighbors shown on the left and right can of course be higher, just as we can show the annotation values instead of the segment contents (be it key segments or their neighbors). For example, the following table gives 2 direct neighbors of each letter by showing their annotation value for the key *letter category*:

__id__	__pos__	2L	1L	__key_segment__	1R	2R
1	1	—	—	*a*	*consonant*	*vowel*
2	2	—	*vowel*	*s*	*vowel*	*consonant*
3	3	*vowel*	*consonant*	*i*	*consonant*	*consonant*
4	4	*consonant*	*vowel*	*m*	*consonant*	*consonant*
5	5	*vowel*	*consonant*	*p*	*consonant*	*vowel*
6	6	*consonant*	*consonant*	*l*	*vowel*	*vowel*
7	7	*consonant*	*consonant*	*e*	*vowel*	*consonant*
8	8	*consonant*	*vowel*	*e*	*consonant*	*vowel*
9	9	*vowel*	*vowel*	*x*	*vowel*	*consonant*
10	10	*vowel*	*consonant*	*a*	*consonant*	*consonant*
11	11	*consonant*	*vowel*	*m*	*consonant*	*consonant*
12	12	*vowel*	*consonant*	*p*	*consonant*	*vowel*
13	13	*consonant*	*consonant*	*l*	*vowel*	—
14	14	*consonant*	*consonant*	*e*	—	—

The particularity of such tables is that they give the context of every segment of the single considered segmentation. In general, we are rather interested in certain specific segments, which we can indicate by means of a distinct segmentation. Supposing that we have, in addition to the *letters* segmentation, a segmentation whose label is *key_segments* and that contains only the occurrences of letter *e* (always in the string *a simple example*):[2]

content	start	end	letter category
e	8	8	*vowel*
e	10	10	*vowel*
e	16	16	*vowel*

[2] It is typically by means of the *Select* widget that we could produce such a segmentation.

By specifying the key segments with this segmentation and the contexts (here the neighboring segments) with the *letters* segmentation, we can then produce the following table:

__id__	*__pos__*	*2L*	*1L*	*__key_segment__*	*1R*	*2R*
1	7	*consonant*	*consonant*	*e*	*vowel*	*consonant*
2	8	*consonant*	*vowel*	*e*	*consonant*	*vowel*
3	14	*consonant*	*consonant*	*e*	—	—

This example of a more typical concordance proves, for that matter, that the position of the key segment in the table (column *__id__*) is not necessarily equal to its position in the segmentation that defined the contexts (column *__pos__*).

In the previous examples, the context of each key segment is defined in the terms of the *neighboring* segments in a segmentation. Another possibility is to define the context on the basis of another segmentation whose segments *contain* the key segments. To illustrate this second mode of context characterization, consider the case where units are specified by the *key_segments* segmentation, as previously, and the contexts by the *words* segmentation:

__id__	*__pos__*	*__left__*	*__key_segment__*	*__right__*
1	2	*simpl*	*e*	—
2	3	—	*e*	*xample*
3	3	*exampl*	*e*	—

This example shows the implications of this change of context specification mode. Firstly, the resulting table now has a fixed width[3] of 5 columns: *__id__* and *__key_segment__* have the same function as before; *__pos__* indicates the position of the context segment that contains each key segment (which allows the user to find and view the context in question with the *Display* widget); finally the columns *__left__* and *__right__* respectively give the left and right part of each context segment containing a key segment.

Moreover in this case, replacing the segment content with one of its annotation values would not make much sense. However, it can be useful to indicate such a value in a separate column, as *part of speech* in the following example which also illustrates the possibility of replacing the content of the key segment with an annotation value (here *letter category*):

__id__	*__pos__*	*__left__*	*__key_segment__*	*__right__*	*part of speech*
1	2	*simpl*	*vowel*	—	*adjective*
2	3	—	*vowel*	*xample*	*noun*
3	3	*exampl*	*vowel*	—	*noun*

These examples highlight the versatility of the **Context** widget, whose possibilities are more diverse than those a basic concordancer typically offers – at the cost of a more complex application since it generally involves being able to build and put in relation two or more distinct segmentations of the analyzed text.

We conclude this overview of the capacities of the widget with the building of collocation lists. First note that this functionality is here conceived as a visualization option applicable to a concordance where the context is defined in terms of the *neighboring* (rather than containing) segments. Instead of representing the neighboring segments of each key segment occurrence, we can in fact build a list of these (types of) segments with an indication of the attraction or on the contrary repulsion between each of them and the key segment.

Consider again the example of the concordance presented earlier where the units are given by the *key_segments* segmentation and the context by the *letter category* annotations values of the *letters* segmentation:

[3] Except in the “pathological” case where no key segment is contained in the context segment.

__id__	__pos__	2L	1L	__key_segment__	1R	2R
1	7	*consonant*	*consonant*	*e*	*vowel*	*consonant*
2	8	*consonant*	*vowel*	*e*	*consonant*	*vowel*
3	14	*consonant*	*consonant*	*e*	—	—

The same data enable the program to produce the following collocation list:

__unit__	__mutual_info__	__local_freq__	__local_prob__	__global_freq__	__global_prob__
consonant	0.292781749228	7	0.7	8	0.571428571429
vowel	-0.51457317283	3	0.3	6	0.428571428571

The column *__mutual_info__* gives the mutual information (in bits) between the key segment (here the letter *e)* and each value of the *letter category* annotation that appeared close by (here at a maximum distance of 3 segments) the key segments. This quantity is the binary logarithm of the ratio of the probability of the *letter category* value in question close to the key segment and its probability in the context segmentation in general.

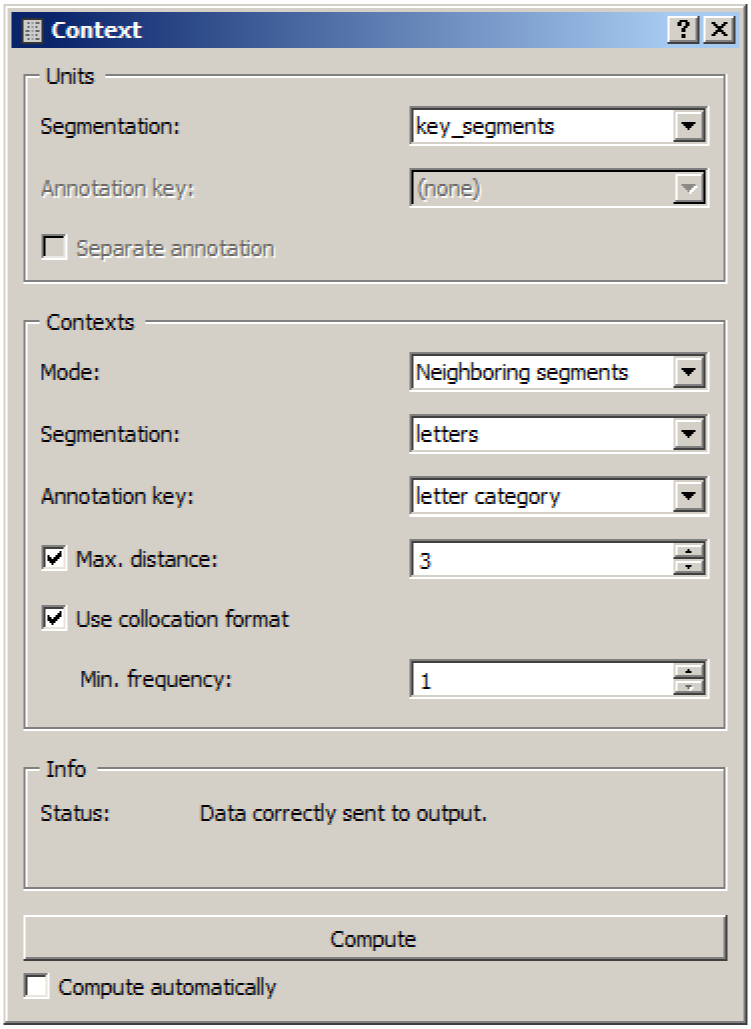
Thus the *consonant* type appears 7 times in the surroundings of *e* (*__local_freq__*), on a total of 10 segments that appeared close, hence the "local" probability of 7/10 = 0.7 (*__local_prob__*); moreover the same type appeared 8 times in the whole *letters* segmentation (*__global_freq__*), on a total of 14 segments, hence the "global" probability of 8/14 = 0.57 (*__global_prob__*). Finally the binary logarithm of 0.7/0.57 = 1.22 is 0.3 bits (*__mutual_info__*), and this (slightly) positive value reflects the (weak) attraction between *e* and the *consonant* type at a maximum distance of 3 segments. Conversely, the negative mutual information between *e* and *vowel* shows that these categories are in a rather repulsive relation in the considered surrounding.

The widget interface (see *figure 1*) is divided in two separate sections of unit specification (**Units**) and context specification (**Contexts**). In the **Units** section, the **Segmentation** drop-down menu allows the user to select among the input segmentations the one whose segments will play the role of key segments. The **Annotation key** menu shows the potential annotation keys associated to the chosen segmentation; if one of the keys is selected the corresponding annotation values will be used; if on the other hand the value *(none)* is selected, it will be the *content* of the segments. The **Separate annotation** button, activated only when an annotation key is selected, enables the user to indicate that the values associated to this key must appear in a separate column (whose header is the corresponding key) rather than replace the segment contents in the column *__key_segment__*. Note that the two buttons (**Annotation key** and **Separate annotation**) are disabled when the button **Use collocation format** is selected.

In the **Context** section, the **Mode** menu allows the user to choose between the two context characterization modes mentioned earlier: in terms of *neighboring* segments of the key segment (**Neighboring segments**) or of segments *containing* them (**Containing segmentation**). In both cases, the segmentation in question is selected among the input segmentation through the **Segmentation** drop-down menu and the **Annotation key** menu shows the potential annotation keys associated to this segmentation. If one of these keys is selected, the display of the corresponding values varies depending on the **Mode** used: in **Neighboring segments** mode, the annotation values replace the content of the segments in the columns *1R**, **1L**, ... ; in **Containing segmentation** mode, they appear in a separate column whose header is the corresponding annotation key.

In **Neighboring segments** mode, the **Contexts** section also allows the user to indicate if a limit should be set to the number of neighboring segments shown for each key segment and where it is set (**Max. distance**). The **Use collocation format** button is used to format the result as a collocation list (rather than a concordance); when it is selected, the **Min. frequency** drop-down menu allows the user to specify the (global) minimal frequency that the segment type must reach in order to appear in the list. Checking the **Treat distinct strings as contiguous** box permits to treat separate strings as if they were contiguous, so that the end of each string is a djacent to the beginning of the next string.

In **Containing segmentation mode** (see *figure 2*), the **Contexts** section allows the user to specify the maximal number of characters that appear in the right and left context of the pivot.



Context

Units

Segmentation: key_segments

Annotation key: (none)

□ Separate annotation

Contexts

Mode: Neighboring segments

Segmentation: letters

Annotation key: letter category

☑ Max. distance: 3

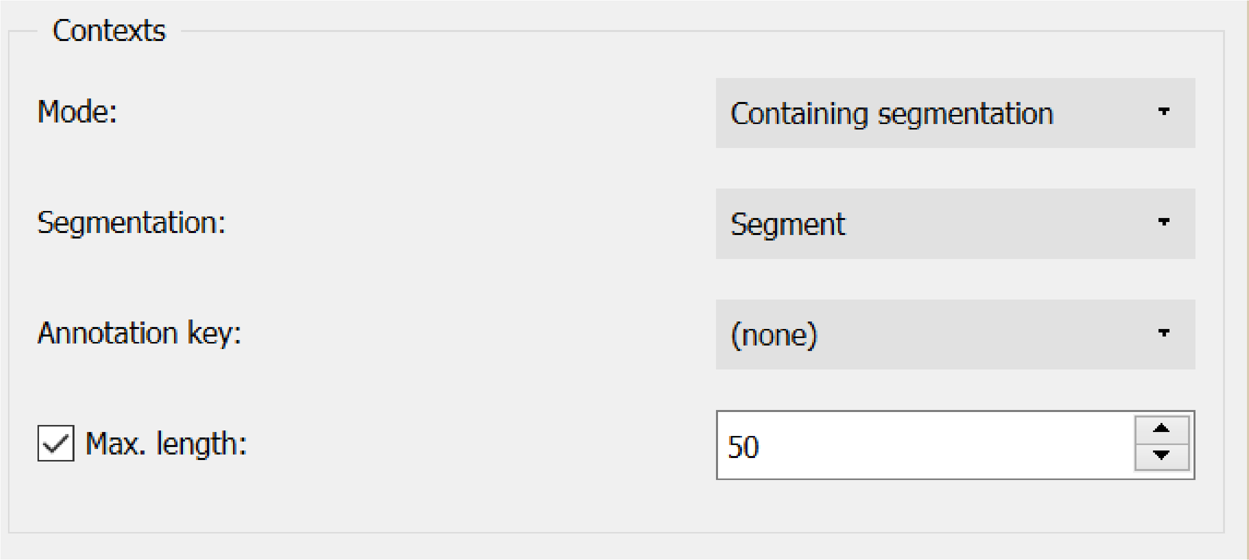
☑ Use collocation format

Min. frequency: 1

Info

Status: Data correctly sent to output.

Fig. 113: Figure 1: Interface of the **Context** widget.



Contexts

Mode: Containing segmentation

Segmentation: Segment

Annotation key: (none)

☑ Max. length: 50

Fig. 114: Figure 2: **Context** widget (Containing segmentation mode).

The **Send** button triggers the emission of a table in the internal format of Orange Textable, to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

The informations generated below the **Send** button indicate if a table was correctly emitted, or the reasons why no table is emitted (typically, because it is empty).

Messages

Information

Data correctly sent to output. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* below).

Warnings

Resulting table is empty. No table has been emitted because the widget instance couldn't find a single element in its input segmentation(s). A likely cause for this problem (when using the **Containing segmentation** mode) is that the unit and context segmentations do not refer to the same strings, so that the units are in effect *not* contained in the contexts. This is typically a consequence of the improper use of widgets *Preprocess* and/or *Recode* (see *Caveat*).

See also

- *Cookbook: Build a concordance*

Footnotes

Category

Build a table with categories defined by segments' content or annotations.

Signals

Inputs:

- `Segmentation` (multiple)

  Segmentation whose segments constitute the basis for category extraction.

Outputs:

- `Textable table`

  Table displaying the extracted categories

Description

This widget inputs one or several segmentations and outputs a tabulated representation of *categories* associated to the segments of one of them; categories are typically defined on the basis of their annotation values of segments for a given annotation key, but may also be defined on the basis of the content of segments.

Typically, tables produced by the **Category** widget are destined to be merged (by means of the built-in **Merge Data** widget of Orange Canvas) with quantitative tables produced by widgets *Count*, *Length*, or *Variety*, in order to associate with each row the piece of categorical information required to train a text classifier (i.e. a system able to automatically predict the membership of a text to a category based on the quantitative profile associated with it). Here is an example of a table with this structure, where the second column would have been constructed by an instance of **Category**, and the columns to its right by an instance of *Count*:

__context__	*__category__*	*noun*	*verb*	...
text1	*news*	35	12	...
text2	*news*	20	8	...
text3	*poetry*	27	18	...
...	...	...	...	...

The tables produced by this widget only contain two columns. The first (header *__context__*) contains the headers corresponding to the contexts – which are essentially defined in the same way as with the **Containing segmentation** mode of widgets *Count*, *Length*, and *Variety*: by the segment types appearing in a segmentation. The second column (header *__category__*) contains the annotation(s) associated with each segment type.

To take a simple example, consider two segmentations of the string *a simple example*[1]:

A) label = *words*

content	start	end	*part of speech*	*word category*
a	1	1	*article*	*grammatical*
simple	3	8	*adjective*	*lexical*
example	10	16	*noun*	*lexical*

B) label = *letters* (extract)

content	start	end	*letter category*
a	1	1	*vowel*
s	3	3	*consonant*
i	4	4	*vowel*
…	…	…	…
e	16	16	*vowel*

Based on the latter segmentation, we can produce the following table, giving the annotation value associated with the key *letter category* for each distinct letter:

__context__	*__category__*
a	*vowel*
s	*consonant*
i	*vowel*
m	*consonant*
p	*consonant*
l	*consonant*
e	*vowel*
x	*consonant*

In this illustration, each letter is only associated to a single category. In a more general case, the contexts can be associated to several categories; for example, if the contexts are defined based on the *word category* annotation of the *words* segmentation and the extracted categories are defined as the segment contents of the *letters* segmentation:

__context__	*__category__*
grammatical	*a*
lexical	*e-m-l-p-a-i-s-x*

In this case, the user will have to choose (a) the order (frequential or ASCII-betical) in which the multiple values will be sorted and (b) whether they should all be shown or only the first (in the selected order).

The widget interface (see *figure 1*) has three separate sections, for unit specification (**Units**), for multiple values processing specification (**Multiple Values**), and for context specification (**Contexts**).

In the **Units** section, the **Segmentation** drop-down menu allows the user to select among the input segmentations the one whose segments will be examined to determine the categories. The **Annotation key** menu shows the possible annotation keys associated to the chosen segmentation; if one of these keys is selected, the corresponding annotation values will be used; if on the other hand the value *(none)* is selected, the *content* of the segments will be used. The **Sequence length** drop-down menu allows the user to indicate if the widget should consider the isolated segments or

[1] By convention, we do not indicate here the string index associated with each segment but only its start and end positions, along with the various annotation values associated with it; moreover, for the sake of readability, we do indicate the content of each segment, though it is not formally part of the segmentation (but rather of the string to which the segmentation refers).

the *n–grams* of segments. In this latter case, the (optional) string specified in the **Intra-sequence delimiter** text field will be used to separate the content or the annotation value corresponding to each individual segment.

In the **Multiple Values** section, the **Sort by** drop-down menu allows the user to select the sorting criteria of multiple values, namely either the frequency (**Frequency**) or the ASCII order (**ASCII**). The **Sort in reverse order** checkbox reverses the sorting order, and the **Keep only first value** checkbox allows the program to retain only the first value (in the selected order). The **Value delimiter** field is used to indicate the character string to insert in-between multiple values.

Unlike other table contruction widgets , here the context specification can only be done in relation to a segmentation containing the unit segmentation (thus the equivalent of the **Containing segmentation** mode of widgets *Count*, *Length*, and *Variety*:). This segmentation is selected among the input segmentation by means of the **Segmentation** drop-down menu. The **Annotation key** menu shows the possible annotation keys associated to the selected segmentation; if one of these keys is selected, the corresponding annotation values will will constitute the row headers; if on the other hand the value *(none)* is selected, the *content* of the segments will be used.

The **Send** button triggers the emission of a table in the internal format of Orange Textable, to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

The informations generated below the **Send** button indicate if a table has been correctly emitted, or the reasons why no table is emitted (no input data, typically).

Messages

Information

Data correctly sent to output. This confirms that the widget has operated properly.

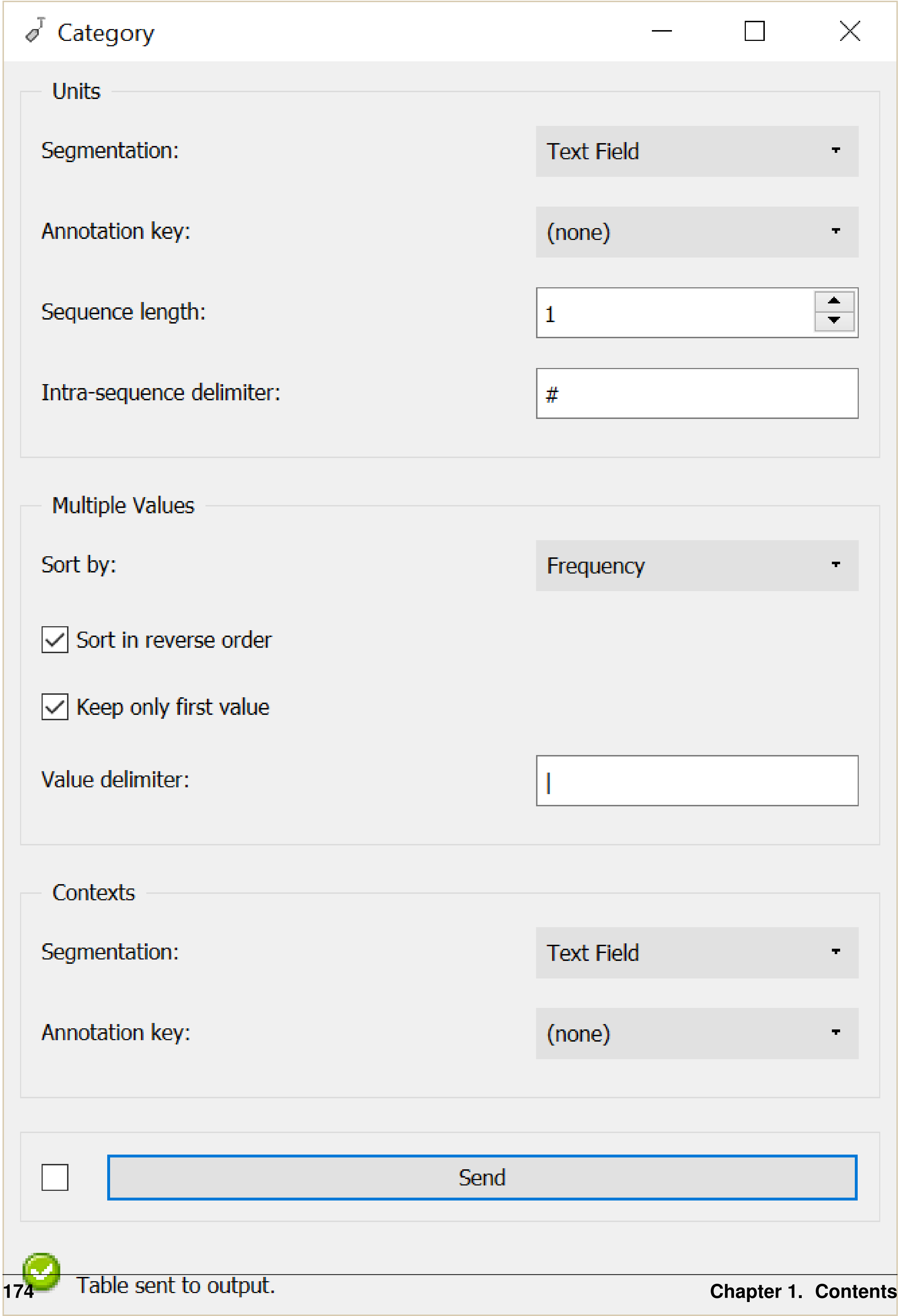
Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* below).

Warnings

Resulting table is empty. No table has been emitted because the widget instance couldn't find a single element in its input segmentation(s). A likely cause for this problem (when using the **Containing segmentation** mode) is that the unit and context segmentations do not refer to the same strings, so that the units are in effect *not* contained in the contexts. This is typically a consequence of the improper use of widgets *Preprocess* and/or *Recode* (see *Caveat*).



Category

Units

Segmentation: Text Field

Annotation key: (none)

Sequence length: 1

Intra-sequence delimiter: #

Multiple Values

Sort by: Frequency

☑ Sort in reverse order

☑ Keep only first value

Value delimiter: |

Contexts

Segmentation: Text Field

Annotation key: (none)

Table sent to output.

Fig. 115: Figure 1: Interface of the **Category** widget.

Footnotes

1.6.4 Conversion/export widgets

The widgets of this category serve diverse purposes unified by the notion of "conversion". *Convert* takes as input tabular data in Orange Textable format and converts them to other formats, in particular the *Table* format appropriate for further processing within Orange Canvas; *Convert* also makes it possible to apply various standard transforms to a table, such as sorting, normalizing, etc., as well as to export its contents in tab-delimited text format. *Message* takes as input a segmentation containing data in a specific JSON format (see *Reference: JSON im-/export format*) and converts them to a "message" that can be used to control the behavior of other widgets.

Convert

Convert, transform, or export Orange Textable tables

Signals

Inputs:

- `Textable Table`

  Table in the internal format of Orange Textable.

Outputs:

- `Orange Table` (default)

  Data in the standard *Table* format of Orange Canvas (possibly transformed).

- `Textable Table`

  Table in the internal format of Orange Textable (possibly transformed).

- `Segmentation`

  Segmentation containing the output table in tab-delimited format.

Description

Convert, inputs data in the internal format of Orange Textable and enables the user to modify them (sorting, normalization, etc.), to convert them to other formats, in particular the standard *Table* format of Orange Canvas (suitable for further processing within Orange Canvas), or to export them in tab-delimited text format (either to a file or to the clipboard).

Table formats

The table representation format of Orange Canvas (*Table* type) presents compatibility issues with Unicode encoded data. Since this encoding is emerging as the most widely used standard for languages of the world, Orange Textable provides its own Unicode-friendly table representation format.

Widgets *Count*, *Length*, *Variety*, *Category*, and *Context*) thus produce tables in Orange Textable format. In order to be manipulated by the numerous tabulated data processing widgets offered by Orange Canvas, these data must be converted to the standard *Table* format of Orange Canvas (and to an encoding supported by this latter format).

Note that the internal Orange Textable *Table* type subdivides in several subtypes. In particular, the contingency tables (see *Count* widget) belong to the *Crosstab* subtype which itself subdivides in *PivotCrosstab*, *FlatCrosstab*, and *WeightedFlatCrosstab*. These three subtypes are equivalent with regard to the information they allow the user to store, and the easiest way to understand what differentiates them is to see an example.

Consider the following contingency table, of *IntPivotCrosstab*[1] type (such as produced by the *Count* widget):

__context__	unit1	unit2
context1	1	3
context2	2	1

Here is the same information converted in *FlatCrosstab* format:

__id__	__unit__	__context__
1	*unit1*	*context1*
2	*unit2*	*context1*
3	*unit2*	*context1*
4	*unit2*	*context1*
5	*unit1*	*context2*
6	*unit1*	*context2*
7	*unit2*	*context2*

This representation contains three columns carrying the headers *__id__*, *__unit__* and *__context__*, and a number of rows equal to the total count of the contingency table. It is the standard way of encoding a contingency table in Orange Canvas, and it is required by widgets such as *Correspondence Analysis* (after conversion to the *Table* type defined by Orange Canvas).

The *WeightedFlatCrosstab* format produces a more compact representation by keeping only one copy of each distinct unit–context pair and by adding a column *__count__* to save information on the number of repetition of each pair:

__id__	__unit__	__context__	__weight__
1	*unit1*	*context1*	1
2	*unit2*	*context1*	3
3	*unit1*	*context2*	2
4	*unit2*	*context2*	1

This format is sometimes used to represent contingency tables in third-party data analysis software. It is often called “sparse” matrix format. format.

Output channels

Regardless of the selected output table format (or the transforms that have been applied to the data, see *Advanced interface* below), the **Convert** widget emits data on three distinct output channels:

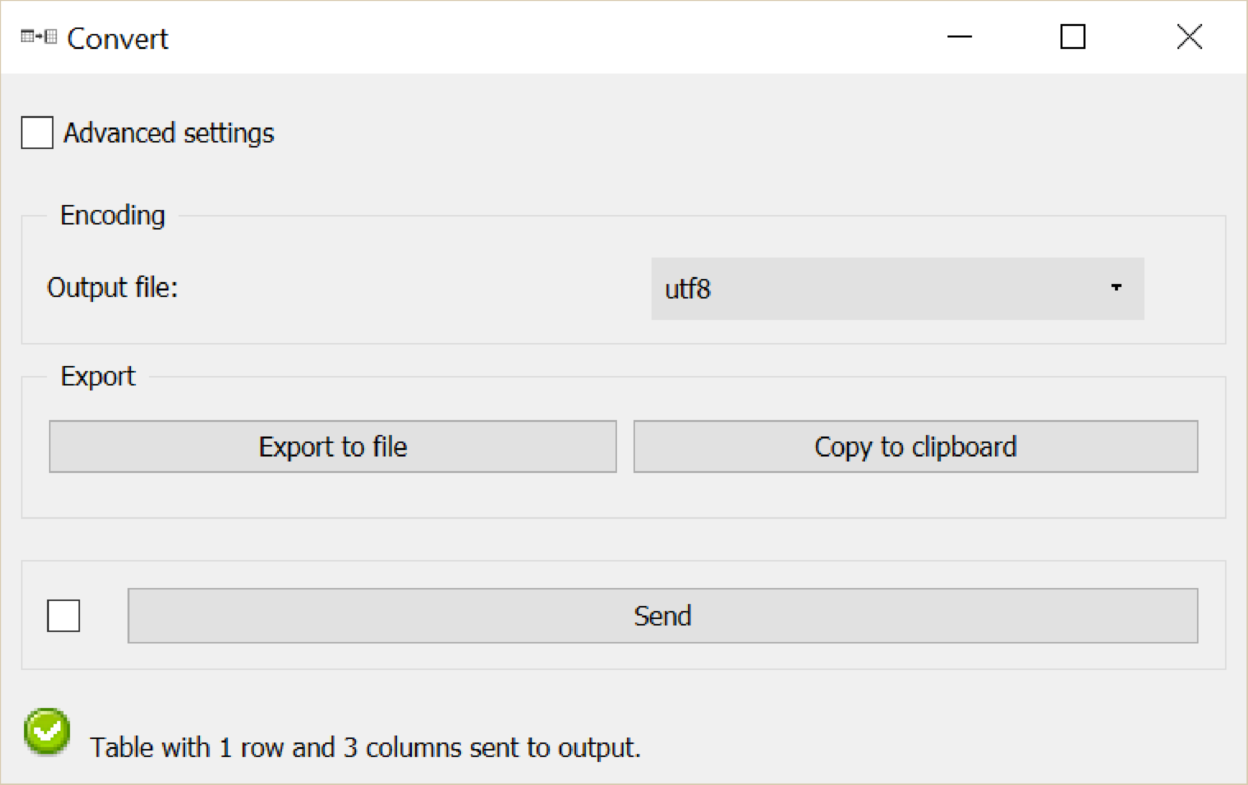
- The default output channel (*Orange Table*) emits data converted to standard *Table* format of Orange Canvas; it will typically be used for passing them to built-in Orange Canvas table processing widgets.

[1] *IntPivotCrosstab* is in turn a subtype of *PivotCrosstab* (and similarly *IntWeightedFlatCrosstab* is a subtype of *WeightedFlatCrosstab*), whose specificity is to be limited to integer values.

- The *Textable Table* channel outputs a table in the internal format of Orange Textable (usually after applying some set of transforms); it can then be sent to another instance of **Convert** (in cases where it is useful to apply transforms in distinct steps) or to an instance of the built-in **Python script** widget of Orange Canvas, for accessing the content of the table in a programmatic fashion.
- The *Segmentation* channel emits a segmentation with a single segment enclosing a version of the (possibly transformed) table in tab-delimited text format (in utf-8 encoding), which is suitable for further textual processing using Orange Textable widgets such as *Recode* or *Segment* for instance.

Basic interface

The basic version of the widget (see *figure 1* below) is essentially limited to the **Encoding** section, which allows the user to select an encoding for the output data. This can be done for the data possibly exported to a text file in tab-delimited format (**Output File**). If certain characters cannot be converted to the specified encoding (for example accentuated characters in the ASCII encoding), they are automatically replaced by corresponding HTML entities (for example *&#233;* for *é*).


Convert

□ Advanced settings

Encoding

Output file: utf8

Export

□ Send

Table with 1 row and 3 columns sent to output.

Fig. 116: Figure 1: **Convert** widget (basic interface).

The **Export** section allows the user to export a version of the (possibly transformed) table in tab-delimited text format, either to a text file (**Export to file**) or to the clipboard (**Copy to clipboard**), in order to paste it to a spreadsheet opened in a third-party program for instance. In the former case, the **Output file** drop-down menu (section **Encoding**) is used to indicate which encoding the data should be converted to before being saved; typically, except for a limit imposed by the further processing planned for the saved data (for example by a specific data analysis program), we will seek to keep here the maximum amount of information by specifying either the original encoding of the data, or a more general encoding (a variant of Unicode for example). Note that when the data are copied to the clipboard, the utf-8 encoding is used by default (regardless of what has been selected in the **Encoding** section).

Advanced interface

The advanced version of the **Convert** widget (see *figure 2* below) contains an additional section (**Transform**) allowing the user to apply a number of standard modifications to the incoming table. The different operations defined in this section are applied to input data in the order in which they appear in the interface, top to bottom. The modified data can then be emitted on output connections or exported (either to a file or to the clipboard).

The **Sort rows by column** checkbox triggers row sorting. If it is selected, the column headers of the table appear in the drop-down menu directly on the right and the user can thus select the column on the basis of which the rows will be sorted. If the **Reverse** box on the right of the drop-down menu is checked, rows will be sorted by *decreasing* value.

Sort columns by row controls in a similar way column sorting. It should be noted in this case that the first column (containing row headers) will always stay in the same position; the sorting only affects the following columns. To sort the columns based on the header row, you must select the first option in the **Sort columns by row** drop-down menu in the right. It will typically contain a name predefined by Orange Textable but which does not appear in the table (*__unit__* if it is a contingency table of *PivotCrosstab* type such as produced by the *Count* widget, and the generic header *__col__* in every other case).

The **Transpose** checkbox allows the user to transpose the table, which means invert its rows and columns. This option is only available for *PivotCrosstab* type contingency tables.

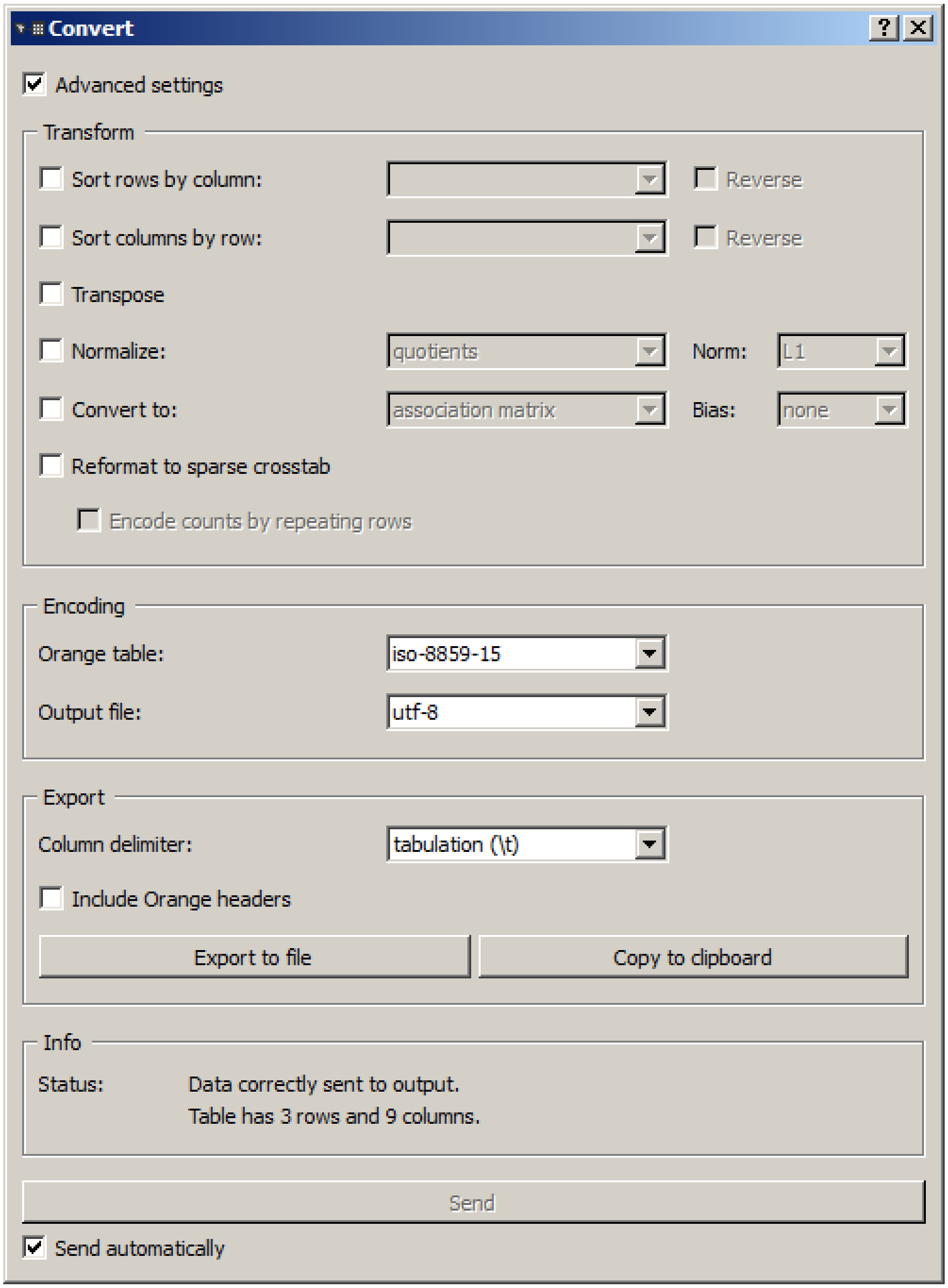
The **Normalize** checkbox triggers the normalization of the table (in a rather loose sense of the term); it is only applicable for *PivotCrosstab* type contingency tables. If it is selected, the user can choose in the drop-down menu directly on the right whether the normalization should be applied by rows (**rows**) or by columns (**columns**); the **Norm** drop-down menu allows the user to select the type of normalization, either **L1** (division by the sum of the row/column) or **L2** (division by the root of the sum of the squares of the row/column).

Three more operations (which are not usually classified as normalizations in the strict sense of the term) can be selected in the drop-down menu, each of which deactivates the **Norm** drop-down menu on the right:

- In **quotients** mode, the count stored in each cell of a contingency table (of *PivotCrosstab* type) is divided by the corresponding “theoretical” count under the hypothesis of independence between table rows and columns. This quotient is superior to 1 if the row and the column in question are in a mutual attraction relation, inferior to 1 in case of repulsion between the row and the column, finally equal to 1 if the row and column do not repulse nor attract each other particularly.
- In **TF–IDF** mode, the count stored in each cell of a contingency table (of *PivotCrosstab* type) is multiplied by the natural log of the ratio of the number of rows (i.e. contexts) having nonzero frequency for this column (i.e. unit) to the total number of rows.
- In **presence/absence** mode, counts greater than 1 are replaced by the value 1, so that the resulting table can contain only 0’s and 1’s.

The common property of all operations available in the **Normalize** drop-down menu is that they preserve the original dimensions of the input contingency table. On the contrary, the **Convert to** checkbox (only applicable for *PivotCrosstab* type tables) allows the user to trigger the application of transforms which actually modify the dimensionality of the table :

- In **document frequency** mode, a new contingency table is created, which giver, for each column (i.e. unit) the number of distinct rows (i.e. contexts) that have nonzero frequency (hence the resulting table contains a single row).
- In **association matrix** mode, a new symmetric table is constructed, where each cell gives a measure of the (Markov) associativity between a pair of columns (i.e. units) in the original contigency table: two columns are thus strongly associated if they have similar profiles of attraction/repulsion with rows (i.e. contexts). Selecting this mode activates the **Bias** drop-down menu on the right, which allows the user to select between three predefined ways of weighing the contributions of high versus low frequencies in this computation: **frequent** emphasizes strong associations between frequent units; **none** provides a balanced compromise between frequent



Convert

☑ Advanced settings

Transform

☐ Sort rows by column: ☐ Reverse

☐ Sort columns by row: ☐ Reverse

☐ Transpose

☐ Normalize: quotients Norm: L1

☐ Convert to: association matrix Bias: none

☐ Reformat to sparse crosstab

☐ Encode counts by repeating rows

Encoding

Orange table: iso-8859-15

Output file: utf-8

Export

Column delimiter: tabulation (\t)

☐ Include Orange headers

Info

Status: Data correctly sent to output.
Table has 3 rows and 9 columns.

☑ Send automatically

Fig. 117: Figure 2: **Convert** widget (advanced interface).

and rare units; **rare** emphasizes strong associations between rare units (note that in this particular case, values greater than 1 express an attraction and values lesser than 1 a repulsion)[2].

It is worth mentioning that the **Normalize** and **Convert to** checkboxes are mutually exclusive and deactivate one another.

Finally, the **Reformat to sparse crosstab** checkbox allows the user to convert a contingency table from the *PivotCrosstab* format to the *WeightedFlatCrosstab* or from *IntPivotCrosstab* to *IntWeightedFlatCrosstab* (see the *Table formats* section above). In turn, data in *IntWeightedFlatCrosstab* format can be converted to *FlatCrosstab* by further selecting option **Encode counts by repeating rows**; the latter option is only available when dealing with tables containing integer values.

Compared to its basic version (see *Basic interface* above), the advanced version of the **Export** section offers two extra controls. The **Column delimiter** drop-down menu allows the user to select the column separator that will be inserted between cell values when exporting a table in text format; possible choices are *tabulation (t)*, *comma (,)*, and *semicolon (;)*. The **Output Orange headers** checkbox allows the user to indicate if the output should include every header line of the format *.tab* specific to Orange Canvas (**Output Orange headers**)–which is useful only for re-importing the exported table using the built-in **File** widget of Orange Canvas (and in fact often necessary in that case). Both parameters (**Column delimiter** and **Output Orange headers** also apply to the data sent on the *Segmentation* output channel)

The **Send** button triggers data emission to the output connection(s) (see *Output channels* above). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically send data at every modification of its interface or when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

The informations generated below the **Send** button indicate the number of lines and columns in the output table, or the reasons why no table is emitted (no input data).

Messages

Information

Data correctly sent to output: table has <n> and <m> columns. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input table. The widget instance is not able to emit data to output because it receives none on its input channel(s).

See also

- *Cookbook: Display table*
- *Cookbook: Export table*

[2] For more details on the calculation of Markov associativities, see Bavaud F. and Xanthos A. (2005). Markov associativities. *Journal of Quantitative Linguistics, 12*:123–137. Details on the effect of the **bias** parameter can be found in Deneulin, P., Gautier, L., Le Fur, Y., and Bavaud, F. (2014). Corrélats textuels autour du concept de minéralité dans les vins. In Actes des 12èmes Journées internationales d'analyse statistique des données textuelles (JADT 2014), pp. 209–223; the predefined values of this parameter (**frequent**, **none**, and **rare**) correspond respectively to values 1, 0.5 and 0 of parameter *alpha* in the above cited reference.

Footnotes

Message

Parse JSON data in segmentation and use them to control other widgets.

Signals

Inputs:

- `Segmentation`

  Segmentation containing a single segment with the JSON data to be parsed

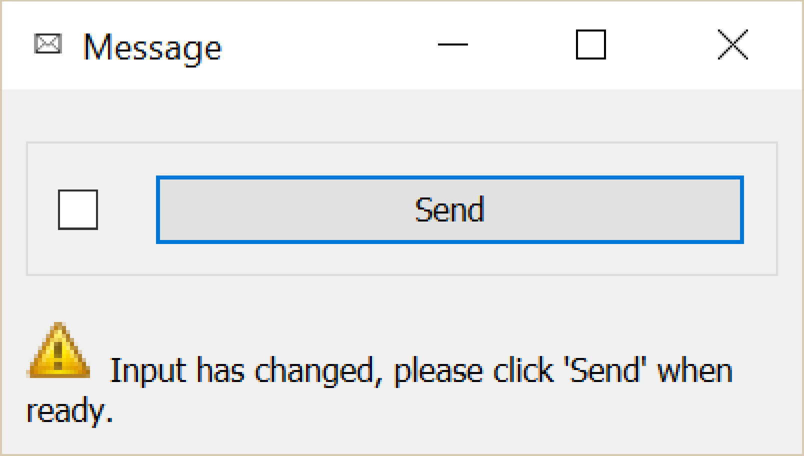
Outputs:

- `Message`

  JSONMessage object that can be sent to other widgets

Description

This widget inputs a segmentation containing a single segment whose content is in JSON format. After validation, the data are converted to a *JSONMessage* object and emitted to the widget's output connections. Provided that the data conform to one of the formats described in section *JSON im-/export format*, the *JSONMessage* object can be sent to an instance of the corresponding widget (either *Text Files*, *URLs*, *Recode*, or *Segment*) and used to control its behavior remotely.



Input has changed, please click 'Send' when ready.

Fig. 118: Figure 1: Interface of the **Message** widget.

The widget's interface offers no user-controlled option (see *figure 1* above).

The **Send** button triggers the emission of a **JSONMessage** object to the output connection(s). When it is selected, the **Send automatically** checkbox disables the button and the widget attempts to automatically emit a segmentation when its input data are modified (by deletion or addition of a connection, or because modified data is received through an existing connection).

The informations generated below the **Send** button indicate the number of items present in the parsed JSON data, or the reasons why no *JSONObject* can be emitted (no input or invalid data, input segmentation containing more than one segment).

Messages

Information

Data correctly sent to output: <n> items. This confirms that the widget has operated properly.

Settings were* (or *Input has*) *changed, please click 'Send' when ready. Settings and/or input have changed but the **Send automatically** checkbox has not been selected, so the user is prompted to click the **Send** button (or equivalently check the box) in order for computation and data emission to proceed.

No data sent to output yet: no input segmentation. The widget instance is not able to emit data to output because it receives none on its input channel(s).

No data sent to output yet, see 'Widget state' below. A problem with the instance's parameters and/or input data prevents it from operating properly, and additional diagnostic information can be found in the **Widget state** box at the bottom of the instance's interface (see *Warnings* and *Errors* below).

Warnings

Input segmentation contains more than 1 segment. The input segmentation must contain exactly 1 segment.

Errors

JSON parsing error. The input JSON data couldn't be correctly parsed. Please use a JSON validator to check the data's well-formedness.

See also

- *Reference: Text Files widget*, *Remote control*
- *Reference: URLs widget*, *Remote control*
- *Reference: Segment widget*, *Remote control*
- *Reference: Recode widget*, *Remote control*
- *Reference: JSON im-/export format*

1.6.5 JSON im-/export format

Beyond a restricted number of sources, substitutions, or regular expressions, it becomes tedious to configure instances of widgets *Text Files*, *URLs*, *Recode*, and *Segment* using their advanced interface. To alleviate this issue, these widgets enable the user to import or export manually edited configuration lists in JSON format as described in the following sections.

Generalities

The general format of JSON configuration files is the following:

```
[
    {
        "key_1": value_1,
        "key_2": value_2,
        ...
        "key_N": value_N
    },
    ...
    {
        "key_1": value_1,
        "key_2": value_2,
        ...
        "key_N": value_N
    }
]
```

NB:

- the file must be encoded in utf-8
- the whole file is included between square brackets `[ ... ]`
- each entry of the list is included between braces `{ ... }` and separated from the next by a coma
- each entry contains a list of key–value pairs separated by comas, in an arbitrary order
- key and value are separated by a colon `:`
- the key is always a string between double quotation marks `"..."`
- the value may be a string between double quotation marks, or one of the Boolean keywords *true* and *false*
- inside each string, the backslash \ and the double quotation marks `"` must be preceded (“escaped”) by a backslash; line break and tabulation are obtained with n and t respectively; the notation uDDDD (where each D represents a digit) is accepted for Unicode characters.
- Certain keys have a default value and are thus optional; the others are compulsory.

File list (Text Files widget)

The keys (and associated values) for the file lists are the following:

Key	Type	Default	Value	Remark
path	*string*	—	*file path (absolute or relative)*	*be careful to escaping the backslash*
encoding	*string*	—	*file encoding*	*cf.* Python doc (codecs)
annotation_key	*string*	—	*annotation key*	—
annotation_value	*string*		*annotation value*	—

Example:

```
[
    {
        "path":               "data\\Balzac\\Eugenie_Grandet.txt",
```

(continues on next page)

(continued from previous page)

```
        "encoding":          "iso-8859-1",
        "annotation_key":    "auteur",
        "annotation_value": "Balzac"
    },
    {
        "path":              "data\\Balzac\\Le_Pere_Goriot.txt",
        "encoding":          "iso-8859-1",
        "annotation_key":    "auteur",
        "annotation_value": "Balzac"
    },
    {
        "path":              "data\\Daudet\\Lettres_de_mon_moulin.txt",
        "encoding":          "iso-8859-15",
        "annotation_key":    "auteur",
        "annotation_value": "Daudet"
    },
    {
        "path":              "data\\Daudet\\Tartarin_de_Tarascon.txt",
        "encoding":          "iso-8859-15",
        "annotation_key":    "auteur",
        "annotation_value": "Daudet"
    }
]
```

URL list (URLs widget)

The keys (and associated values) for the URLs lists are the following:

Key	*Type*	*Default*	*Value*	*Remark*
url	*string*	—	*file url (absolute)*	*be careful to include the indication* `http://`
encoding	*string*	—	*file encoding*	*cf.* Python doc (codecs)
annotation_key	*string*	—	*annotation key*	—
annotation_value	*string*		*annotation value*	—

Example:

```
[
    {
        "url":               "http://www.imsdb.com/scripts/Alien.html",
        "encoding":          "iso-8859-1",
        "annotation_key":    "genre",
        "annotation_value": "sci-fi"
    },
    {
        "url":               "http://www.imsdb.com/scripts/Pulp-Fiction.html",
        "encoding":          "iso-8859-1",
        "annotation_key":    "genre",
        "annotation_value": "crime"
    }
]
```

Substitution list (Recode widget)

The keys (and associated values) for the file lists are the following:

Key	Type	Default	Value	Remark
regex	*string*	—	*regular expression*	*be careful to escape the slashes and backslash*
replacement_string	*string*	*replacement string*	—	
ignore_case	*Boolean*	*false*	*option -i*	*cf.* Python doc (re.UNICODE)
multiline	*Boolean*	*false*	*option -m*	*cf.* Python doc (re.MULTILINE)
dot_all	*Boolean*	*false*	*option -s*	*cf.* Python doc (re.DOTALL)
unicode_dependent	*Boolean*	*false*	*option -u*	*cf.* Python doc (re.IGNORECASE)

Example:

```
[
    {
        "regex":                  "<.+?>",
        "replacement_string":  ""
    },
    {
        "regex":                  "(behavi|col|neighb)our",
        "replacement_string":  "&1or",
        "ignore_case":          true,
        "unicode_dependent":   true
    },
    {
        "regex":                  "a (\\w+) of mine",
        "replacement_string":  "my &1",
        "unicode_dependent":   true
    }
]
```

Regular expression list (Segment widget)

The keys (and associated values) for the file lists are the following:

Key	Type	Default	Value	Remark
mode	*string*	—	*"split" or "tokenize"*	—
regex	*string*	—	*regular expression*	*be careful to escape the backslash*
ignore_case	*Boolean*	*false*	*option -i*	*cf.* Python doc (re.UNICODE)
multiline	*Boolean*	*false*	*option -m*	*cf.* Python doc (re.MULTILINE)
dot_all	*Boolean*	*false*	*option -s*	*cf.* Python doc (re.DOTALL)
unicode_dependent	*Boolean*	*false*	*option -u*	*cf.* Python doc (re.IGNORECASE)
annotation_key	*string*	—	*annotation key*	—
annotation_value	*string*		*annotation value*	—

Example:

```
[
    {
        "mode":                 "Tokenize",
        "regex":                ".",
        "dot_all":              true,
        "annotation_key":       "type",
        "annotation_value":     "other"
    },
    {
        "mode":                 "Tokenize",
        "regex":                "\\w",
        "ignore_case":          true,
        "unicode_dependent":    true,
        "annotation_key":       "type",
        "annotation_value":     "consonant"
    },
    {
        "mode":                 "Tokenize",
        "regex":                "[aeiouy]",
        "ignore_case":          true,
        "annotation_key":       "type",
        "annotation_value":     "vowel"
    },
    {
        "mode":                 "Tokenize",
        "regex":                "[0-9]",
        "annotation_key":       "type",
        "annotation_value":     "digit"
    }
]
```