FROM WORDS TO WISDOM

An Introduction to Text Mining with KNIME®

Vincenzo Tursi and Rosaria Silipo

This book has been updated for KNIME 3.7.

For information regarding permissions and sales, write to:

KNIME Press
Technoparkstr. 1
8005 Zurich
Switzerland

knimepress@knime.com

ISBN: 978-3-9523926-2-1

Table of Contents

Foreword

From scientific papers, over Wikipedia articles, patents, tweets, to medical case reports, and product reviews, textual data is generated and stored in various areas, to document, educate, tell, influence, or simply to entertain. Not only the amount of textual data is growing massively every year, also the areas in which text is generated and can be mined are increasing.

Due to the complexity of human natural language and the unstructured and sequential nature of the data, it is especially complex to mine and analyze text. In order to handle this complexity, specific methods have been invented in the fields of text mining and natural language processing. Whereas pure text mining is focusing on the extraction of structured knowledge and information from text, natural language processing is approaching the problem of the understanding of natural language.

Many of these methods and algorithms have been implemented in a variety of tools and platforms. For example, important open source libraries are Stanford NLP and Apache OpenNLP as well as packages in R and Python. Both of them, Stanford NLP and Apache OpenNLP, are integrated in the KNIME Text Processing extension. Due to the visual programming paradigm of KNIME, the Text Processing extension enables also non-programmers and non-scripters, not only to use those libraries, but also to easily combine them with a variety of other functionalities.

Still, text mining is not an easy task, even with the right tool. Text processing functionality needs to be well understood and correctly used, before applying them. This is why this book will prove to be extremely helpful. Also the timing for the book release is perfect: The KNIME Text Processing extension was moved out of KNIME Labs* recently, with the release of the KNIME Analytics Platform version 3.5.

Rosaria and Vincenzo have done an outstanding job writing this truly comprehensive book describing the application of text mining and text processing techniques via the KNIME Text Processing extension in combination with other KNIME Analytics Platform data science resources.

Kilian Thiel

* KNIME Labs category in KNIME Analytics Platform is dedicated to advanced and not yet fully established data science techniques.

Acknowledgements

When writing a book, it is impossible not to ask and learn from a few people. That was the case for this book as well. So, here it is our chance to thank all those people who taught us more about text mining, who provided us with some level of technical support, who gave us interesting ideas, and, in general, who have stood us through these last few months. Here they are.

First of all, we would like to thank Kilian Thiel for explaining how a few mysterious nodes are working. Kilian, by the way, was the developer zero of the KNIME Text Mining extension.

We would like to thank Heather Fyson for correcting our writing and, especially, for anglicizing our English from the strong Italian influences.

Frank Vial is responsible for exactly four words in this book: the title.

Finally, a word of thanks to Kathrin Melcher and Adrian Nembach who provided precious help for the neural network and deep learning part.

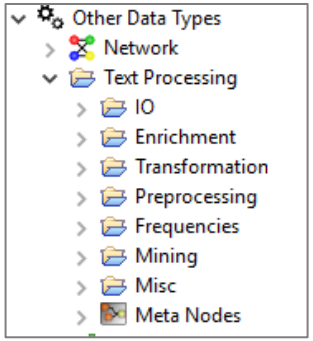
Chapter 1. Introduction

1.1. Why Text Mining?

We often hear that we are in the age of data [1], that data may become more important than software [2], or that data is the new oil [3], but much of these data are actually texts. Blog posts, forum posts, comments, feedbacks, tweets, social media, reviews, descriptions, web pages, and even books are often available, waiting to be analyzed. This is exactly the domain of text mining.

KNIME Analytics Platform offers a text processing extension, fully covering your needs in terms of text analytics. This extension relies on two specific data objects: the Document and the Term.

Figure 1.1. Node folders in the Text Processing extension from the Node Repository panel



A Document object is not just text, but it also includes the text title, author, source, and other information. Similarly a Term is not just a word, but it includes additional information, such as its grammar role or its reference entity.

The KNIME Text Processing extension includes nodes to read and write Documents from and to a variety of text formats; to add word information to Terms; to clean up sentences from spurious characters and meaningless words; to transform a text into a numerical data table; to calculate all required word statistics; and finally to explore topics and sentiment.

The goal of this book is to explore together all steps necessary and possible to pass from a set of texts to a set of topics or from a set of texts to their in between the lines sentiments.

1.2. Install the KNIME Text Processing Extension

The KNIME Text Processing extension, like all KNIME extensions, can be installed within the KNIME Analytics Platform from the top menu items:

- File -> Install KNIME Extensions …

Or

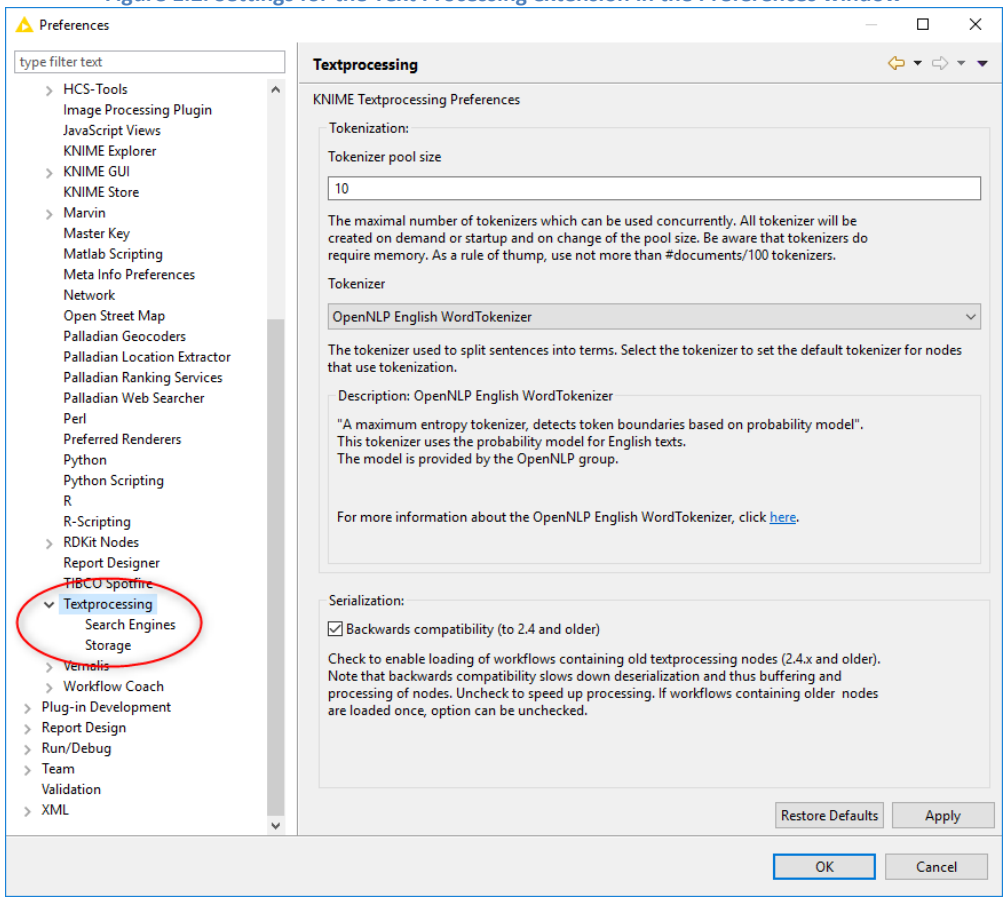
- Help -> Install New Software …

Both menu items open to the “Install” window.

- In the text box labelled “Work with:” connect to the KNIME Analytics Platform Update Site (i.e. ‘http://update.knime.com/analytics-platform/3.5’ for KNIME Analytics version 3.5);
- Expand item “KNIME & Extensions” and select extension “KNIME Text Processing” and the language packs you wish to use;
- Click “Next” and follow the installation instructions.

If installation has been successful, you should end up with a category Other Data Types/Text Processing in the Node Repository panel. No additional installation is required, besides downloading occasional dictionary files for specific languages. Usually such dictionary files can be found at academic linguistic departments, like for example at the WordNet site.

Figure 1.2. Settings for the Text Processing extension in the Preferences window



After the installation of the KNIME Text Processing extension, you can set a few general preferences for the Text Processing nodes.

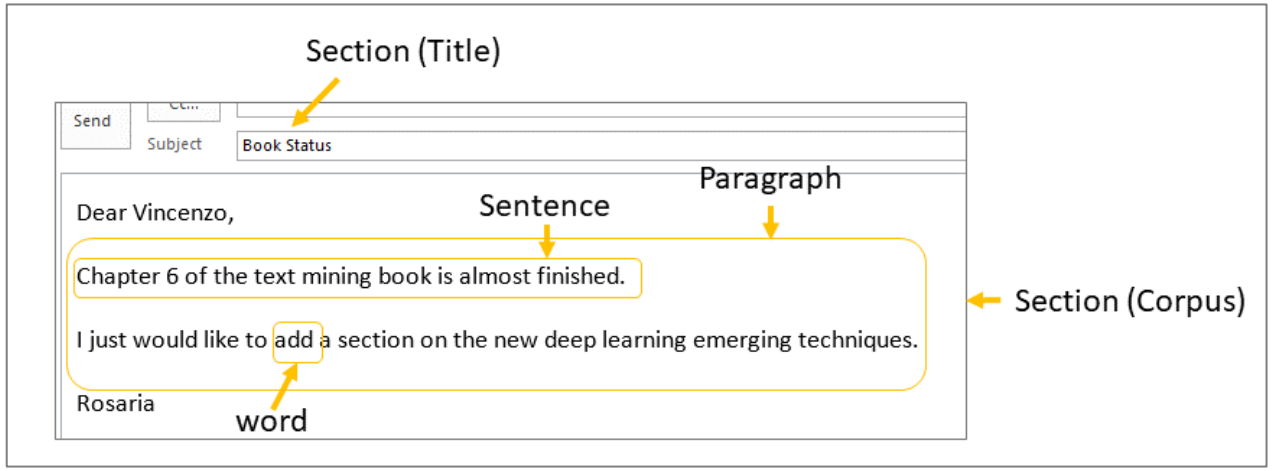
Under Preferences -> KNIME -> Text Processing, you can set the tokenizer properties. Here you can also set how to store text data cells and, in case of file based storage, the chunk size; that is the number of Documents to store in a single file. Finally, you can define the list of search engines appearing in the Document view, allowing the search for meaning or synonyms.

1.3. Data Types for Text Processing

Nodes in the KNIME Text Processing extension relies on two new types of data: **Document**and **Term**.

A raw text becomes a Document when additional metadata, such as title, author(s), source, and class, are added to the original text. Text in a Document gets tokenized following one of the many tokenization algorithms available for different languages. Document **tokenization** produces a hierarchical structure of the text items: sections, paragraphs, sentences, and words. Words are often referred to as tokens. Below you can see an example of the hierarchy produced by the tokenization process applied to an email.

Figure 1.3. Tokenization items in an email Document



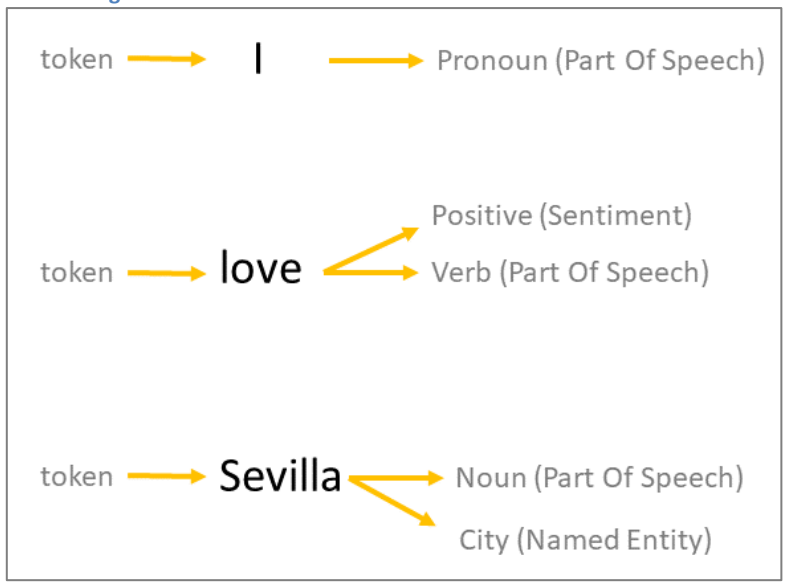
Similarly to the Document object, a token becomes a Term with the addition of related metadata, and specifically tags. Tags describe sentiment, part of speech, city (if any), person name (if any), etc … covered by the word in the Term. Below you can see a few Term examples from the sentence “I love Sevilla”.

Term “I” includes token (word) “I” and it’s Part Of Speech = “Pronoun”.

Term “love” includes token (word) “love”, Part Of Speech = “Verb”, and Sentiment = “Positive”.

Term “Sevilla” includes token (word) “Sevilla”, Part Of Speech = “Noun”, and Named Entity = “City”.

Figure 1.4. Term structure from the sentence “I love Sevilla”.



1.4. The Text Mining Process

The whole goal of text data preparation is to convert the text into numbers, as to be able to analyze it with all available statistical and machine learning techniques.

The process always starts with ***text reading***, whatever the text format is.

After that, we transform the simple text String into a more complex Document object. For this transformation, a ***tokenization*** operation is required. Tokenization algorithms identify and label parts of the input texts as sections, paragraphs, sentences, and terms. Once all those text parts have been detected, labelled, and stored, the Document object is born.

After defining the hierarchical structure of the Document, it is possible to attach specific tags to some terms, such as grammar roles (Part Of Speech, POS), sentiment, city names, general entity names, dictionary specific tags, and so on. This tagging operation is named ***enrichment***, since it enriches the information content of the Term.

Now that we have tokenized the text down to Terms and that we have included extra information in some of the Terms, if not all, we can proceed with more aggressive ***clean up***. The main goal of the cleanup phase is to get rid of all those words carrying too little information. For example, prepositions and conjunctions are usually associated with grammar rules, rather than with semantic meaning. These words can be removed using:

- A tag filter, if a POS tagging operation has been previously applied;
- A filter for short words, i.e. shorter than N characters;
- A filter for stop words, specifically listed in a dictionary file.

Numbers could also be removed as well as punctuation signs. Other ad hoc cleaning procedures could also help to make the Document content more compact. Cleanup procedures usually go together with other generic pre-processing steps.

A classic pre-processing step consists of ***stemming***, i.e. of extracting the word stem. For example, the words "promising" and "promise" carry the same meaning in two different grammar forms. With a stemming operation, both words would be reduced to their stem "promis[]". The stemming operation makes the word semantic independent of the grammar form.

Now we are ready to collect the remaining words in a ***bag of words*** and to assign a frequency based score to each one of them. If the words in the bag of words are too many, even after the text cleaning, we could consider the option of summarizing a Document through a set of ***keywords***. In this case, all words receive a score, quantifying their summary power, and only the top n words are kept: the n keywords. Words/keywords with their corresponding score pass then to the next phase: Transformation.

Figure 1.5. The many phases of a Text Analytics process

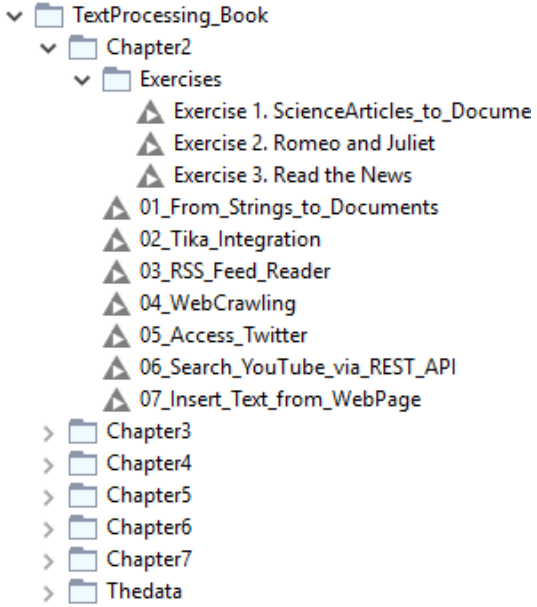
Transformation covers ***encoding*** and ***embedding***. Here the Document moves from being represented by a set of words to being represented by a set of numbers. When using encoding we refer to the presence (1) / absence (0) of a word in a Document text: 1 if the word is present, 0 if it is absent. We then define a matrix where each word gets a dedicated column and each Document is represented by a sequence of 0s and 1s, depending on the presence/absence of each column word in the Document. Instead of 1s the frequency based score of the word could also be used. Embedding is another way of representing words and Documents with numbers, even though the number sequence is in this case not interpretable.

Finally, the application of Machine Learning techniques, generally available for data analytics or specifically designed for text mining, allows us to discover ***sentiment*** and ***topics*** hidden in the text.

1.5. Goals and Organization of this Book

The goal of this book is to give an overview of the whole text mining process and of how to implement it in KNIME Analytics Platform.

Figure 1.6. Workflow structure after importing the Download Zone .knar file



We will start of course with importing texts from various sources and in different formats. Chapter 2 is completely dedicated to this topic, including text files, kindle files, social media channels, access to REST APIs, web crawling, and text from forms in web pages. Then in chapter 3 we will cover text-processing techniques: tagging, filtering, stemming, and bag of words extraction. Chapter 4 is dedicated to frequency measures, keyword extraction, and corresponding score calculation.

The first exploratory phase in any data analytics project consists of data visualization, and text analytics is no exception. In chapter 5 the most commonly used text visualization techniques are described. Chapter 6 finally moves to Machine Learning and statistical algorithms for topic detection and classification, while chapter 7 uses Machine Learning algorithms for sentiment analysis.

This book comes with a set of example workflows and exercises. Indeed, when you bought this book you should have received an email with a link to the Download Zone. The Download Zone is just a KNIME file (extension **.knar**) containing all workflows you need to follow the learning path of this book. Import the .knar file into KNIME Analytics Platform, either via double-click the file or via menu option "File" -> "Import KNIME Workflow" or via right-click LOCAL workspace in KNIME Explorer panel and then "Import KNIME Workflow".

If the import is successful, you should find in the KNIME Explorer panel a workflow group named TextProcessing_Book with the structure shown in figure 1.6.

The subfolder named "Thedata" contains all data sets used in the following chapters. Each workflow group, named "Chapter …", contains the example workflows and the exercise workflows for that chapter.

If you are a novice to KNIME Analytics Platform, you will not find much of the basics in this book. If you need to know how to create a workflow or a workflow group or if you still need to know how to create, configure, and execute a node, we advise you to read the first book of this series "KNIME Beginner's Luck" [4].

There are a few more resources on the KNIME web site about the Text Processing extension.

- Text Processing extension documentation page https://www.knime.com/documentation-3
- Text Processing examples and whitepapers https://www.knime.com/examples
- Text Mining courses regularly scheduled and run by KNIME https://www.knime.com/courses

A number of example workflows can also be found in KNIME EXAMPLES server, at the top of the KNIME Explorer panel, under 08_Other_Analytics_Types / 01_Text_Processing.

Chapter 2. Access Data

2.1. Introduction

As every data analytics project, a text analytics project also starts with reading data. Data in this case are not numbers but texts, or better enriched texts including additional information like title, authors, and other metadata. The object in KNIME Analytics Platform containing a text and its metadata is a Document. In this chapter we will explore a few nodes that can access texts and/or transform them into Documents.

As data sources go, many are already available at the moment of writing and probably many more will be available in the next years. We have chosen to focus on just a few ones, the most commonly used.

Classic text files will never go out of fashion and therefore it is good to know how to read and import them into KNIME Analytics Platform and how convert them into Documents. Here you have two options.

- You read a dataset containing texts using the appropriate reader node and then convert them into Documents with the Strings To Document node
- You rely on the Tika Integration nodes, to read and parse one or more text files in a number of format, and on the Strings To Document node, to convert them into Documents.

Another important source of text data is the web, with all its web pages, posts, journal articles, forums, and much more. In this chapter we will focus on two ways to import data from the web: via RSS Feed and via web crawling.

Another very important source of text data are the social media channels. There are many social media channels and growing. For some of them KNIME Analytics Platform offers dedicated connectors, which simplify the connection process considerably, while, for some others, no dedicated connector is available. However, most of them are reachable via REST API, provided that we hold all the required authorization keys and tokens. We cannot explore here how to access all of the social media channels. We then choose two social media channels as examples: Twitter and YouTube metadata.

We choose Twitter, because a dedicated connector is available and we want to show how to use dedicated connectors to social media. We choose YouTube metadata because no dedicated connector is available, it is however accessible via REST API, and we want to show how it is possible to access any channel via REST API. In particular, the YouTube REST API belongs to the Google REST API family and therefore it is representative of a very large set of standard REST services.

If this chapter leaves any questions unanswered about connecting to other data and text sources, you can always refer to the e-book "Will they blend? The blog post collection" [5]. This e-book contains all blog posts of the "Will they blend?" series, all dealing with data access problems, even the weirdest ones. The e-book is free and can be downloaded from the KNIME Press at the following link https://www.knime.com/knimepress/will-they-blend.

2.2. Read Text Data and Convert to Document

All nodes from the KNIME Text Processing extension work on Document type data columns. A Document type contains much more than just a text. It contains the tokenized text and the document meta-information, such as Title, Author, etc… Document tokenization produces a hierarchical structure of the text items: sections, paragraphs, sentences, and words. A text has been tokenized when it has been split down to words (tokens). To properly identify tokens within a text you need a Natural Language Processing (***NLP***) model or more precisely an NLP Tokenizer.

Many NLP tokenizers are available. The simplest tokenizer of all is the ***OpenNLP Simple Tokenizer***, which assumes that a sequence of the same type of characters, like alphabetical characters, is a token. As simple as it is, it works on all languages.

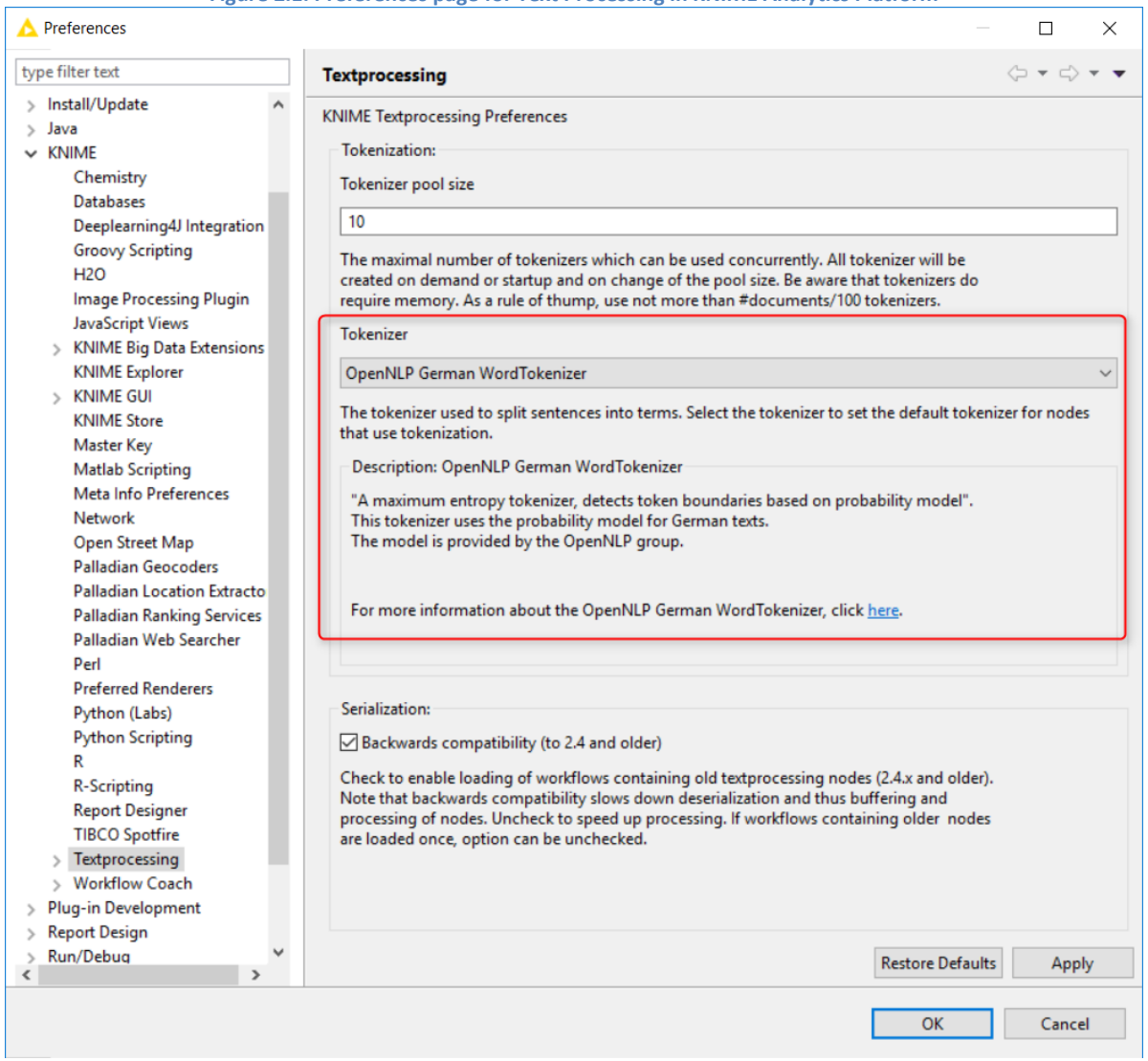
A slightly more sophisticated tokenizer is the ***OpenNLP Whitespace Tokenizer***, which assumes that all non-space characters between two space characters make a token. This approach though does not apply to most Asian written languages, like for example Chinese.

If we are willing to climb the complexity scale and take into account grammar, rules, and probabilities of the language at hand, we can move to Tokenizers for English (***Stanford NLPTBTokenizer***, ***OpenNLPEnglishWordTokenizer***), Spanish (***SpanishNLPStanfordTokenizer***), German (***OpenNLPGermanWordTokenizer***), and Chinese (***StanfordNLPChineseTokenizer***).

All those tokenizers are available in the KNIME Text Processing Extension. You can find more information on each one of those tokenizers under *File -> Preferences -> KNIME -> Textprocessing*.

If you read a text file, with a File Reader, a CSV Reader, a Table Reader, or any other Reader node, you will likely end up with a String type column for your text. All text processing nodes from the KNIME TextProcessing extension work on Document type columns. Thus, the question for this section is: how do I convert my text from a String type to a Document type? Even more precisely: how do I transform a number of different String values, including text, author, title, etc …, into a Document object? The answer is: I use a Strings to Document node.

Figure 2.1. Preferences page for Text Processing in KNIME Analytics Platform



Strings To Document

The Strings To Document node collects values from different columns and turns them into a Document object, after tokenizing the main text. The node requires the following settings:

- The Document Title, as column value, RowID, or empty String
- The column containing the main text

- The Document source, as fixed String for all rows or as column values. If column values must be used, remember to enable the corresponding flag

- The Document category, as fixed String for all rows or as column values. If column values must be used, remember to enable the corresponding flag. Often the Document category contains the Document class.

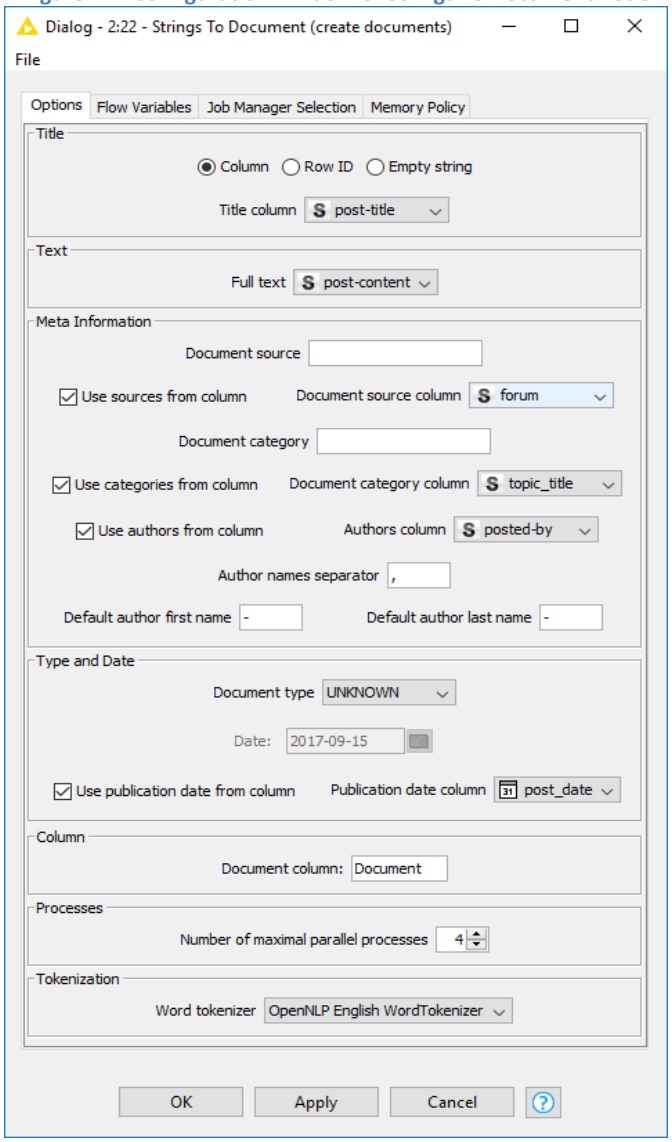
- The Document author(s), as fixed String ("Default author first/last name") for all rows or as column values. If column values must be used, remember to enable the corresponding flag.

- The Document publication date, as fixed String for all rows or as column values. If column values must be used, remember to enable the corresponding flag.

- The Document type, as Transaction, Proceeding, Book, or just unknown. "UNKNOWN" fits most of the input data.

- The name of the output Document column. Default name is "Document".
- The maximum number of parallel processes to execute the Word Tokenizer
- The Word Tokenizer algorithm.

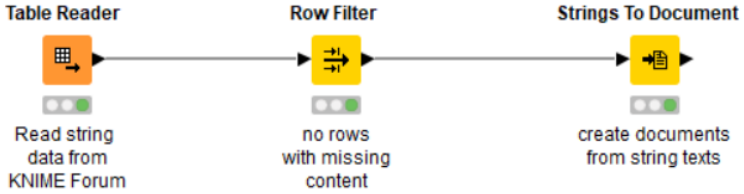
Figure 2.2. Configuration window of Strings To Document node



As an example, we read the table file in *knime://knime.workflow/../../Thedata/ForumData_2013-2017.table*. This file contains all posts and comments from the KNIME Forum published between May 31st 2013 and end of May 2017. In particular, it contains: the original forum source, the post title, the post URL, the post content, the number of replies, the author username, the publishing date, and a few additional less interesting columns.

After removing the data rows with empty posts, we applied the Strings To Document node to collect all those columns' values into a single Document object with tokenized content. After execution, the Strings To Document node presents a new column in the output table, named Document, of type Document, and containing a Document object with post title, post tokenized text, post author, and post date for each data row.

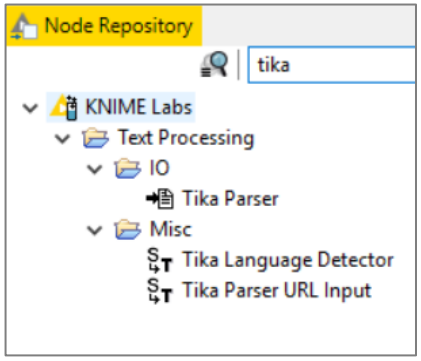
Figure 2.3. Transforming String values into a Document object for each data row in file Forumdata_2013-2017.table



Note. If you are working with a language for which there is no dedicated tokenizer in the Strings To Document node, then use the generic *OpenNLPSimpleTokenizer* or, if it is a language that uses whitespaces to separate words, the Open *NLPWhitespaceTokenizer*.

2.3. The Tika Integration

Figure 2.4. Tika Nodes in the Node Repository



If you have a document to read in Word, pdf, PowerPoint, Kindle, or some other format, the Tika Parser node is likely the node you need.

The Apache Tika library provides tools to parse and detect document metadata and text from over a thousand different file types (such as PPT, XLS, PST, EPUB, PDF, and many more). In particular, the Tika Parser and Language Detector functions have been integrated in KNIME.

Let's try to read a pdf document, like for example the product sheet of the KNIME Analytics Platform. The pdf file, named KNIME_Analytics_Platform_Product_sheet.pdf, is available in Thedata folder that comes with the workflows of this book. To read and parse this document we can use the Tika Parser node or the Tika Parser URL Input node.

Now what does parsing exactly mean when importing a pdf document? The Tika parsing function refers to the document metadata; that is: Filepath, MIME-type, Author, Title, Content, Author(s), Contributor(s), Description, Date Modified, Publisher, Print Date, and a few more.

Tika Parser

The Tika Parser node searches a folder for specific document types (ppt, doc, xls, epub, pdf, and many more) and imports and parses all found files.

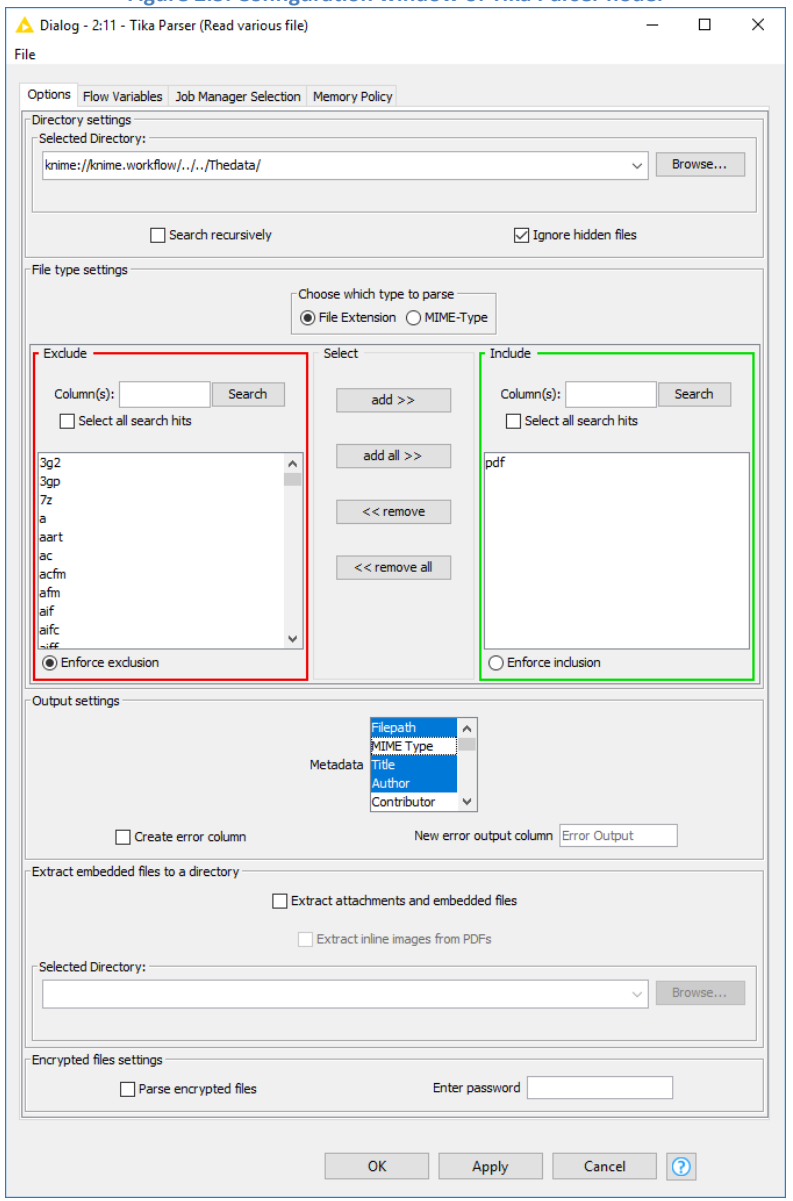
- "Selected Directory" setting defines the source folder
- "Search recursively" includes subfolders in the document search and import
- "Ignore hidden files" allows the search to skip hidden files even if they are of the selected type.

- The selected radio button in box "Choose which type to parse" restricts the search to MIME-type file or to file extensions.
- Based on the previous selection (MIME or file type) the Include/Exclude framework selects one or more file/MIME types to import. Use buttons "add", "add all", "remove", and "remove all" to move file extensions / MIME types from the Include to the Exclude frame and vice versa.

- The "Metadata" list below the Include/Exclude framework contains the full list of metadata available in the original document. Here you can select the metadata of interest to import from the document.
- "Create error column" and "New error output column" allow to store possible parsing errors in a separate column in the output data table.

- The last part of the configuration window is reserved for attachments and images. There you can select whether to extract attachments and/or images and where to store the generated files. If attachments are encrypted, you can still import them by entering the required password.

Figure 2.5. Configuration window of Tika Parser node.



We chose to search the Thedata folder for PDF document files, not recursively, ignoring hidden PDF documents and possible parsing errors, and to extract the following metadata: Filepath, Author, Title, and Content.

The Tika Parser node produces 2 output tables. The top output table, named “Metadata output table”, contains the metadata resulting from the document parsing. The lower output table, named “Attachment output table”, contains the paths to attachments and images, if any.

Note. In the “Selected Directory” field we chose to use a relative path rather than an absolute path. The knime:// protocol, indeed, allows for the definition of relative paths. *knime://LOCAL*, for example, refers to the LOCAL workspace folder; ***knime://knime.workflow*** refers to the current workflow folder; and *knime://<name of server mountpoint>* refers to the default workspace folder of the specified connected KNIME Server.

If we execute the Tika Parser node now, all pdf documents in folder Thedata are detected and their metadata are extracted: Filepath, Title, Author, and most important of all Content. The data table at the top output port contains as many columns as many metadata and as many data rows as many documents found in the selected folder.

Note. All resulting columns in the output data table are of type String. You need a Strings To Document node to convert the text results into a Document type column.

In our case, folder Thedata contains only one PDF file, named KNIME_Analytics_Platform_Product_sheet.pdf. The imported metadata from this file at the top output port of the Tika Parser node are shown in figure 2.6.

The same result could have been obtained using the “Tika Parser URL Input” node. The Tika Parser URL Input node implements exactly the same parsing functionality as the Tika Parser node, using even the same configuration window. However, while the Tika Parser node requires no input and the folder path is specified as a configuration setting, the Tika Parser URL Input node requires a String input URL column containing the file paths.

In order to get the same results as using the Tika Parser node, we feed the Tika Parser URL Input node with the output table of a Table Creator node, where we wrote one file URL path. The workflow illustrating how to use the two Tika Parser nodes is stored in Chapter2/02_Tika_Integration and shown in figure 2.7.

Figure 2.6. Metadata from file KNIME_Analytics_Platform_Product_sheet.pdf available at the upper output port of the Tika Parser node: Filepath, Title, Author, and Content.

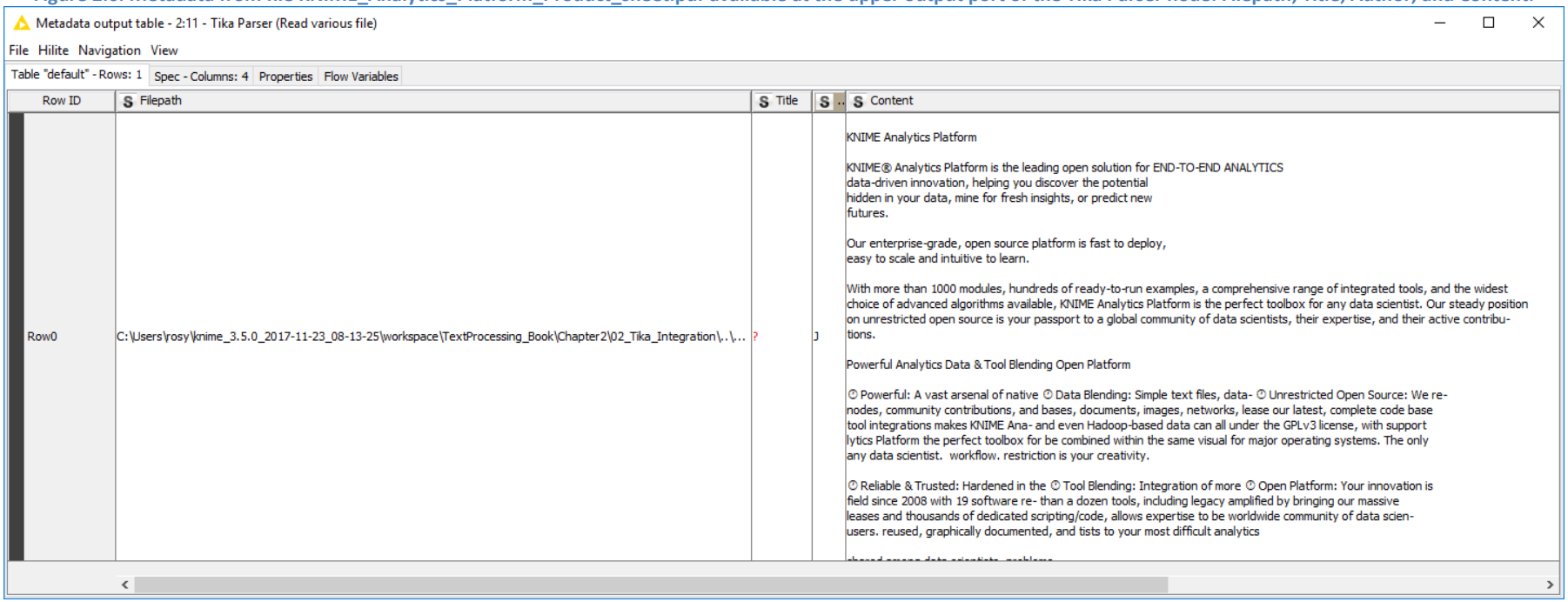
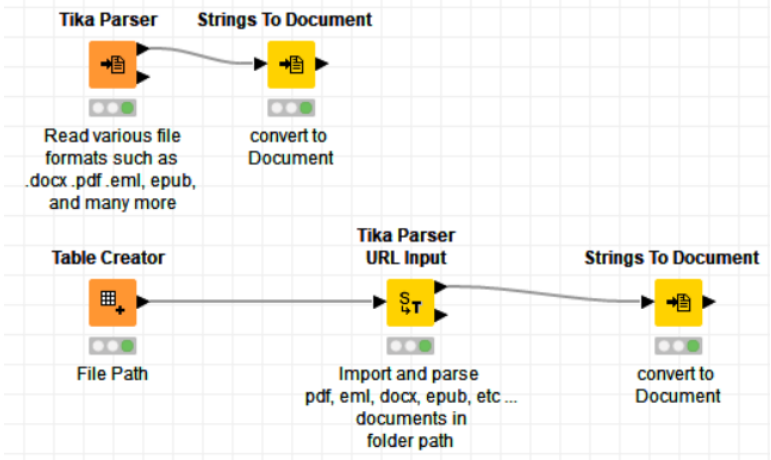


Figure 2.7. The two Tika Parser nodes used to read pdf file KNIME_Analytics_Platform_Product_sheet.pdf. The Tika Parser node searches and imports pdf files in the folder specified by the path in its configuration window. The Tika Parser URL Input node imports the pdf files specified at its input port.



2.4. Access Data from the Web

A big source of text documents is of course the Web: journal articles from RSS Feeds, web pages, feedbacks and comments, entire books, pdf documents, and, for sure, much more. In this chapter we describe some possible approaches to download documents from the web: getting news through an RSS feed, crawling web pages, and connecting to social media channels.

2.4.1. RSS Feeds

To import data from an RSS feed, we can use the RSS Feed Reader node. The RSS Feed Reader node takes an RSS Feed URL at the input port, connects to the RSS endpoint, and finally downloads and parses all available documents.

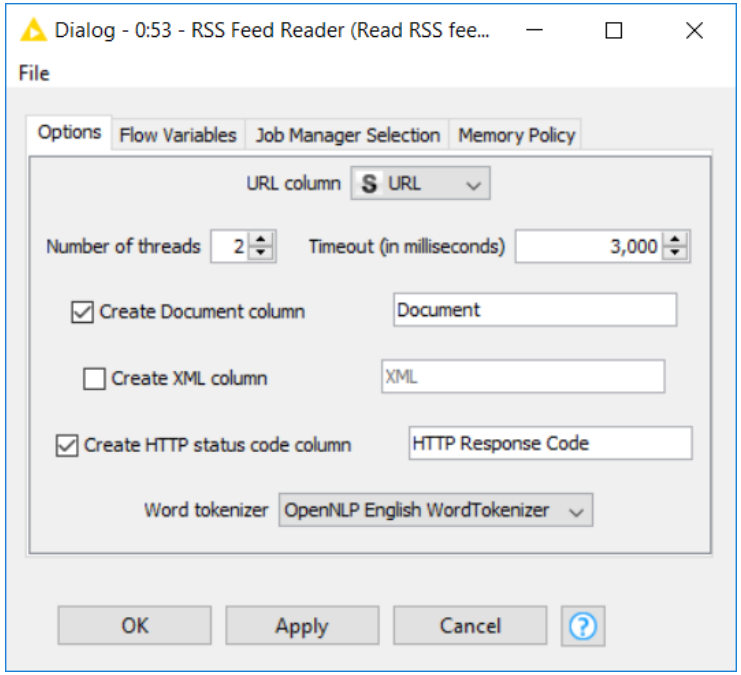
RSS Feed Reader

The RSS Feed Reader node connects to the URL of an RSS Feed, imports and parses the available documents, and presents the results at the output port. The configuration window of the RSS Feed Reader requires:

- The input column containing the RSS Feed URL(s)
- The number of parallel threads to use for faster processing
- A time out in ms to guarantee against endless waiting if the RSS Feed host for whatever reason does not respond
- Whether to create a Document type or an XML type or both type output column(s) containing the RSS Feed documents
- Whether to create an output column containing the HTTP Status of the request
- The word tokenizer algorithm. In fact, in order to produce the Document column in the output data table, the extracted text needs to be tokenized via a word/sentence tokenizer algorithm.

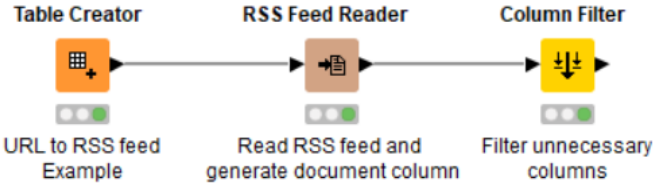
There are many word tokenizer algorithms available in this node, including different algorithms for different languages. Just go to *File -> Preferences -> KNIME -> Textprocessing* to read the description of the available word tokenizers.

Figure 2.8. Configuration window of the RSS Feed Reader node



Thus, a workflow to collect documents from an RSS Feed looks similar to the one in figure 2.9, where the Table Creator prepares a list of RSS Feed URL sources and passes them to the RSS Feed Reader node.

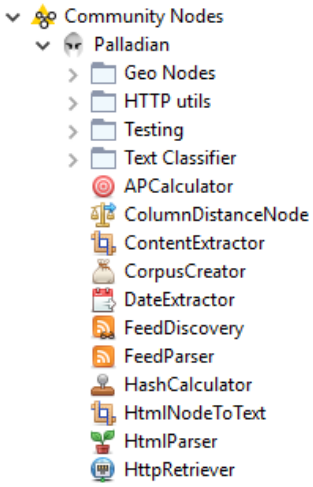
Figure 2.9. Workflow to extract available documents from an RSS Feed endpoint



In our example, we connect to the RSS Feed of the New York Times journal http://rss.nytimes.com/services/xml/rss/nyt/US.xml. The URL is defined in column "URL" of the Table Creator node and passed to the input port of the RSS Feed Reader node. We decided to create a Document column for more text processing later on and to use the "OpenNLP English WordTokenizer" algorithm.

2.4.2. Web Crawling

Figure 2.10. Palladian category in Node Repository



Another way to retrieve text data from the web is to run a web crawler. No worries! You do not need to leave the KNIME Analytics Platform to run a web crawler. Indeed, one of the KNIME Community Extension provided by Palladian offers a large number of nodes for web search, web crawling, geo-location, RSS feed, and many more. In this section we will explore only two of these many nodes: the HttpRetriever and the HtmlParser node. Those are the only two nodes we need to implement a web crawler. They are located in *Palladian/HTTP utils* sub-category.

The HttpRetriever node takes a URL as input and downloads all web content available at the selected URL. The first output is an HTTP-type column, named Result and containing the download status code and the downloaded content. The second output port contains the cookies set during the node execution, if any. Optionally, it is possible to provide existing cookies at the second input port.

The HtmlParser node imports and parses the HTTP-type object named Result and generated by an HttpRetriever node. At its output port the content of the HTTP column is presented in the form of an XML object.

After executing an HttpRetriever node and an HtmlParser node, the content of the web page has been downloaded and imported into KNIME as an XML cell. We now need to extract the title, the author, the full content, and more metadata from the original web page we have crawled.

HttpRetriever

This node allows to download content from a web page via a number of different HTTP methods: GET, POST, HEAD, PUT, and DELETE. The node then imports the web content and the cookies as binary data and allows to specify arbitrary HTTP headers.

The configuration window is organized on three tabs.

Tab "Options" requires:

- The data column with the URLs of the pages to connect to.
- Each URL can require a different download method. If this is the case, you can set the value of an input column as download method for each URL. If no such column is provided, all contents are downloaded via GET.
- The HTTP entity column, and optionally type, can be specified, if an HTTP entity is required for the download to start.
- An emergency stop criterion on the download file size
- Whether to append an output column with the download status

Tab "Headers"

If HTTP headers are required for the transfer, the column(s) containing HTTP Headers are set in this tab.

Tab "Advanced" requires:

- the settings for stopping the execution in case of errors or connection problems (# retries and timeouts)
- the HTTP user agent for the data transfer
- a tolerance flag about accepting SSL certificates (this is of course a sensitive security option)
- Force an error on the node execution, if the HTTP status code for the data transfer returns some kind of failure

Figure 2.11. Configuration window of the HttpRetriever node: "Options" tab

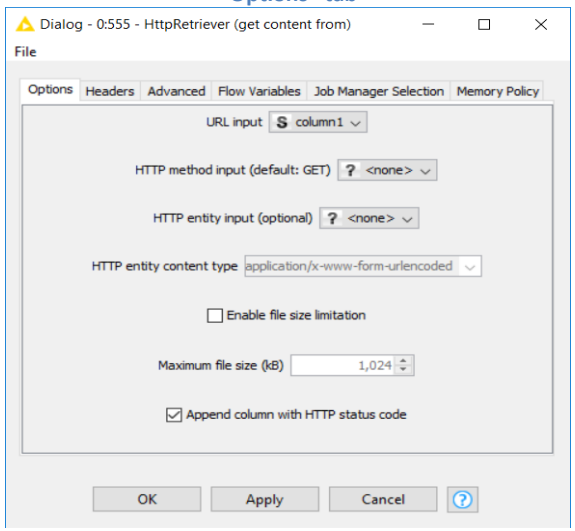
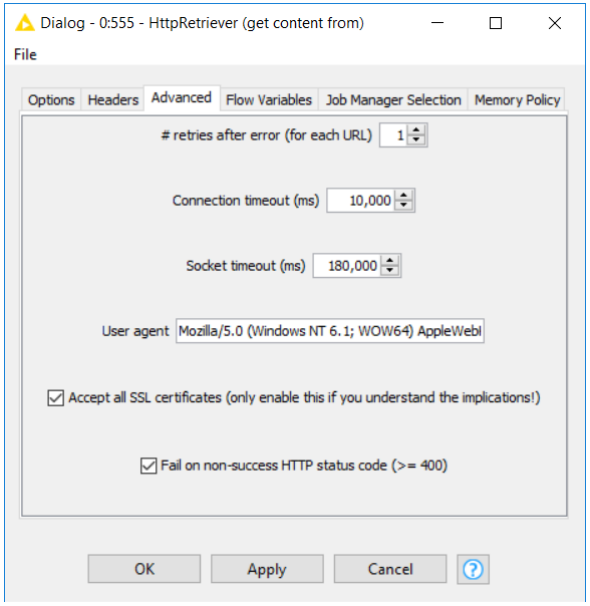


Figure 2.12. Configuration window of the HttpRetriever node: "Advanced" tab



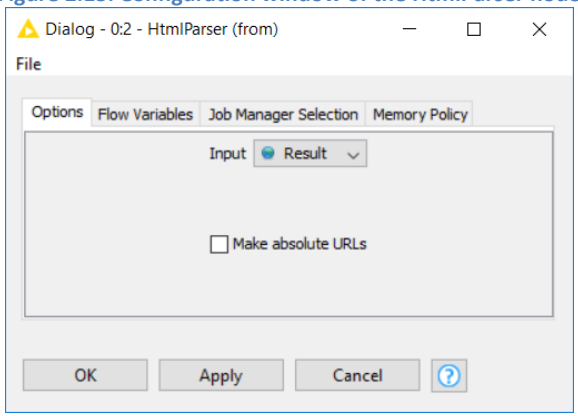
HtmlParser

The HtmlParser node transforms an HTTP object into an XML object. To perform this task, it relies on the HTML Parser available in the Validator.nu Java library for HTML5.

The node does not need much in terms of configuration settings, just the name of the HTTP column at its input port to be transformed into an XML type column.

The option “Make absolute URLs”, when enabled, converts all relative URLs in the document to absolute URLs.

Figure 2.13. Configuration window of the HtmlParser node



Note. To install the Palladian Community Extension, go to *File -> Install KNIME Extensions*; in the “install” window, open *KNIME Community Contribution – Other* and select *Palladian for KNIME*; then hit the “Next” button and follow the installation instructions. If installation is successful, you will find the Palladian category in the Node Repository full of nodes for all your web access needs.

As an example, for this section we want to download the KNIME Manifesto page "KNIME for Data Scientists". First, we write the page URL (https://www.knime.com/knime-for-data-scientists) in a Table Creator node. We then feed an HttpRetriever node with this URL. During execution, the HttpRetriever node connects to the web page, downloads its content, and presents it at its output port. The page is then parsed using an HtmlParser node. The output of the HtmlParser node exhibits the page content as an XML-structured object.

Now, we need to create a Document type column including: author, title, and full content from the original web page. We extract such information from the XML cell using one or more XPath nodes. In particular, we use a first XPath node to extract the title and the content and then the author from the imprint at the end of the page.

Notice that while author and title can easily be extracted as String objects, the full content needs some cleaning from residual HTML code. Thus, a second XPath node isolates the items in a list and all subsequent text. A GroupBy node concatenates the text from all items together and a String Manipulation node concatenates this result with the subsequent text. At the end of the metanode named “HTML clean up” we get the full content clean from all original HTML code.

Now we have all required ingredients to define a Document: content text, title, and author. A Strings to Document node collects all of them into a Document type data cell. For this operation we chose the OpenNLP English WordTokenizer, since the text of the web page is in English language. The final workflow is shown in figure 2.16 and is available under Chapter2/04_WebCrawling.

XPath

This node parses XML data cells via XPath (1.0) queries.

The configuration window then requires an XML-type column from the input data table to work on. Once the input XML column has been selected, a preview of the first cell appears in the lower part of the window.

The text in the XML-Cell Preview panel is interactive. If you click on any of the XML item, let's say "article" in our case, you will automatically get - in green right above the panel - the correct query syntax to extract it. If you click the button "Add XPath" the query will be added to the query list in panel XPath Summary and will run during node execution.

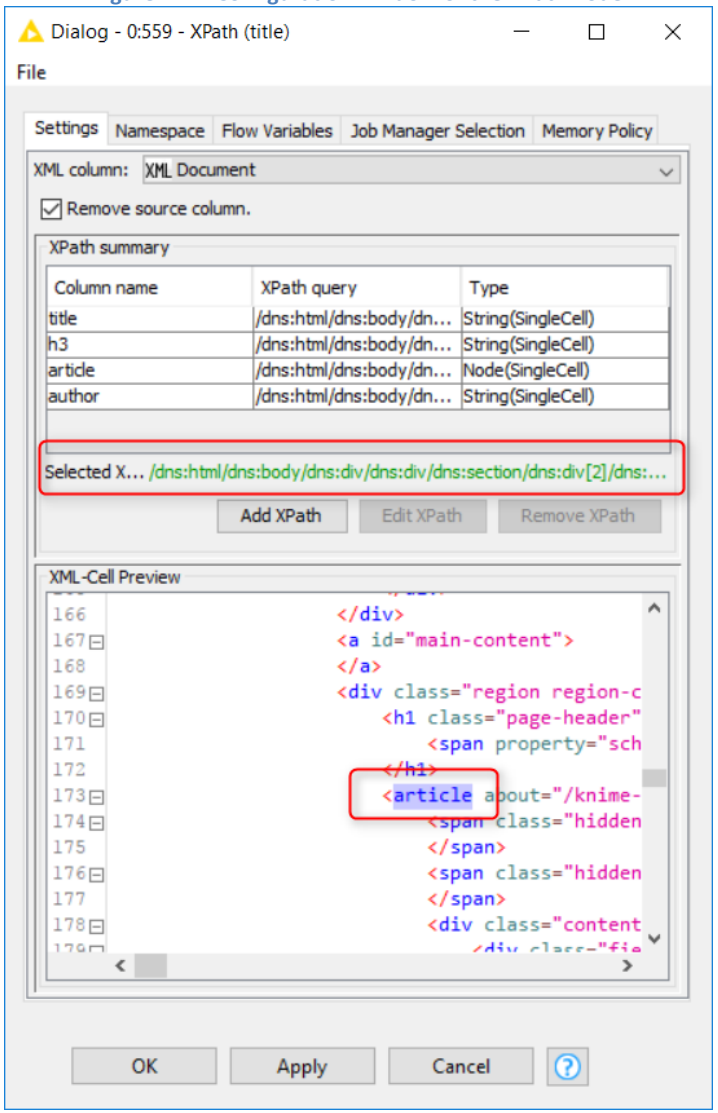
When double-clicking any of the queries in the list, you get a chance to customize it, particularly the name and type of the output column, the query itself, and the multiple tag options.

The column return type can be: Boolean, Integer, Double, String, and Node, where Node means an XML-type column.

Multiple tag options can assume the following values: Single Cell, Collection Cell, Multiple Rows, and Multiple Columns. Single Cell produces one result cell for each queried XML cell. However, sometimes you might get a list of items as a result of the query. In this case, you need to use one of the other multiple tag options. Multiple Rows (Columns) produces as many rows (columns) as many items, with each item on a single row (column). Collection Cell aggregates all resulting items into a single cell of type Collection. You will need an Ungroup node or a Split Collection Column node to retrieve the single items later on.

Settings in the Namespace tab allow to define prefixes and namespaces to be used in the XPath queries.

Figure 2.14. Configuration window of the XPath node



Content Extractor

Also this node belongs to the Palladian extension. This node helps with the extraction of text from HTML and, more generally, XML documents, by removing all HTML tags, footers and headers, ads, navigation etc.

There two possible extraction algorithm: “Readability” imported from Arc90 and the native “Palladian”. Readability performs best for general content extraction, while Palladian performs best when parsing comment sections.

The other configuration setting consists of the XML-type input data column.

This single node performs the work of many consecutive XPath nodes.

Figure 2.15. Configuration window of the Content Extractor node

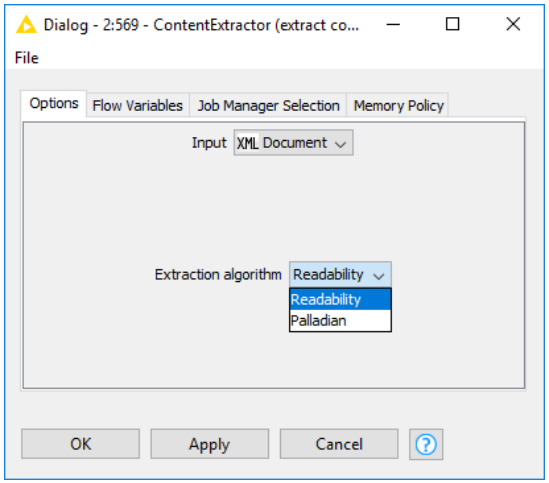
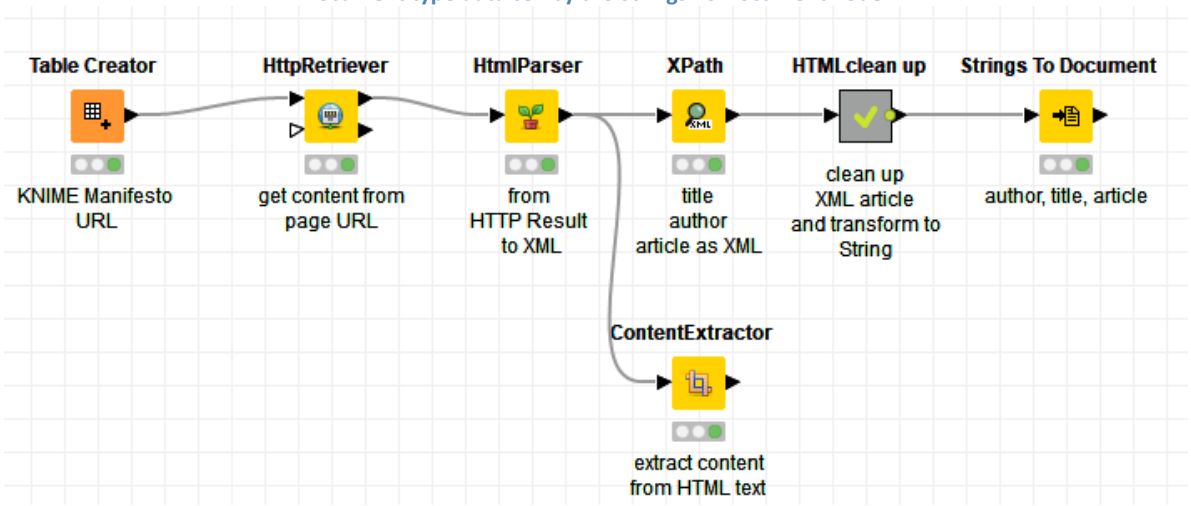


Figure 2.16. Web Crawler Workflow. The URL to crawl is specified in the Table Creator node, its content is downloaded by the HttpRetriever node and parsed by the HtmlParser node; finally a series of XPath nodes extract and clean from residual HTML code author, title, and page content. Author, Title, and content are finally collected together into a Document type data cell by the Strings To Document node.



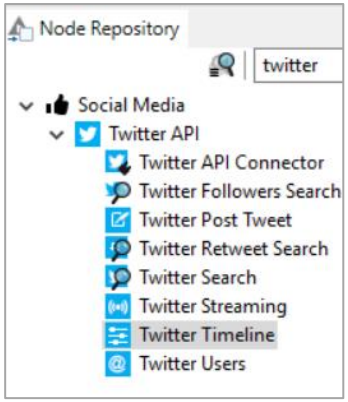
2.5. Social Media Channels

Social media channels could not miss from this very short list of possible sources of text data. Twitter, Facebook, and Co. are a huge repository of comments and feedbacks just waiting to be analyzed. KNIME Analytics Platform offers dedicated nodes to connect to some of these social media. For those social media channels, where no dedicated connection nodes are available, we can always resort to a connection via REST service. We will show both options in this section.

2.5.1. Twitter

As part of the nodes for social media channels offered by KNIME Analytics Platform, you can find a full category dedicated to Twitter and including a large variety of functionalities: connect; search tweets, retweets, or followers; post tweets and retweets; get the tweet streaming; get a user timeline; and more. All these nodes wrap functionalities from the Twitter API service.

Figure 2.17. Twitter API nodes in Node Repository



In order to access the Twitter API, you need an API Key and an Access Token. In order to get both, you will need to get a Twitter Developer account at https://dev.twitter.com/resources/signup and then to register your application under "My Apps" -> “Create New App” (https://apps.twitter.com/app/new). The API Key should then become immediately available and Access Tokens may be generated and revoked as needed. Both, API Key and access tokens, should be reachable from the “API Keys” tab on the page of your registered application. API Key and Access Tokens come respectively with an “API secret” and an “Access Token secret”. Remember all of them: API Key, Access Token, and their secrets.

In this section, we build a small workflow to show how to use the Twitter nodes. The goal of this workflow is to search and import tweets revolving around the *#knime* hashtag and to isolate and import the timeline of user *@DMR_Rosaria*, one of this book’s authors.

To do that, we will use only 3 nodes of the Twitter extension: Twitter API Connector, Twitter Search, and Twitter Timeline. We leave to you the joy of the exploration of the other Twitter API nodes.

Twitter API Connector

This node creates a connection to access the Twitter API.

For the connection you need to be logged in with a Twitter Developer account (see above how to get a Twitter Developer account); that is with the API Key and an Access Token associated to that Developer Account.

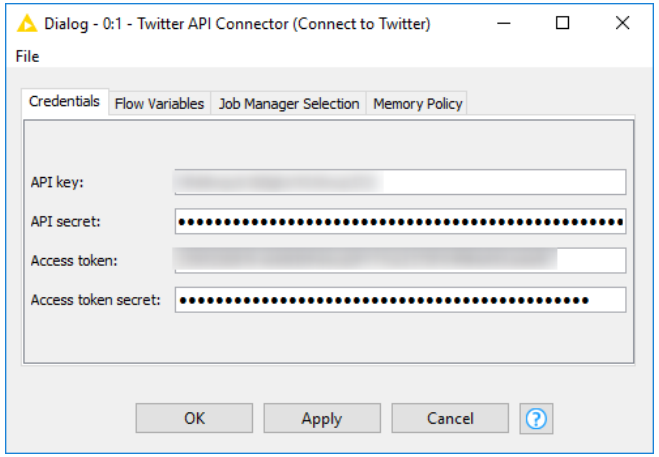
The configuration window of the node then requires:

- API Key
- Access Token
- API secret
- Access Token secret

All of which you have received when signing up for the developer account and when registering your application.

The node output port is a pale blue square, which indicates a Twitter connection.

Figure 2.18. Configuration window of the Twitter API Connector node



The first node of our workflow is a Twitter API Connector node, including the API Key, the Access Token, and their secrets that we received when we opened the Twitter Developer account (https://dev.twitter.com/resources/signup) and registered the application.

Obviously we have not made our API Keys, access tokens, and secrets available in the workflow. You will need to get your own account, register your own application, and get your own API Key, Access Token, API secret, and Access Token secret.

Note. The workflow that we are building here and that is available under Chapter2/05_Access_Twitter will not run on your machine. You will need to provide your own API Key, Access Token, API secret, and Access Token secret in the Twitter API Connector node configuration window.

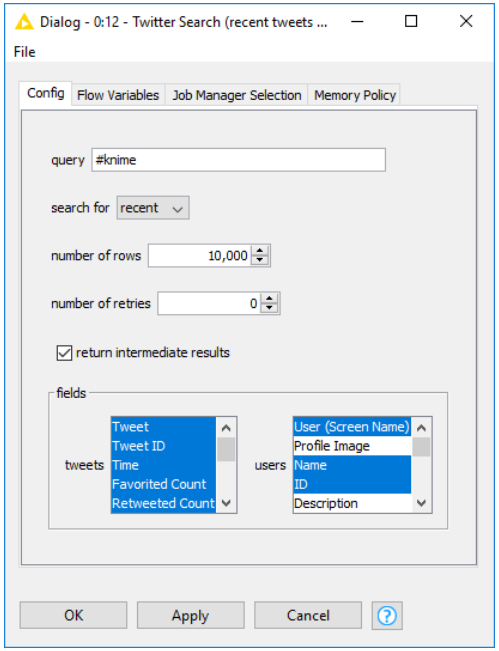
After execution, a connection with Twitter has been established and the workflow is now ready to query and run Twitter services.

Twitter Search

The node implements a Twitter search, the same that you can run from the Twitter page or from https://twitter.com/search-home . To run your search you need to define:

- A query based on words, hashtags, and/or operators for an advanced search (see https://support.twitter.com/articles/71577#)
- The scope of your search. For example the most popular tweets or the most recent tweets, or both
- The maximum number of tweets (rows) to retrieve
- The number of retries. Twitter offers a 15 minutes time window to send and execute the query. If the time window has already been used and no results have been obtained, you can try again for the next 15 minutes, and again, and again. You can define here how many times to try again.
- The option “return intermediate results”. If execution is cancelled for some reason, if this option is enabled, you will get whatever has been retrieved till cancellation.
- Finally the fields to be retrieved for the tweets as well as for the tweeters.

Figure 2.19. Configuration window of the Twitter Search node



Twitter API accepts a maximum number of requests in a 15 minute time window (see API Rate Limits and Rate Limits: Chart): 180 via user authentication or 450 via application authentication. One request can return a maximum of 100 tweets. Therefore, the maximum number of tweets that can be retrieved in a 15 minute window is 18000 or 45000, respectively. Also tweets are retrieved only within a one week past time scope. For more details check the Twitter page GET search/tweets.

Note. The Twitter's search service and, by extension, the Twitter Search API is not meant to be an exhaustive source of tweets. Not all tweets will be indexed or made available via the Twitter search interface.

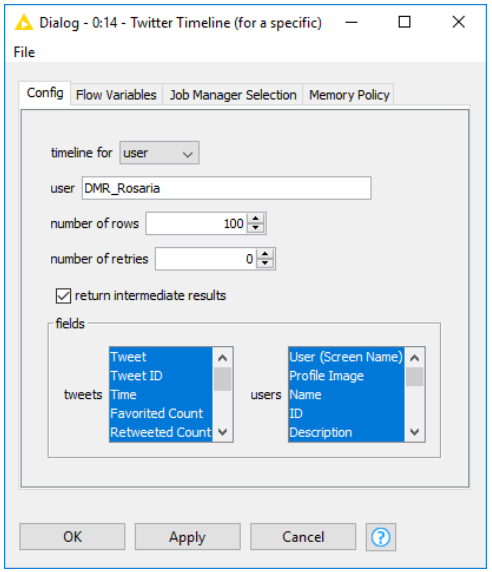
We set the search string to the hashtag *#knime*, to see what people have been saying on Twitter about KNIME in the past week.

Twitter Timeline

The Twitter Timeline node selects a user and retrieves the posted tweets by the user (user), the retweets of his/her tweets (retweets), the mentions containing @username (mentions), and the posted tweets by user and followers (home). The same limitations as above apply to this tweet retrieval operation. The service required settings are:

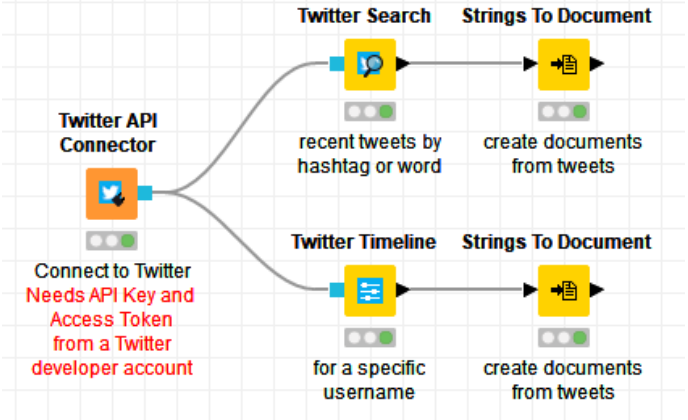
- The scope of the time line (user, home, mentions, retweets)
- The user name of the selected user
- Maximum number of rows to be retrieved
- The number of retries. Twitter offers a 15 minutes time window to send and execute the query. If the time window has already been used and no results have been obtained, you can try again for the next 15 minutes, and again, and again. You can define here how many times to try again.
- The option “return intermediate results”. If execution is cancelled for some reason, if this option is enabled, you will get whatever has been retrieved till cancellation.
- Finally the fields to retrieve about the tweets as well as about the tweeters.

Figure 2.20. Configuration window of the Twitter Timeline node



We set the user to *DMR_Rosaria*, the handle of one of this book’s authors. The goal is to retrieve her timeline, i.e. her tweets and retweets.

Figure 2.21. Workflow to extract last week’s tweets containing hashtag *#knime* and from *@DMR_Rosaria* timeline.



The final workflow is reported in figure 2.21. It starts with a Twitter API Connection to connect to the Twitter API; then it runs a Twitter search around the hashtag *#knime* on one branch and retrieves the tweets timeline of *@DMR_Rosaria* in the other workflow branch. All tweets retrieved from both searches are then transformed into Document cells, where Tweet body is used as title and as full text, user-name is used as author, "Twitter" is the fixed document source, and current fixed time is the publication date.

Other example workflows using the Twitter API can be found on the KNIME EXAMPLES server under *08_Other_Analytics_Types/04_Social_media*.

2.5.2. REST API Example: YouTube Metadata

Another massive source of web data is brought to us via REST API, from Google services, Microsoft cloud, Amazon Cloud, Weather Stations, CRM software, geo-location services, etc …

Google, in particular, offers a high number of REST API services to access, search, and modify data and profiles from its many social channels, cloud services, and mobile tools. A list of the available Google REST API Services can be found on the Google Cloud Platform Library. Google also offers a wide variety of services in the 3 authorization categories: no authorization, only via API key, and with the OAuth2 authorization protocol. Because of the high number of services offered and because of the standard authorization procedures required, we have decided to show you in this section how to retrieve data from a Google REST API service. The hope is that this example will be easy to extend to other REST API services, even if not run by Google.

One of the many interesting REST APIs offered by Google accesses YouTube metadata. YouTube metadata involves video metadata as well as comment and channel metadata. The YouTube REST API is part of the more general Google REST API. As for all Google APIs, there are two types of authorization. If all you want to run is a search on public data, an API key based authorization is sufficient. If you want to change the data or access private content, a more strict OAuth 2.0 authorization protocol is enforced.

In this section, we want to run a simple search on public YouTube videos around the tag "KNIME" and we want to extract the comments and the properties for each resulting video. To do that, we use 3 YouTube REST API services:

The "***search***" service searches for videos. In particular, we will search for videos tagged with the word "KNIME" (*q=KNIME*), for a maximum of 50 videos in the response list (*maxResults=50*), and of any duration
https://www.googleapis.com/youtube/v3/search?q=KNIME&part=id&maxResults=50&videoDuration=any&key=<your_API_key>.

The "***videos***" service extracts the details, such as duration, permission flags, total number of views, likes, etc.., for the selected video (*videoId=<videoID>*)
https://www.googleapis.com/youtube/v3/videos?id=<videoID>&part=snippet,statistics,contentDetails&key=<your_API_key>.

The "***commentThreads***" service retrieves the comments posted on the selected video (*videoId=<videoID>*), including the comment text, author ID, and posting date https://www.googleapis.com/youtube/v3/commentThreads?videoId=<videoID>&part=snippet,id&key=<your_API_key>.

These three services require only to specify an API key for the authorization procedure. You can request your own key API directly on the Google API Console. Remember to enable the key for the YouTube API services. The available services and the procedure to get a key API are described in these 2 introductory pages:

- https://developers.google.com/apis-explorer/?hl=en_US#p/youtube/v3/
- https://developers.google.com/youtube/v3/getting-started#before-you-start

These three services can also be accessed via a simple GET Request. In KNIME Analytics Platform, a GET Request can be sent to the selected REST service through a **GET Request** node. The idea here is to extract a list of YouTube videos tagged with the word "KNIME" via the service "search"; then, for each resulting video, to extract its properties via the service "videos" and its comments via the service "commentThreads". We need to prepare three GET Requests through three GET Request nodes. The key-piece of this puzzle is the GET Request node.

GET Request

This node issues HTTP GET requests, to retrieve data from a REST web service. To do that you need to define settings in the three configuration tabs.

Connection Settings. Here, the REST service URL, including possible parameters, is set. This can be a hard-coded URL in the "URL" box or a dynamic URL from an input data column in the menu named "URL column". The node will send a GET Request to each URL in the selected input column and will collect the results in the output data table.

"Delay (ms)" sets the time between consecutive requests and "Concurrency" sets the number of concurrent requests to send.

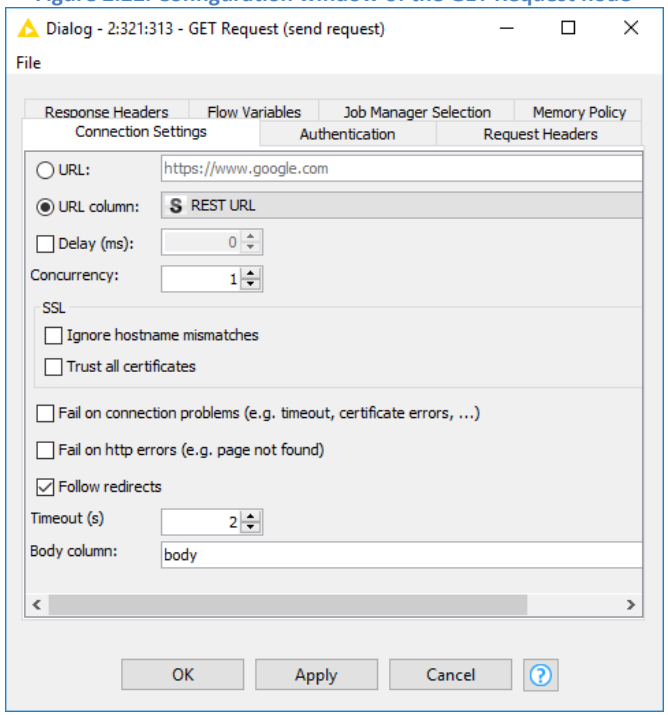
"SSL" frame sets tolerance and flags and error handling flags

The last option sets the name of the response column in the output data table

Request/Response Headers. Here you can specify request headers, if needed, and response headers, if wished. By default, the content type and the HTTP status code are added to the response object and therefore to the output data table.

Authentication. The node supports several authentication methods, e.g. NONE, BASIC, and DIGEST.

Figure 2.22. Configuration window of the GET Request node



For each one of the three GET Requests, we build the appropriate String URL - with all the right parameters - through a String Manipulation node; then we send the request to the REST service via a GET Request node; the JSON formatted result array is extracted using a JSON Path node and then ungrouped into single rows; and finally the remaining JSON structured rows are transformed into KNIME data rows with a JSON to Table node. This metanode structure (Fig. 2.23) is repeated three times in the workflow to execute the three different GET Requests. Eventually video comments and video descriptions are transformed from String into Document objects with a Strings To Document node. The final workflow is shown in figure 2.24.

Note. Authentication here is not provided via the Auhtentication tab in the node configuration window, but by appending the API key at the end of the REST service URL, as required by the YouTube API.

Figure 2.23. Metanode template to send a GET Request to a YouTube REST service (such as "search", "videos", "commentThreads") and extract the results.

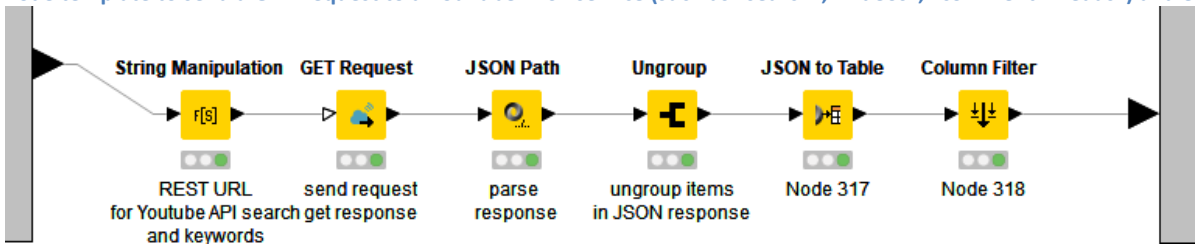
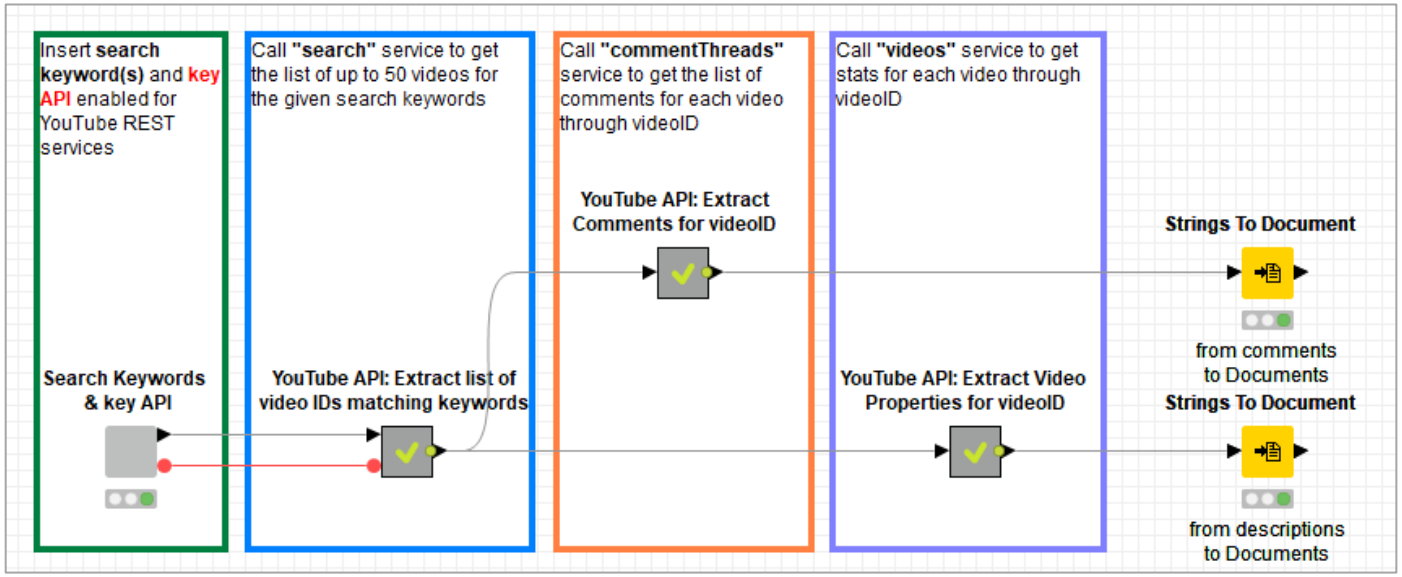


Figure 2.24. Final Workflow to send a GET Request to the three YouTube REST services "search", "videos", "commentThreads". The three metanodes all contain a String Manipulation, a GET Request, and a few JSON nodes, as shown in Fig. 2.23.



Other example workflows using the REST API nodes can be found on the KNIME EXAMPLES server under *01_Data_Access/05_REST_Web_Services*.

2.6. Text Input Form in a Web Page

Sometimes, applications require a web text input. This can be achieved with a String Input node. The String Input node is a **Quickform** node; that is one of those special nodes that outputs a flow variable.

Quickform nodes are special nodes, because, besides outputting a flow variable, they provide a User Interface frame to input the flow variable value. The User Interface frame can vary from a simple textbox to radio buttons, checkboxes, and menus. Each Quickform node produces one or more of these UI options.

If a Quickform node is wrapped into a wrapped metanode, the metanode acquires a dialog consisting of the UI input forms of the contained Quickform node(s). In addition, if the same wrapped metanode is executed from the WebPortal of a KNIME Server, its acquired dialog becomes a full web page with the same input forms.

There are many Quickform nodes, each one producing a different flow variable type, implementing a different task, and using a different UI form. Besides input Quickforms, we find Quickform nodes for selection, filtering, file download, output text, and more in category Workflow Control / Quickforms in the Node Repository panel.

A Text Mining application often collects its input data from a text input form on a web page. This text input form on a web page can be implemented, for example, using a simple String Input Quickform node.

We used a String Input node to just input a value for a String flow variable from a TextBox.

We wrapped this same String Input node into a wrapped metanode, which we named “Input Text”. The wrapped metanode then acquired a dialog window with the same TextBox. When opening the metanode’s configuration window, we find a multi-line TextBox and a flag to change its default value. By enabling the flag “Change” we can input a new text to feed our text mining workflow. This same UI TextBox input form appears on a web browser, if the workflow is running on the WebPortal of a KNIME Server.

String Input

The String Input Quickform node is a very simple node. It produces a String type flow variable. Its default value is set in the node configuration window. The configuration window contains settings for the UI form and for the generated flow variable..

UI Form

- "Label" is an explanatory text displayed close to the input form in the metanode dialog and in the web UI.
- Similarly "Description" is a descriptive text of the value required in the UI. This descriptive text shows as a tooltip when hovering on the input form in the metanode dialog/web page.
- "Editor Type" defines the line span of the input text, whether a single line or a multi-line paragraph.
- "Multi-line editor width/height" defines the layout setting of input textbox
- "Regular Expression" and "Common Regular Expressions" can be used only in combination with the single-line input mode and define rule(s) for valid entries.
- If an invalid entry is typed, the text in "Validation Error Message" is displayed.

Output Flow Variable

- "Variable Name" defines the name of the output variable generated by this node.
- Similarly, "Default Value" defines the default value for the output flow variable
- "Parameter Name" is the name of this flow variable when assigned on the run, for example during batch execution.

Figure 2.25. Configuration window of the String Input node

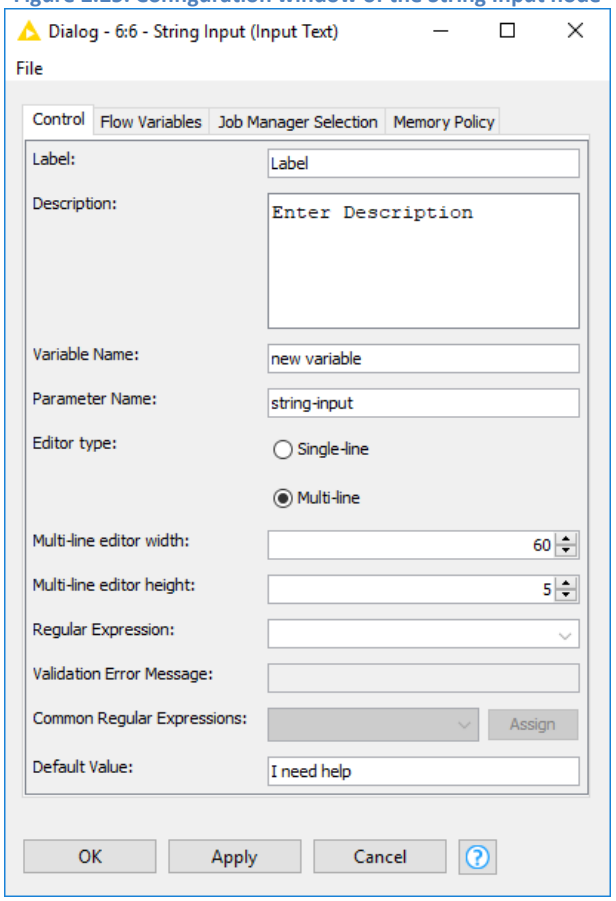


Figure 2.26. The String Input Quickform node and a wrapped metanode containing a String Input Quickform node.

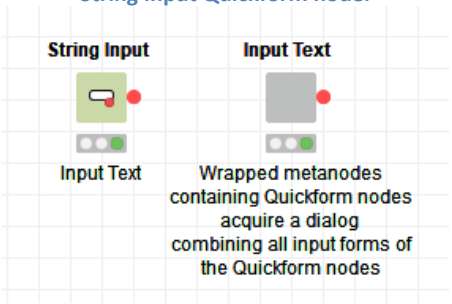
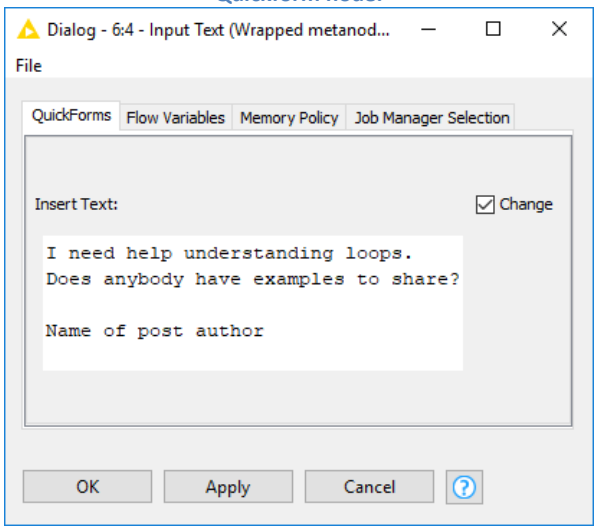


Figure 2.27. The dialog acquired by the wrapped metanode containing a String Input Quickform node.



2.7. Exercises

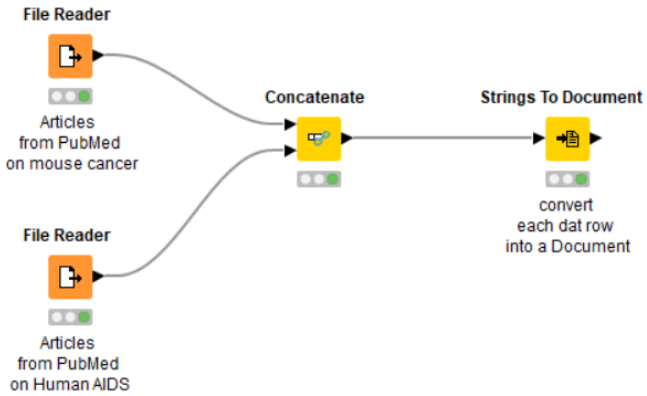
Exercise 1

Read datasets Thedata/mouse-cancer.csv and Thedata/Human-AIDS.csv. Both datasets contain scientific articles from PubMed with abstract, text, author(s), title, etc ... Each data row is a scientific article.

Concatenate the two datasets together and transform all articles into Documents.

Solution

Figure 2.28 Workflow to read, concatenate and convert science articles from PubMed into Documents



Exercise 2

Read the content of epub book pg1513.epub in folder Thedata, which contains the tragedy "Romeo and Juliet".

Who is the author? The keywords? And the publishing rights?

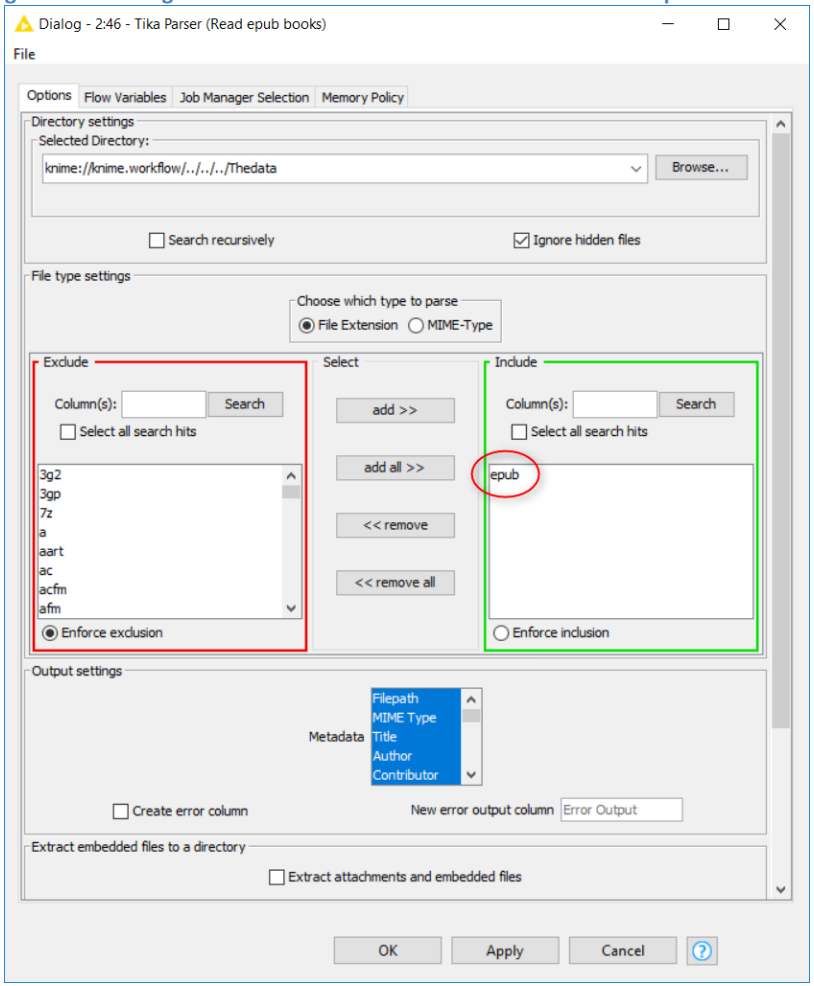
Solution

To read the epub book file, you can use the Tika Parser node or the Tika Parser URL Input node. Configuration window of the Tika Parser node set to parse epub documents is shown in the figure below.

In the very unlikely case that you do not know the answers:

- Author is “William Shakespeare”
- Keywords are “Vendetta – Drama”
- Publication Rights are “Public Domain in the USA”.

Figure 2.29. Configuration window of the Tika Parser node to read epub documents.



Exercise 3

Connect to one of these three RSS Feeds:

- New York Times World (English) URL: http://rss.nytimes.com/services/xml/rss/nyt/World.xml
- BBC (Chinese) URL: http://feeds.bbci.co.uk/zhongwen/simp/rss.xml
- Frankfurter Allgemeine Zeitung (FAZ) (German) URL: http://www.faz.net/rss/aktuell/

and download the news!

Solution

The workflow is similar to the one depicted in figure 2.9.

However, remember to set the Word Tokenizer to one compatible with the RSS Feed language! English, Chinese, and German tokenizers are available in the Word Tokenizer menu in the configuration window of the RSS Feed Reader node.

Chapter 3. Text Processing

3.1. What is Text Processing?

In the previous chapters, we have seen how to access and retrieve text data in a variety of formats, such as simple text files, RSS feeds, social media channels, and web crawling. What comes next? Eventually we will need to convert those texts into numbers to train machine learning algorithms. However, before we reach that point, it might be useful to process the text, in order to make its hidden information easier to retrieve. This probably means to remove all not strictly meaningful words, remove grammar forms through stemming, and apply tags to identify particular words. Despite all our cleanup efforts, the number of words left in the text might still be too high to train a machine learning in a reasonable time. We might not need all those words after all. A more compact text representation could be achieved via keyword extraction. Keywords still nicely summarize the Document content and are usually noticeably less than all original text words. This is the Text Processing phase.

Classic steps in the text processing phase are:

- enrichment
- filtering
- stemming
- word or keyword extraction
- transformation of the remaining text into numbers

Not all steps are always necessary; this really depends on the goal of the analysis. In the following sections we will show the nodes or the sequence of nodes to implement each one of those steps in a KNIME workflow.

Note. The KNIME Text Processing extension contains mostly nodes for this part of the analysis. In order to train Machine Learning models we will refer to the nodes in the Mining category.

3.2. Enrichment: Taggers

Now that the Document has been parsed, we can focus on enriching the final tokens, i.e. tagging them as named entities or by grammar role. Nodes performing this action are the so called Taggers.

Tagger nodes scan Documents, recognize the role of each word, and tag it accordingly. Depending on the definition of the "role of each word", different tag types are available: part of speech, named entities like cities or person names, sentiment, and other more domain specific types. Inside a tag type a number of tag values are possible. For example if a word has to be tagged for sentiment, this could be "very positive", "very negative", "neutral", or any of the nuances in between.

Most tagger nodes are specialized for some type of tagging and all tagger nodes can be found under the Enrichment category in the Node Repository.

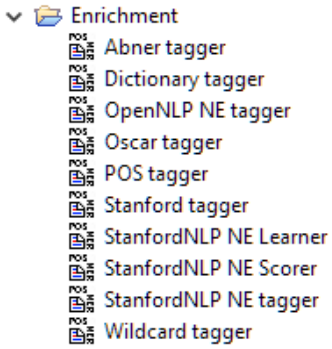


Figure 3.1. Enrichment nodes (Taggers) available in the Node Repository

All enrichment nodes available in KNIME Analytics Platform work on an input column of Document type. The output port presents the modified Document type column as a new column, thus preserving the original one, or as a replacement of the original Document type column. Some terms in the new Document cells have been recognized and now carry a tag set by the tagger node during execution. Such tags can be recovered later and can be used for filtering only specific terms.

KNIME Analytics Platform offers dedicated nodes to perform the enrichment of a given document, such as POS Tagger, Stanford Tagger, Stanford NLP NE Tagger, Abner Tagger, Wildcard Tagger, and more. In this chapter we present an overview of these nodes and of their implementation in a KNIME workflow. The workflows used here are stored in Chapter3/01_POS_NE_Taggers and Chapter3/02_Dictionary_Taggers.

Note. As we have seen in the previous chapter, the first step is always to convert a text into a Document. Nodes implementing this conversion, like the *Strings To Document* node, perform a tokenization of the original text. However, all enrichment nodes require a second tokenization. Indeed tagging is performed on full sentences, treated as single Strings. Thus, after tagging, these sentences need to be tokenized again.

Two special tagger nodes are the Wildcard Tagger and the Dictionary Tagger. They are special in the sense that they do not refer to a pre-defined tag type, but they can scan the text for words from a dictionary and attach to them any type of tag, be it sentiment or part of speech or named entity or some other tag type.

Node. In case of conflicting tags, the last enrichment node in the workflow decides the final tag. It also decides the final tokenization. The word combination produced by previous nodes does not exist anymore.

Before we proceed with the description of the many tagger nodes, we would like to draw the attention on a common feature.

Since all tagger nodes require a tokenization algorithm, a Document type input column, and a Document type output column, all these settings have been collected in a single tab in the configuration window of any tagger nodes. You will find this same tab (Fig. 3.2) – named “General options” - in the configuration window of all tagger nodes. The POS Tagger node has the simplest configuration window, consisting of only this tab.

3.2.1. Part-Of-Speech Taggers

Parts-of-speech (also known as POS) tagging is the process of assigning a part-of-speech marker to each word in a given document. In this process eight parts-of-speech - considering the English vocabulary - are included: nouns, verbs, pronouns, prepositions, adverbs, conjunctions, participles, and articles. Hence, in POS tagging the goal is to build a model whose input is a sentence and whose output is a tag sequence. Part-of-speech tags are useful for information retrieval and word sense disambiguation.

The POS tagging is commonly defined as a disambiguation task. Words are ambiguous, which means that they can have more than one possible part-of-speech. For example, “delay” can be a verb (*the flight will be delayed at the gate for two or more hours*) or a noun (*possible delay in delivery is due to external factors*). Thus, the problem of POS-tagging is to resolve these ambiguities, choosing the proper tag for the context.

There are two main nodes to perform Part Of Speech tagging: POS Tagger and Stanford Tagger.

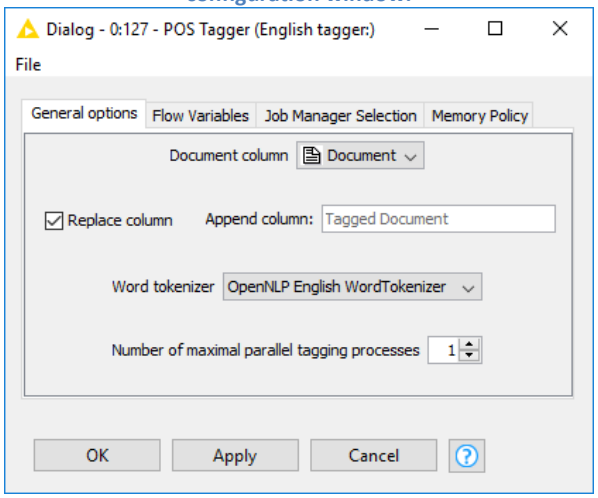
POS Tagger

This node assigns to each term of a Document a part of speech (POS) tag for the English language by using the Penn Treebank tag set.

„General options“ Tab

- The name of the Document type input column to work on
- Whether the resulting Document type cells must overwrite the input cells; if not, then the name of the new Document type output column
- The tokenization algorithm
- The maximum number of parallel tagging process; this is just a performance setting and does not influence the outcome of the tagging process.

Figure 3.2. “General Options” tab in the configuration window of the POS tagger node. All tagger nodes show this same tab in their configuration window.



The *POS Tagger* node assigns English POS tags. The assigned tag type is "POS" and the POS values are the ones of the Penn Treebank tags. There are many lists of part-of-speech tags out there, but most modern language processing on English uses the tags defined by the Penn Treebank Project. An extract of the full alphabetical list of part-of-speech tags used in the Penn-Treebank project can be seen in figure 3.3.

Figure 3.3. POS Tags and their meanings from the full list of Part-Of Speech used in the Penn-Treebank project

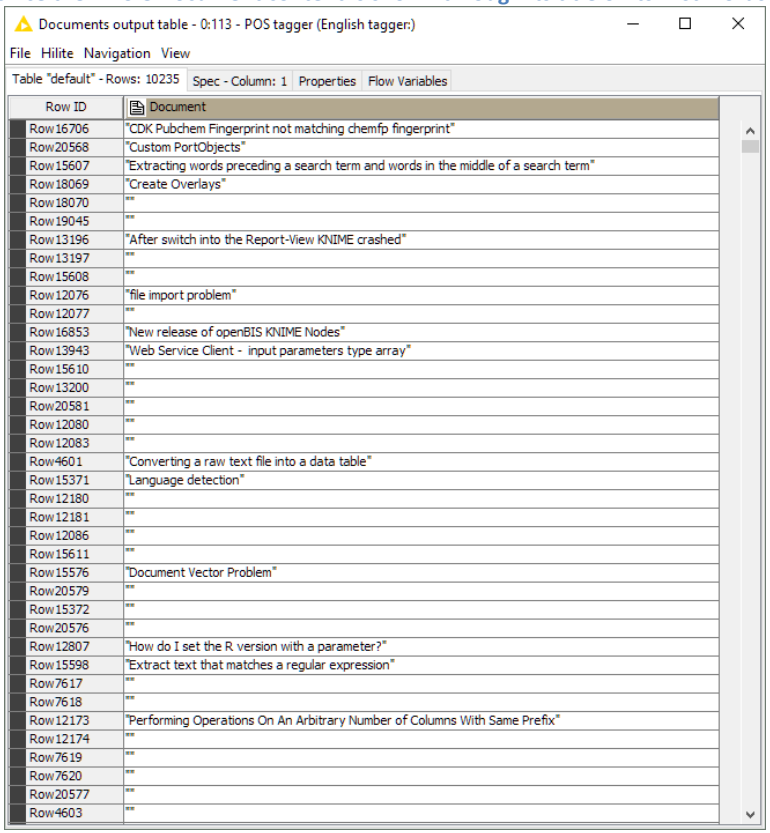
Tag	Description
CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential there
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun

Tag	Description
PRP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol
TO	to
UH	Interjection
VB	Verb, base form
VBD	Verb, past tense
VBG	Verb, gerund or present participle
VBN	Verb, past participle
VBP	Verb, non3rd person singular present
VBZ	Verb, 3rd person singular present
WDT	Whdeterminer
WP	Whpronoun
WP$	Possessive whpronoun
WRB	Whadverb

As an example, we have POS tagged the Documents in the KNIME Forum data set in the workflow Chapter 3/01_POS_NE_Taggers. This data set contains all posts that have been published between May 31st 2013 and May 31st 2017 on the KNIME Forum. That is: the forum source, the post title, the post URL, the post content, the number of replies, the author's username, the publishing date, and a few additional less interesting columns. After removing a few data rows, where no post content has been retrieved, applying the Strings To Document node, and filtering all the columns except for the Document one, we apply the POS tagger node. The output is a table containing the tagged Documents, according to the Penn Treebank Tagset (Fig. 3.4).

Note. When visualizing the output data table of a Text Processing node, Document type cells are not easy to inspect, since the whole content of the Document is hidden behind the title or the first words. Therefore it is not easy to see the result of a tagging process just by visually inspecting the output table. For a deeper inspection you will need to use a Document Viewer node or extract the words one by one using a Bag of Words node.

Figure 3.4. Output Documents of the *POS Tagger* node. Document content and tags cannot be seen from the output table, since the whole Document content is shown through its title or its first words.



Row ID	Document
Row16706	"CDK Pubchem Fingerprint not matching chemfp fingerprint"
Row20568	"Custom PortObjects"
Row15607	"Extracting words preceding a search term and words in the middle of a search term"
Row18069	"Create Overlays"
Row18070	""
Row19045	""
Row13196	"After switch into the Report-View KNIME crashed"
Row13197	""
Row15608	""
Row12076	"file import problem"
Row12077	""
Row16853	"New release of openBIS KNIME Nodes"
Row13943	"Web Service Client - input parameters type array"
Row15610	""
Row13200	""
Row20581	""
Row12080	""
Row12083	""
Row4601	"Converting a raw text file into a data table"
Row15371	"Language detection"
Row12180	""
Row12181	""
Row12086	""
Row15611	""
Row15576	"Document Vector Problem"
Row20579	""
Row15372	""
Row20576	""
Row12807	"How do I set the R version with a parameter?"
Row15598	"Extract text that matches a regular expression"
Row7617	""
Row7618	""
Row12173	"Performing Operations On An Arbitrary Number of Columns With Same Prefix"
Row12174	""
Row7619	""
Row7620	""
Row20577	""
Row4603	""

The *Stanford Tagger*node assigns POS tags following the models developed by the Stanford NLP group (http://nlp.stanford.edu/software/tagger.shtml). Unlike the POS Tagger node, the *Stanford* Tagger node works on many different languages, i.e. English, German, French, and Spanish.

POS tagging is normally implemented through sequence models, which can be Hidden Markov Models (HMM) [6] or Conditional Markov Models (CMM) [7]. These models approach the text from left to right, which means that the tag from the previous token can be used for the current token.

With the Stanford tagger node comes the English bidirectional POS tagging [8]. In the English bidirectional model, the graph has cycles. Shortly, this model inferences the tag for the current token based on the information available in tokens positioned in different parts of the document, not considering exclusively the left-to-right factors. Usually, this model performs better than others standard models at the price of higher computational expenses. Below is a summary of the models available in the Stanford Tagger node.

Tagging Option	Type of Model Used	Additional Info about the Model Used
English bidirectional	Penn Treebank tag set: https://nlp.stanford.edu/software/tagger.shtml	Accurate model. Trained on WSJ Corpus - sections 0-18 - using a bidirectional architecture and including word shape and distributional similarity features.
English left 3 words	Penn Treebank tag set: https://nlp.stanford.edu/software/tagger.shtml	Trained on WSJ Corpus - sections 0-18 - using the left 3 words architecture and includes word shape features.
English left 3 words caseless	Penn Treebank tag set: https://nlp.stanford.edu/software/tagger.shtml	Trained on WSJ Corpus - sections 0-18 – left 3 words architecture and includes word shape and distributional similarity features. This model ignores cases.
French		Trained on the French treebank.
German dewac	STSS tag set: http://www.ims.uni-stuttgart.de/forschung/ressourcen/lexika/TagSets/stts-table.html	
German fast	STSS tag set: http://www.ims.uni-stuttgart.de/forschung/ressourcen/lexika/TagSets/stts-table.html	
German hgc	STSS tag set: http://www.ims.uni-stuttgart.de/forschung/ressourcen/lexika/TagSets/stts-table.html	Trained on the first 80% of the Negra Corpus, which uses the STTS tagset. The Stuttgart-Tübingen Tagset (STTS) is a set of 54 German POS tags. This model uses features from the distributional similarity clusters built over the HGC (Huge German Corpus).
Spanish	PTB tag set (but with extra rules for Spanish contractions and assimilations): https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/international/spanish/process/SpanishTokenizer.html	
Spanish Distdim	PTB tag set (but with extra rules for Spanish contractions and assimilations): https://nlp.stanford.edu/nlp/javadoc/javanlp/edu/stanford/nlp/international/spanish/process/SpanishTokenizer.html	

Note. Tag sets across different languages are not compatible. The POS labels used in each tag set differ from the POS labels used in other tag sets.

Stanford Tagger

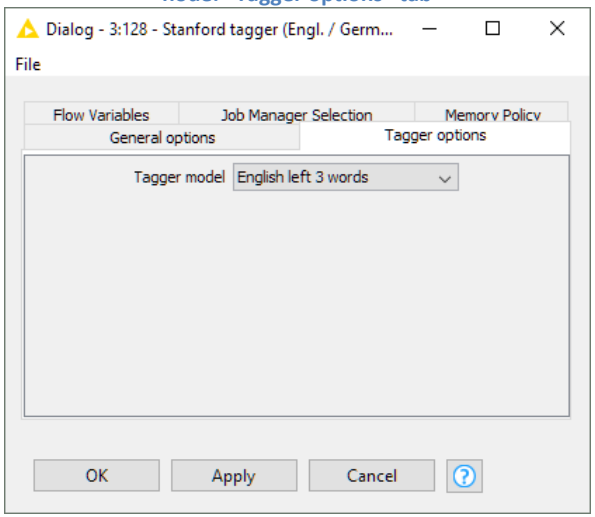
This node assigns a part of speech (POS) tag to each term in a Document using the Stanford Tagger tag set. POS models are available for English, French, German, and Spanish.

The node configuration window has two setting tabs: “General options” and “Tagger options”. The “General options” tab is the tab available for every tagger node (see Fig. 3.2) and requires input, output, and tokenizer settings.

„Tagger options“ Tab
The “Tagger options” tab contains the settings about the POS model. Available POS models are: English bidirectional, English left 3 words, English left 3 words caseless, French, German dewac, German fast, German hgc, Spanish, and Spanish distsim.

You will need to choose the model based on language, desired computational effort, and best accuracy.

Figure 3.5. Configuration window of the Stanford tagger node: “Tagger options” tab

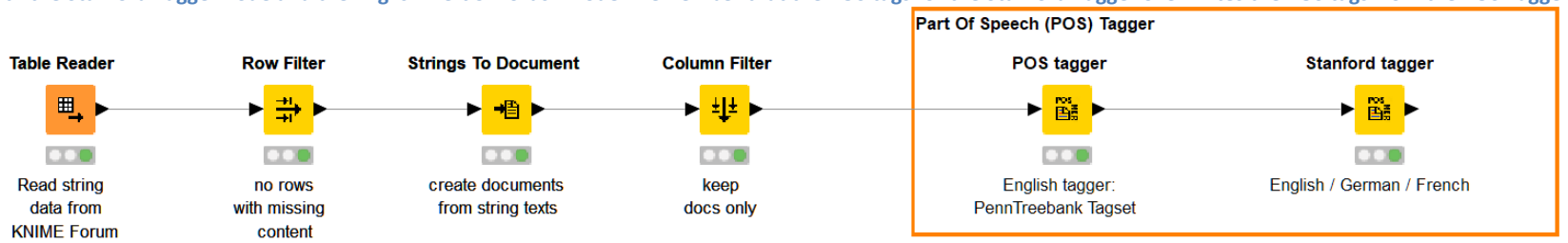


Note. **T**he tagger models available in the Stanford Tagger node vary in memory consumption and processing speed. Especially the models English bidirectional, German hgc, and Germany dewac require a large amount of memory. For the usage of these models it is recommended to run KNIME Analytics Platform with at least 2GB of heap space memory.

To increase the head space memory change the **-Xmx** setting in the knime.ini file. If KNIME Analytics Platform is running with less than 1.5GB heap space memory it is recommended to use POS models “English left 3 words”, “English left 3 words caseless”, or “German fast” respectively to tag English and German texts.

Here we extended the workflow Chapter 3/01_POS_NE_Taggers with a Stanford Tagger node, using the “English left 3 words” POS model, right after the POS Tagger node. Remember that in this case the Stanford Tagger node overwrites the POS tags of the previous POS Tagger node, since it comes after in the pipeline.

Figure 3.6. Upper branch of workflow Chapter 3/01_POS_NE_Taggers running two POS tagging operations on posts from the KNIME Forum. Once with the POS Tagger node and once with the Stanford Tagger node and the English "left 3 words" model. Remember that the POS tags for the Stanford Tagger overwrites the POS tags from the POS Tagger node.



3.2.2. Domain Taggers

Part of speech tags are not the only tags we can assign to tokens. Other tags involve entities, such as cities, names, times, money, or even molecules, genes, or other entities in other domains. KNIME Analytics Platform offers a few tagger nodes dedicated to recognize and tag entities.

Let's start with common named entities, such as cities, names, or money. The tagger node for generic named entities is the OpenNLP NE Tagger node. The OpenNLP NE Tagger node uses the models that have been developed within the Apache OpenNLP project, to recognize named entities and assign them the corresponding tag. The named entity category consists of three main sub-categories: entity names, temporal expressions, and number expressions.

- **Entities** (*organizations, persons, locations*). As *organizations* it is meant everything related to named corporate, governmental, or other organizational entities. As *persons* it is intended all named persons or families. Finally, as *locations* it is intended names of politically or geographically defined locations, such as cities, provinces, countries, international regions, bodies of water, mountains, etc…

- **Times** (*dates, times*). The tagged tokens for dates refer to complete or partial date expressions, instead tagged tokens for times refer to complete or partial expressions of time of day.

- **Quantities** (*monetary values, percentages*). In this case the numbers may be represented in either numerical or alphabetic form.

Note. It is important to keep in mind that the named entity models are language dependent and only perform well if the model language matches the language of the input text. The Open NLP NE Tagger node works only for English.

Actually, the OpenNLP NE Tagger node gives the opportunity to use even external OpenNLP models to recognize entities that can be used besides the ones provided.

OpenNLP NE Tagger

This node recognizes standard named entities in English based on the Apache OpenNLP models, version 1.5.2 and assigns them the corresponding tag.

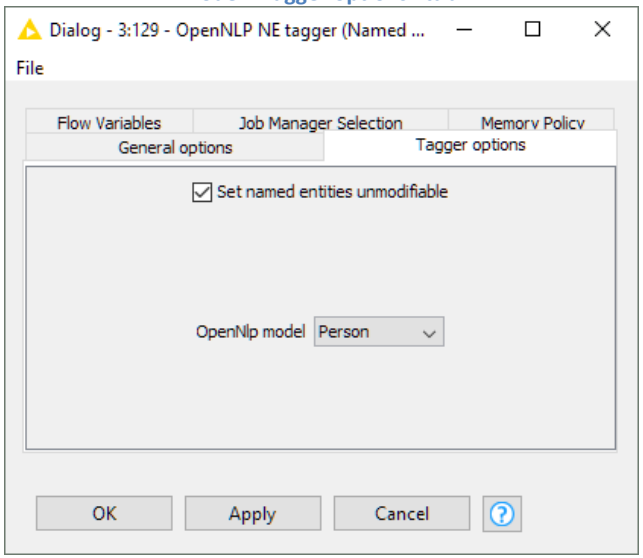
The node configuration window has two setting tabs: “General options” and “Tagger options”. The “General options” tab is the tab available for every tagger node (see Fig. 3.2) and requires input, output, and tokenizer settings.

„Tagger options“ Tab

- *Set named entities unmodifiable*. This option sets the named entity tags as unmodifiable. That is, whenever any subsequent tagger node in the workflow will not affect the previously tagged named entities.

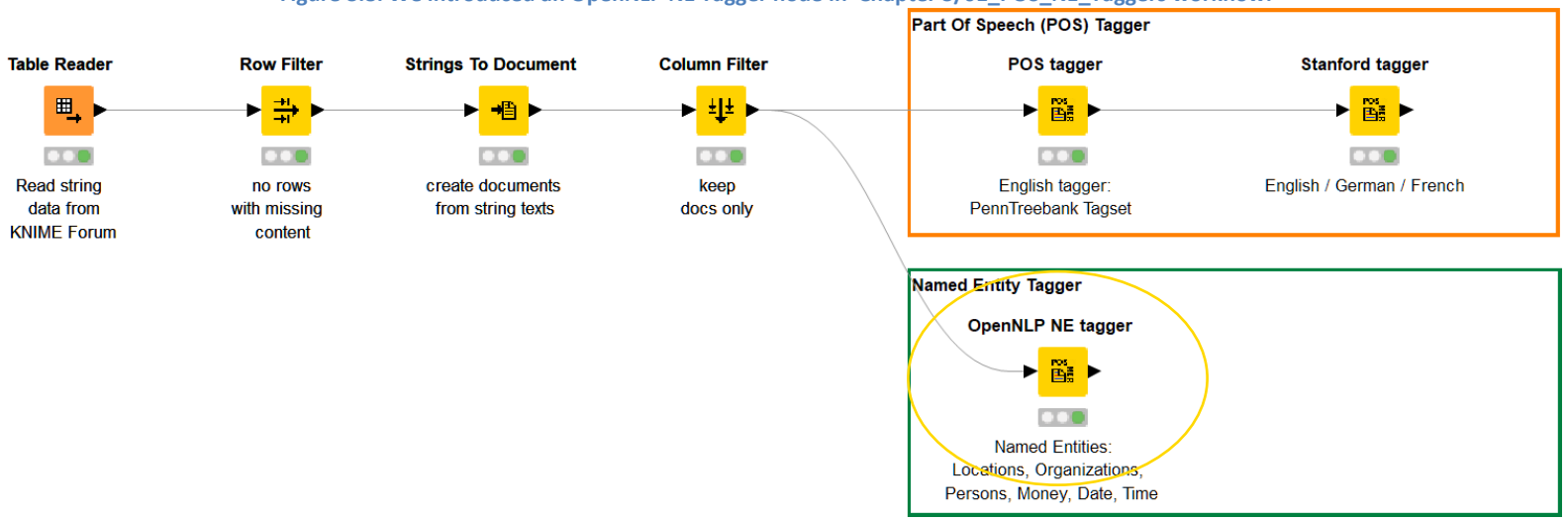
- The dropdown menu *OpenNlp model* allows to select the model to use on the input Documents. The selected model is specialized in recognizing a given type of named entities, among the ones described earlier.

Figure 3.7. Configuration window of the OpenNLP NE tagger node: “Tagger options” tab



Continuing with the workflow Chapter 3/01_POS_NE_Taggers, we add an OpenNLP NE Tagger node, using the tag model for “Person”, that is for person names. We introduce this node on a parallel branch, even though POS tagging and named entity tagging should not conflict and even though we did set the named entities as unmodifiable in the node configuration window.

Figure 3.8. We introduced an OpenNLP NE Tagger node in Chapter 3/01_POS_NE_Taggers workflow.



There are also two tagger nodes handling biological named entities (Abner Tagger node) and chemistry named entities (OSCAR Tagger node).

The Abner Tagger node incorporates a library from the ABNER software, a biological named entity recognizer. This node recognizes occurrences of PROTEINS, DNA, RNA, and CELL-TYPE entities.

The OSCAR Tagger node integrates the Open Source Chemistry Analysis Routines (OSCAR) software. The OSCAR Tagger node identifies chemical named entities in a text and tags the terms accordingly.

Even though, these nodes operate on a particular domain, their configuration settings are very similar to the configuration settings in the previous tagger nodes, including the “General options” tab and the “Set named entities unmodifiable” flag.

Abner Tagger

This node is based on the ABNER software and recognizes biological named entities.

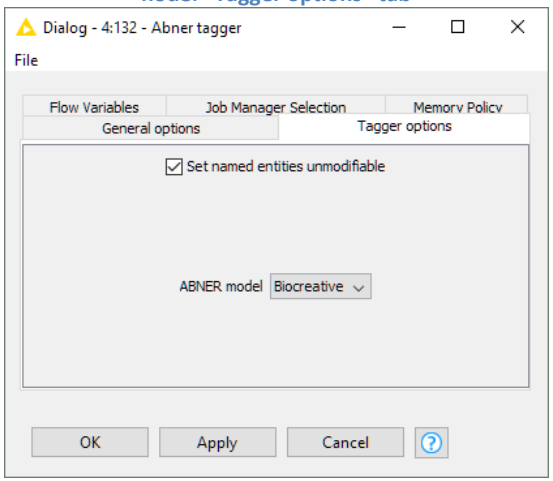
The node configuration window has two setting tabs: “General options” and “Tagger options”. The “General options” tab is the tab available for every tagger node (see Fig. 3.2) and requires input, output, and tokenizer settings.

„Tagger options“ tab

- *Set named entities unmodifiable*. This option sets the named entity tags as unmodifiable. That is, whenever any subsequent tagger node in the workflow will not affect the previously tagged named entities.

- *ABNER model*. This drop-down menu selects the model to use to recognize and tag biomedical named entities. To be more precise the node includes two models trained on the BIOCreative and the NLPBA corpora.

Figure 3.9. Configuration window of the Abner Tagger node: “Tagger options” tab



OSCAR Tagger

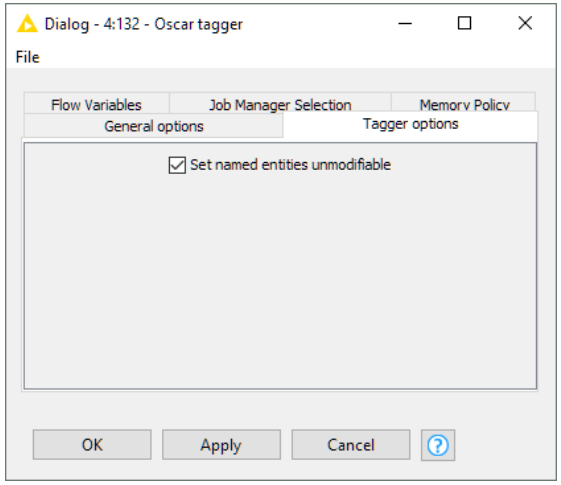
This node integrates the OSCAR chemical named entity recognizer framework (version 4.1.2). The node recognizes chemical named entities and tags them according to the default named entity types of the OSCAR framework.

The node configuration window has two setting tabs: “General options” and “Tagger options”. The “General options” tab is the tab available for every tagger node (see Fig. 3.2) and requires input, output, and tokenizer settings.

„Tagger options“ tab

- *Set named entities unmodifiable*. This option sets the named entity tags as unmodifiable. That is, whenever any subsequent tagger node in the workflow will not affect the previously tagged named entities.

Figure 3.10. Configuration window of the OSCAR tagger node



3.2.3. Custom Taggers

The tagger nodes offered by KNIME Analytics Platform are numerous, but surely do not address the variety of tagging needs. What about some custom tagging? For custom tagging, we rely on the *Dictionary Tagger* node and the *Wildcard Tagger* node.

Both nodes tag terms according to a list of words. The list of words - the dictionary - is fed into the node as a String column through a second input port. During execution, the node matches the Terms in the Documents with the words in the dictionary and tags the terms accordingly. A number of tag types are available, SENTIMENT, POS, STTS, NE, and more, as described in the previous sections.

Dictionary Tagger

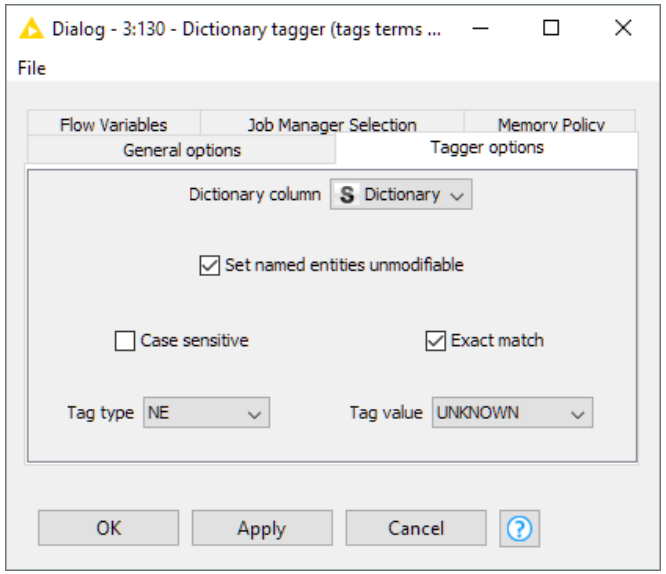
This node takes a dictionary column as second input, matches terms in the Document with words in the dictionary, and assigns a specified tag to the matching terms.

The node configuration window has two setting tabs: “General options” and “Tagger options”. The “General options” tab is the tab available for every tagger node (see Fig. 3.2) and requires input, output, and tokenizer settings.

„Tagger options“ Tab

- *Dictionary column* requires the input String column with the list of words to match
- *Set named entities unmodifiable*. This option sets the named entity tags as unmodifiable. That is, whenever any subsequent tagger node in the workflow will not affect the previously tagged named entities.
- *Case sensitive*. The match must be case sensitive, if this option is checked
- *Exact match* flag refers to the entity matching exactly the dictionary word or just containing it.
- *Tag type* drop-down menu specifies the tag type for the tagging process
- *Tag value* drop-down menu specifies the tag value for the tagging process. This is based on the tag type selected in the previous drop-down menu.

Figure 3.11. Configuration window of the Dictionary Tagger node: “Tagger options” tab



We have a new example workflow to show the usage of the Dictionary Tagger node: workflow Chapter3/02_Custom_Taggers. As usual, as starting point we use the KNIME Forum dataset containing all posts and comments published between May 31st 2013 and May 31st 2017. After transforming the Strings texts into Documents, we decide to tag the words according to their sentiment. We use two list of words (dictionaries), one for the positive words and one for the negative words. The idea is to find such words in the Documents and tag them with a SENTIMENT value either POSITIVE or NEGATIVE. To do that, two Dictionary Tagger nodes are introduced: one tags the positive words, while the other tags the negative words (Fig. 3.13).

The two word lists come from the MPQA Opinion Corpus. The MPQA Corpus contains news articles from a wide variety of sources manually annotated for opinions and other states (i.e., beliefs, emotions, sentiments, speculations, etc.). The datasets provided by the MPQA Opinion Corpus are copyrighted by the MITRE Corporation.

Note. Since we use two *Dictionary Tagger* nodes consecutively, the flag "Set named entities unmodifiable" must be enabled in the first *Dictionary tagger* node, in order to keep the tagged named entities from the first node.

The Widcard Tagger node tags all terms in a text matching wildcard or regular expressions. The match can be term based or sentence based. A term based match means that the comparison applies only to single terms. A sentence based match means that the comparison applies to any word in a sentence.

If part of a sentence matches a wildcard expression, all sentence terms will be tagged. If there are multiple possible matches, the last match overwrites the tags from previous matches. On the opposite, in case of a regular expression, the term is tagged if and only if the expression matches the term completely. If a substring of the sentence matches the regular expression, all its terms will be tagged as one term. In this way, multi-words like "data mining" can be tagged as one single word. All sentences are tagged as long as there are no conflicts with other regular expressions.

A wildcard expression includes some special characters, "*" and "?". Adding * to a word piece means that any sequence of characters can appear at that point to complete the word. For example, wildcard expression "extr*" includes words like "extra", "extract", "extraction", "extracting", and whatever other word starts with "extr". Wildcard expression "*rable" covers all words ending in "rable", like "considerable" or "admirable". Wildcard expression "a*r" covers all words starting with "a" and ending with "r", like for example "author". Special character "?" has a similar functionality to "*", but covers only one character. So "extr?" covers "extra" but not "extraction"; "a?r" does not cover "author" anymore, but "a?t" might cover word "art", and so on.

Regular expressions, or RegEx, are a more complex way of representing rules for Strings, using bars, special characters, and brackets. A tutorial for regular expressions can be found at the Regular-Expressions site.

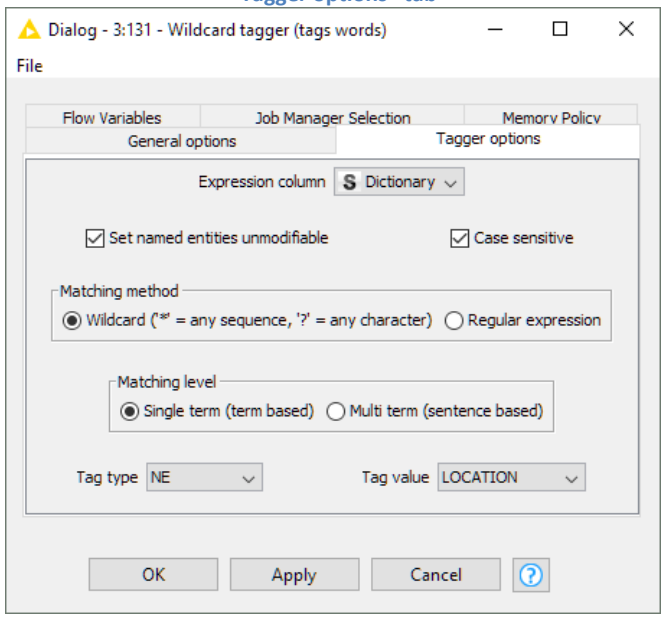
We then expanded the workflow Chapter3/02_Custom_Taggers with a *Wildcard Tagger* node. The *Wildcard Tagger* node is set to tag with NE (Named Entity) value ORGANIZATIONS terms matching the following wildcards: "dat*", "extr*", "scien*". "dat*" should identify words like "data", "database", and other similar words.

Wildcard Tagger

This node takes a dictionary column containing wildcard or regular expressions as second input, matches terms in the Document with the expressions in the dictionary, and assigns a specified tag to the matching terms.

The node configuration window has two setting tabs: “General options” and “Tagger options”. The “General options” tab is the tab available for every tagger node (see Fig. 3.2) and requires input, output, and tokenizer settings.

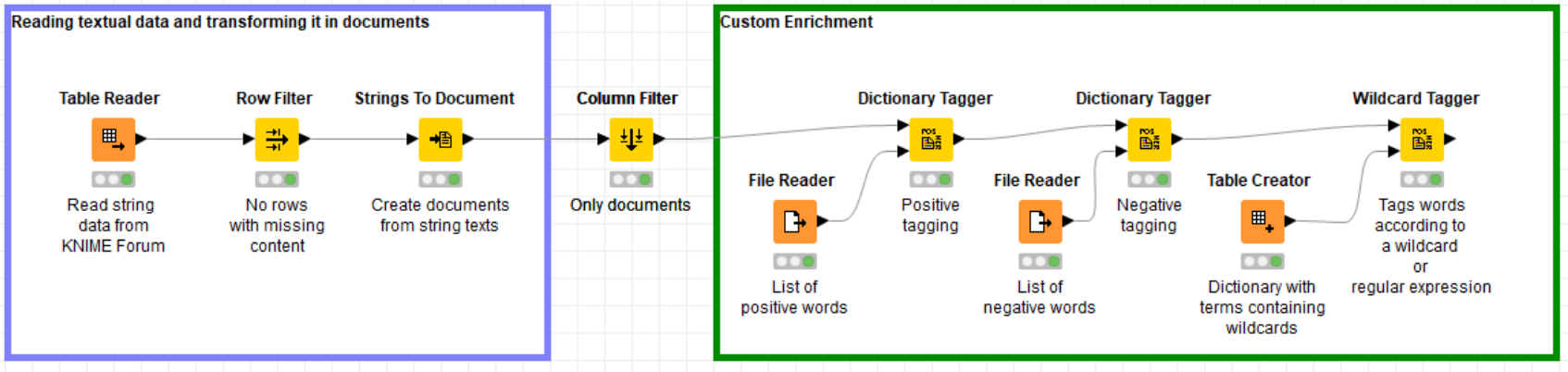
Figure 3.12. Configuration window of the Wildcard Tagger node: “Tagger options” tab



„Tagger options“ Tab

- *Expression column* requires the String column from the second input port containing the wildcard or regular expressions to be used for matching
- *Set named entities unmodifiable*. This option sets the named entity tags as unmodifiable. That is, whenever any subsequent tagger node in the workflow will not affect the previously tagged named entities
- *Case sensitive*. The match must be case sensitive, if this option is checked
- *Matching method* can be based on wildcard or regular expression
- *Matching level* The expressions can be matched with single terms ('Single term') or with whole sentences ('Multi term') level
- *Tag type* drop-down menu helps to specify the type of tag that is applied in the tagging process
- The *Tag value* drop-down menu specifies the tag value to use for tagging recognized named entities. This is based on the tag type selected in the previous drop-down menu.
- *Tag type* drop-down menu specifies the tag type for the tagging process
- *Tag value* drop-down menu specifies the tag value for the tagging process. This is based on the tag type selected in the previous drop-down menu.

Figure 3.13. Workflow Chapter 3/02_Custom_Taggers applies two Dictionary Tagger nodes to tag specific words from a list with SENTIMENT POSITIVE or NEGATIVE. At the end it applies a Wildcard Tagger node to tag words as NE according to a list of wildcard expressions. The workflow runs on the KNIME Forum dataset.



3.3. Filtering

After the enrichment step, we are now ready to clean up our Documents; that is we are now ready to introduce filters. Specifically, we would like to get rid of all those terms that do not carry much semantic information, such as stop words, numbers, punctuation signs, etc. Indeed, removing those terms reduces the dimensionality of the term space and makes the Documents heavier with information.

Of course, KNIME Analytics Platform offers many filtering nodes, to filter out punctuation signs, numbers, stop words, words with specific tags, words with a specific wildcard/RegEx pattern, and more. In the following sections, we will present some of these filtering nodes.

All filtering nodes require a Document type input column and a Document type output column. These two settings have been collected, together with the "Ignore unmodifiable flag" option, in a single tab in the configuration window of any filtering node. You will find this same tab (Fig. 3.14) – named "Preprocessing" - in the configuration window of all filtering nodes.

Let's start with the least complex filtering node: the *Punctuation Erasure* node. This node removes all punctuation characters in the Documents, such as: '!"#$%&()*+,-./:;<=>?@[\]^_`{|}~\t\n',\t,\n, where \t and \n corresponds respectively to tabs and new lines. The Punctuation Erasure node has the simplest configuration window, consisting of only the "Preprocessing" tab.

If we want to remove numbers from our input Documents, we need to use the *Number Filter* node.

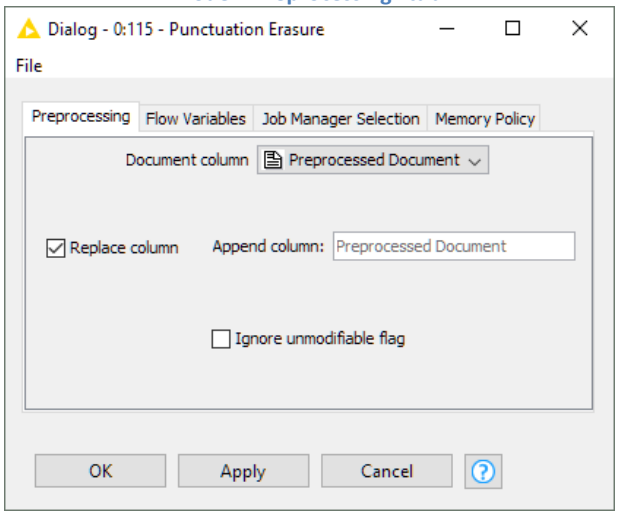
Punctuation Erasure

This node filters all punctuation characters from the input Documents.

„Preprocessing“ Tab

- *Document column* dropdown menu selects the input Document column to work on
- *Replace column* checkbox, if checked, replaces the input Document type column with the newly processed output Document type column. If this flag is not checked, a new column is created and appended to the input data table. The name of this new output column is required in the field *Append column*.
- *Ignore unmodifiable,* if checked, overwrite the unmodifiable flag of a term, if any, and overwrites the term with its new pre-processed version.

Figure 3.14. Configuration window of the Punctuation Erasure node: “Preprocessing” tab



Number Filter

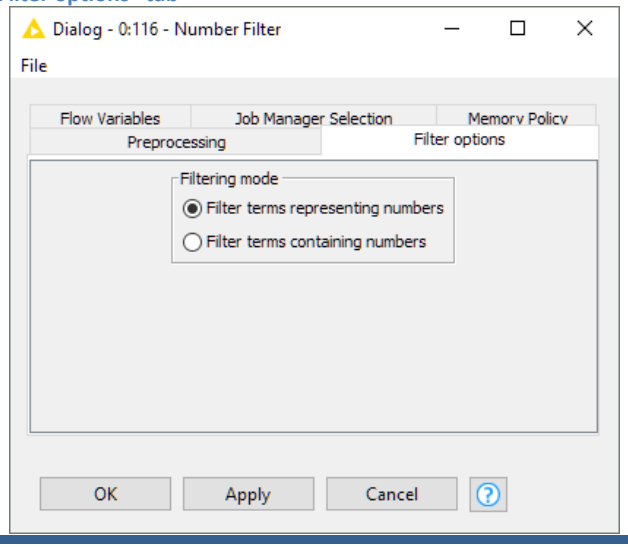
The Number Filter node filters all terms in forms of digits, including decimal separators “,”, or “.” and possible leading “+” or “-“.

The node configuration window has two setting tabs: “Preprocessing” and “Filter options”. The “Preprocessing” tab is available for every filtering node (see Fig. 3.14) and requires input, output, and “Ignore unmodifiable flag” option.

„Filter options“ Tab

- The *Filtering mode* gives the chance to:
  - *Filter terms representing numbers*, like “1234”
    or
  - *Filter terms containing numbers*, like “prefix_1234”

Figure 3.15. Configuration window of the Number Filter node: “Filter options” tab



To remove words carrying little information, the so called *stop words*, we can use the *Stop Word Filter* node. There is some general agreement as to which words can be considered as stop words. This is why the Stop Word Filter node contains a few built-in lists of commonly accepted stop words in English, French, German, Bulgarian, Hungarian, Italian, Polish, Portuguese, Romanian, Russian, and Spanish. However, often the concept of stop word depends on the context. Thus, the node also allows to load custom lists of stop words from text file. Each line in this custom text file contains only one stop word.

Stop Word Filter

This node removes stop words from the input Documents. Stop words are retrieved from a built-in list or from a custom list in an input column at the second optional input port.

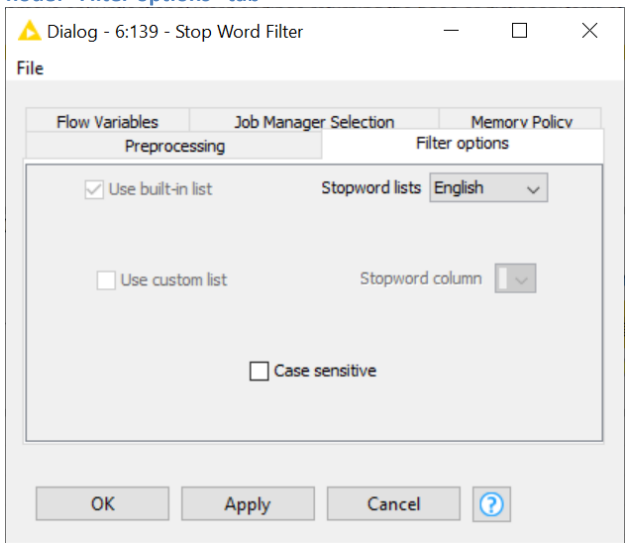
The node configuration window has two setting tabs: “Preprocessing” and “Filter options”. The “Preprocessing” tab is available for every filtering node (see Fig. 3.14) and requires input, output, and “Ignore unmodifiable flag” option.

„Filter options“ Tab

- *Case sensitive* checkbox, if checked, performs a case sensitive match between Document terms and stop words
- *Use built-in list*, if checked, retrieves the stop words from one of the available built-in lists. The built-in list of stop words in the desired language can be selected through the *Stopword lists* dropdown menu.
- *Use custom list,* if checked, the node requires a custom list of stop words in an input column selected in “*Stopword column”* menu.

Note that “Use custom list “ and “Use built-in list” flags are enabled only if an input table is present at the second optional input port.

Figure 3.16. Configuration window of the Stop Word filter node: “Filter options” tab



Another text processing node that might be useful is the *Case Converter* node. This node, as the name says, converts all terms contained in the Documents to upper case or lower case. Even in this case the configuration settings are straightforward.

Case Converter

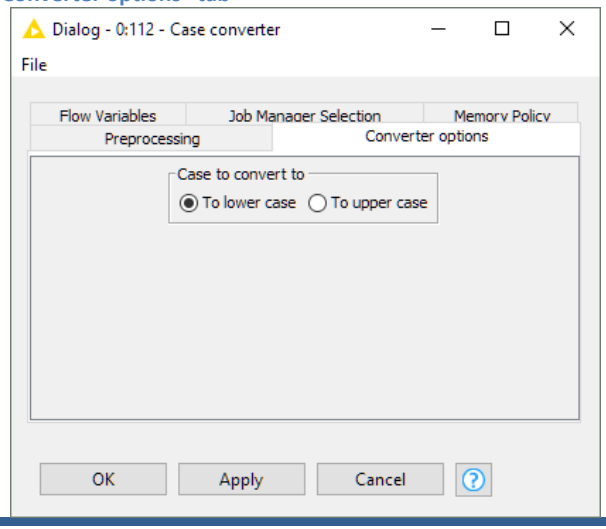
This node converts all terms in a Document to upper case or lower case.

The node configuration window has two setting tabs: “Preprocessing” and “Filter options”. The “Preprocessing” tab is available for every filtering node (see Fig. 3.14) and requires input, output, and “Ignore unmodifiable flag” option.

„Converter options“ Tab

- Options in “*Case to convert to*” radio buttons convert terms:
  - *To lower case*;
  - *To upper case*.

Figure 3.17. Configuration window of the Case Converter node: “Converter options” tab



The last node that we want to introduce in this section is the *Tag Filter* node. This node removes terms in a Document based on their tag. A term is removed if at least one of its tags matches the filter tags.

A common usage of this node is to keep only terms with specific POS tags, like NN for nouns, VB for verbs, or JJ for adjectives, and get rid of the words with the remaining tags. This allows to keep only the words that supposedly are semantically important. Another classic usage is to keep only terms with specific NE tags, such as LOCATIONS or PERSON. This allows to detect only those texts talking about given cities or people.

Workflow Chapter3/03_Filter_ Stemming_Lemmatization cleans up the input Documents before moving to the stemming operation. This cleaning up phase is largely based on the filter nodes described in this section, such as Punctuation Erasure, Number Filter, Stop Word Filter, Case converter, and Tag Filter node.

Note. We used the Punctuation Erasure node first, in order to remove punctuation pollution in future node operations.

Tag Filter

This node removes all terms in a given documents with specific tags.

The node configuration window has two setting tabs: “Preprocessing” and “Filter options”. The “Preprocessing” tab is available for every filtering node (see Fig. 3.14) and requires input, output, and “Ignore unmodifiable flag” option.

„Tag filter“ options Tab

- *Strict filtering* checkbox enforces strict filtering, i.e. terms are removed if all specified tags are present. If this option is not checked, terms are removed if at least one of the specified tags is present.
- *Filter matching* is an include / exclude option
- The *Tag type* dropdown menu sets the type of tags to filter
- The *Tags* list menu selects one or more tags to filter

Figure 3.18. Configuration window of the Tag filter node: “Tag filter options” tab

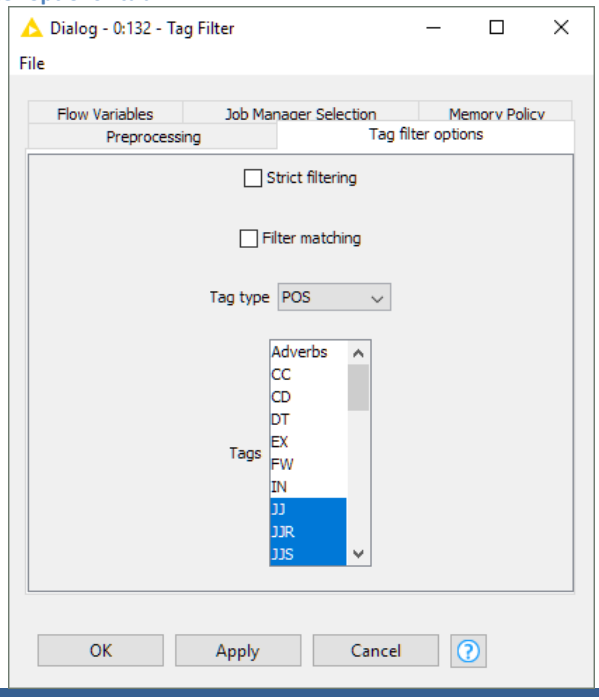
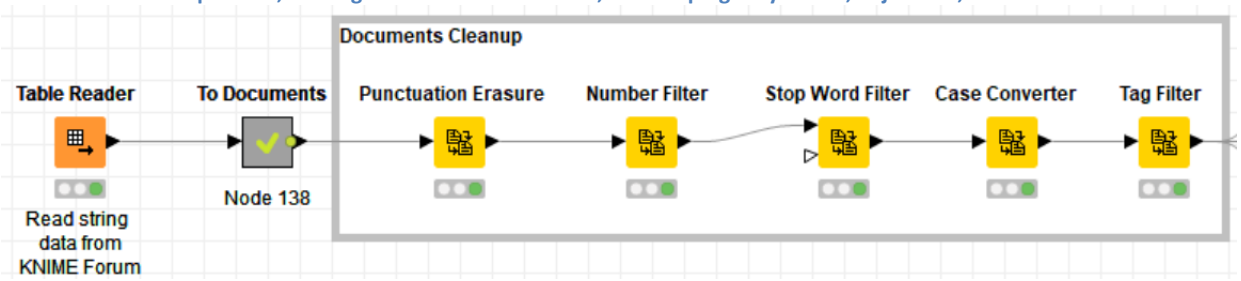


Figure 3.19. Workflow Chapter3/03_Filter_ Stemming_Lemmatization cleans up the posts and comments in the KNIME Forum dataset. By removing punctuation marks, numbers, stop words, moving all words to lower case, and keeping only nouns, adjectives, and verbs.



3.4. Stemming and Lemmatization

For grammatical reasons, texts use different forms of a word. As an example, in the English vocabulary we have words like “run”, “runs”, and “running”, which all have the same root, i.e. “run”. Furthermore, the English vocabulary has families of *derivationally related words* with similar meanings, such as “personal”, “personally”, “personality”. Normalizing terms to their roots might make for an easier information extraction process.

Here is where stemming and lemmatization come in. The goal of both procedures is to reduce inflectional forms and derivationally related forms to a common base form, as it is shown below:

- am, are, is → be
- book, books, book’s, books’ → book

By implementing stemming or lemmatization, the mapping of the text would become something like:

- those books are mine → those book be mine

Obviously, the two words – “are” and “books” - used in this example are not in their exact grammar form, but the general meaning of the statement has not changed.

Stemming refers to a crude heuristic process that chops off the ends of words in the hope of removing inflectional forms and derivational affixes. This chopped off final form of the word is called “stem”.

Lemmatization has the same goal as stemming, but proceeds in a less crude way with the help of a vocabulary and of the morphological analysis of words. The idea here is to return the base or dictionary form of a word. This root word is also known as “lemma”.

Note. While the lemma is a real word, the stem is usually just a part of a word and as such might not be easy to read and understand. For example, the verb “to promise” refers to the root word “promise” and to the root stem “promis[]”.

In general, lemmatization requires to consult a dictionary and this might help for difficult word inflections. For example the verb “to book” - which may appear as *book*, *booked*, *books*, and *booking* - has lemma “book” and stem “book[]”, which is close enough. Let’s take now the word “better”. Its lemma is “good” and its stem cannot be found. This is because the search for the root word requires a dictionary look-up. For this borderline cases, lemmatization might be useful and maybe lead to slightly higher accuracies.

All lemmatization and stemmer nodes require a Document type input column and a Document type output column. These two settings have been collected, together with the “Ignore unmodifiable flag” option, in a single tab, named “Preprocessing”, in the configuration window of all these nodes (Fig. 3.21).

KNIME Analytics Platform offers only one lemmatizer node and a few stemmer nodes. Let’s start with the simplest stemmer node: the Porter Stemmer node.

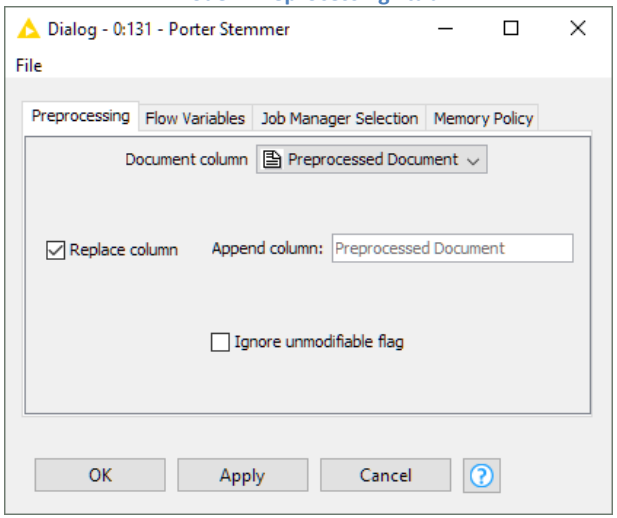
Porter Stemmer

This node stems terms from the input English Documents using the Porter stemmer algorithm [9].

“Preprocessing“ Tab

- *Document column* dropdown menu selects the input Document column to work on
- *Replace column* checkbox, if checked, replaces the input Document type column with the newly processed output Document type column. If this flag is not checked, a new column is created and appended to the input data table. The name of this new output column is required in the field *Append column*.
- *Ignore unmodifiable,* if checked, overwrite the unmodifiable flag of a term, if any, and overwrites the term with its new pre-processed version.

Figure 3.20. Configuration window of the Porter Stemmer node: “Preprocessing” tab



The Porter’s algorithm is the most common algorithm for stemming for the English language and has been proven very effective in a number of studies [9]. Porter's algorithm reduces a word to its stem in 5 sequential steps. The first step is divided in two different sub-steps and deals mainly with plurals and past participle. Below is one of the rule groups applied during the first step.

Rule	Example
SSES → SS	caresses → caress
IES → I	ponies → poni
SS → SS	caress → caress
S →	cats → cat

In the following steps, the rules to remove a suffix are given in the form: *(condition)* S1 → S2, which means that if a term ends with the suffix S1, and the stem before S1 satisfies the given condition, S1 is replaced by S2 [9]. Each step also checks the stem and the syllable length. The algorithm does not remove a suffix when the stem is already too short.

The Porter Stemmer node only applies to English Documents. To overcome this limitation, the Snowball stemmer algorithm was introduced by the same authors of the Porter stemmer [9]. Stemming is not applicable to all languages, like Chinese. However, it works for many Indo-European languages, like all Romance languages (Italian, French, Spanish...), Slavonic languages (Russian, Polish, Czech...), Celtic languages (Irish Gaelic, Scottish Gaelic, Welsh...), German languages (German and Dutch). The Snowball stemmer node offer a stemming option for all those languages.

Note. For English documents, the Porter stemmer is still the best performing stemmer.

Snowball Stemmer

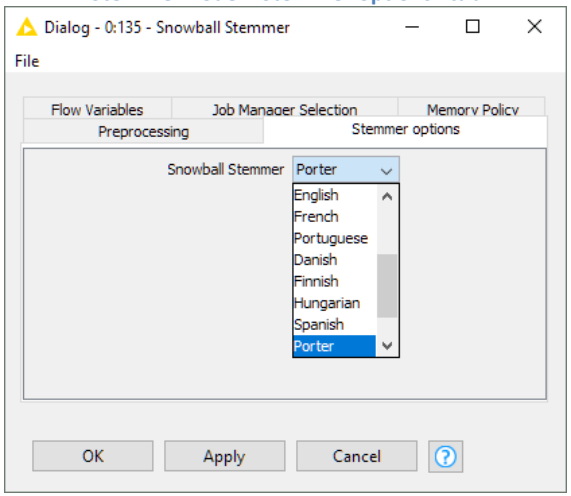
This node reduces the terms from the input documents to their stems using the Snowball stemming library. This stemmer includes models for a number of languages.

The node configuration window has two setting tabs: "Preprocessing" and "Stemmer options". The "Preprocessing" tab is available for every stemming node (see Fig. 3.21) and requires input, output, and "Ignore unmodifiable flag" option.

"Stemmer options" Tab

- The *Snowball Stemmer* dropdown list menu selects and loads the Snowball Stemmer for the required language.

Figure 3.21. Configuration window of the Snowball Stemmer node: "Stemmer options" tab



Another stemmer node is the Kuhlen Stemmer node. This node implements the Kuhlen stemmer algorithm for English Documents only [10].

Let's finish this section with the description of the Stanford Lemmatizer node.

Stanford Lemmatizer

This node reduces terms in the input Documents to their lemma using the Stanford Core NLP library [11].

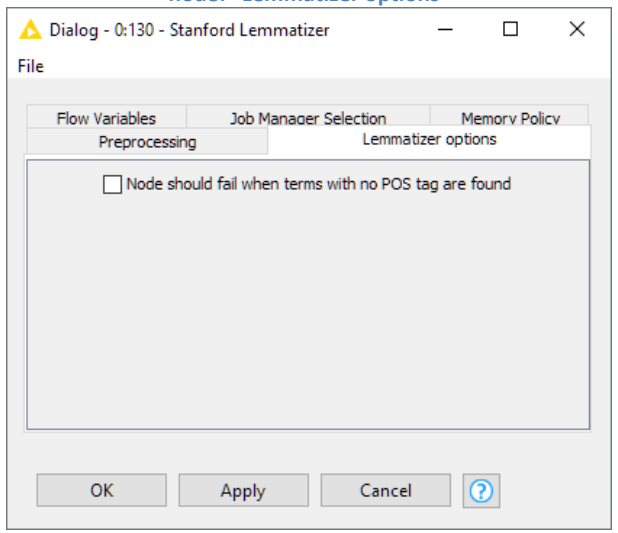
The node configuration window has two setting tabs: "Preprocessing" and "Stemmer options". The "Preprocessing" tab is available for every stemming node (see Fig. 3.21) and requires input, output, and "Ignore unmodifiable flag" option.

„Lemmatizer options" Tab

The analysis is mostly based on Part-Of-Speech (POS) tags. Therefore, either the Stanford Tagger node or the POS Tagger node has to be applied before the Stanford Lemmatizer node. If more than one POS tag is found on one term, only the first tag will be considered.

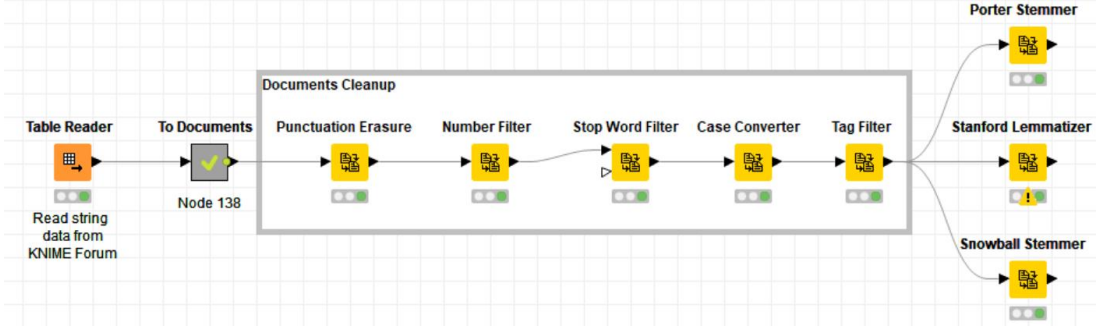
- If option "*Node should fail …*" is checked, the node fails when at least one term has no POS tag. By default, terms with no POS tag are skipped.

Figure 3.22. Configuration window of the Stanford Lemmatizer node: "Lemmatizer options"



Building on the Chapter3/03_Filter_Stemming_and_Lemmatization workflow, we added a lemmatizer and two stemmer nodes, to respectively extract lemmas and stems from the words in the posts and comments from the KNIME Forum dataset. Notice that the lemmatizer node and the stemmer nodes have been applied after all filtering nodes. Indeed, filtering out less important words before applying the stemming and lemmatization algorithm reduces the text dimensionality and the computation expenses of the algorithms.

Figure 3.23. Workflow Chapter3/03_Filter_ Stemming_Lemmatization extracts the stems and lemmas from the terms in the posts and comments in the KNIME Forum dataset.



3.5. Bag of Words

So far, we have tagged words, cleaned the data by filtering, and stemmed or lemmatized terms, but we never saw the effect on the input Documents, since the data table view does not allow much of an inspection into a Document type cell. One way to get an insight on what has been left inside the Document, after all those operations, is to use the get the list of the words in it, i.e. to build the Document's bag of words (BoW). We have reached the moment to unveil the Bag Of Words Creator node.

BoW is a technique used in natural language processing and information retrieval to disaggregate input text into terms. Putting it in a practical way, the BoW technique represents Documents with the bag of their words. With this process the grammar is disregarded and even the order of the words is not taken into account, but the multiplicity is kept. Using the BoW representation, Documents with similar bag of words are supposed to be similar in content. As an example, let's build the Bag of Words for this unfiltered sentence.

```
Mark likes to play basketball. Peter likes basketball too.
```

For this text Documents, the Bag of Words is the following:

```
[
    "Mark",
    "likes",
    "to",
    "play",
    "basketball",
    "Peter",
    "too"
]
```

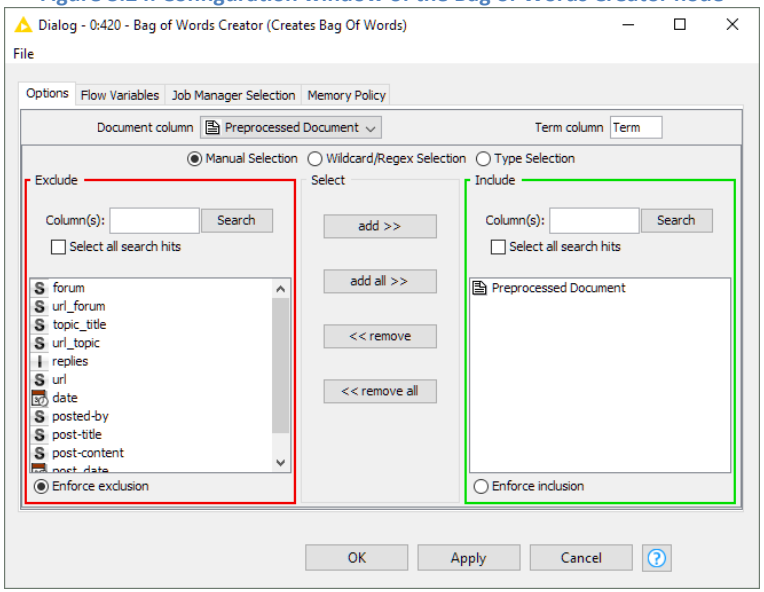
Bag Of Words Creator

This node creates a bag of words (BoW) for each input Document. The Bags of Words are then produced in an output Term-type column.

Options Tab

- *Document column* dropdown menu selects the Document column to be represented with the bag of words.
- *Term column* text field gives the name of the output Term column containing the Bag of Words.
- *Column Selection* framework selects the columns to carry on to the output table. The columns in the "Exclude" panel, will not be available in the output table.

Figure 3.24. Configuration window of the Bag of Words Creator node



A Term-type output column, though, is also not so easy to inspect when visualizing a data table, especially if Terms contain many tags. For better visual clarity, we could export the Terms back to String words using the **Term To String** node. Configuration setting of the Term To String node is straightforward, since the node only requires the name of the input Term-type column.

Similar to the Term To String node, there is also the **String To Term** node implementing the backward transformation, i.e. transforming String words to Term objects. Configuration setting of this node is also straightforward, since it just requires the name of the String type column to be transformed into a Term-type column.

Let's now wrap enrichment, filtering, lemmatization, and Bag of Words into one single workflow, stored in Chapter3/04_BoW_and_Terms (Fig. 3.26).

In this workflow, the Text Processing phases previously explored are wrapped into metanodes: conversion of tests into Documents in the "To Documents" metanode, POS tagging in "Enrichment" metanode, general cleaning with punctuation erasure, number filtering, stop word filtering, and tag filtering in "Pre-processing" metanode. Rememeber that we set the Tag Filter node to keep only nouns, adjectives, and verbs. The Stanford Lemmatizer is then applied to reduce words to their roots.

At this point in the workflow, we would like to verify that:

- The Tag Filter node really only kept nouns, verbs, and adjectives
- There are no punctuation marks
- The root of words were extracted during the lemmatization process.

To check on all those facts, we split the flow in two parallel branches: one applies lemmatization, the other does not. The resulting Bags Of Words from these two branches are then joined together, for comparison (Fig. 3.27). In the final output table, we kept 4 columns: the original Term, the corresponding original String obtained with a Term To String node, the lemmatized Term, and the corresponding lemmatized String.

In all columns, we see no punctuation marks and no stop words, such as “and” or “of”. We also see that all words are nouns, verbs, or adjectives as their POS tags confirm. This confirms that all nodes in the previous pre-processing phase have worked as we expected.

Let’s see now the effect of lemmatization. Let’s check for example the word “following” in the original Term/String column and see that it has become just “follow” in the lemmatized Term/String column, i.e. it has been reduced to its lemma, as it was also to be expected.

Figure 3.25. Workflow Chapter3/04_BoW_and_Terms applies two Bag Of Words Creator nodes to investigate and compare results of text pre-processing for the KNIME Forum dataset on two parallel branches. Comparing the resulting Bags of Words we see the effects of nodes like: Punctuation Erasure, Tag Filter, Stanford Lemmatizer, Stop Word Filter, etc …

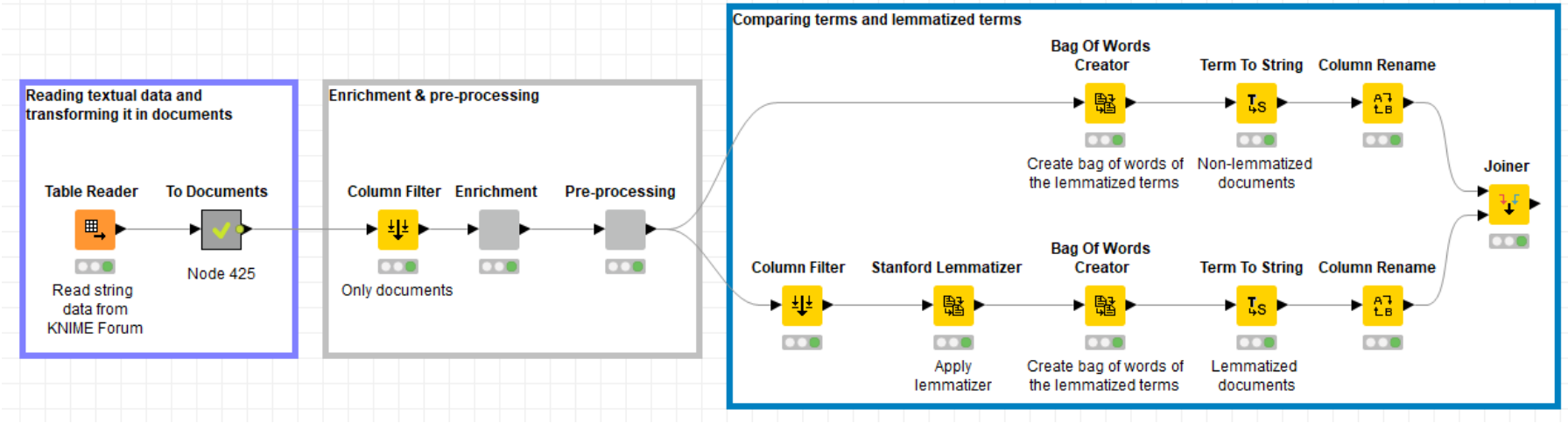
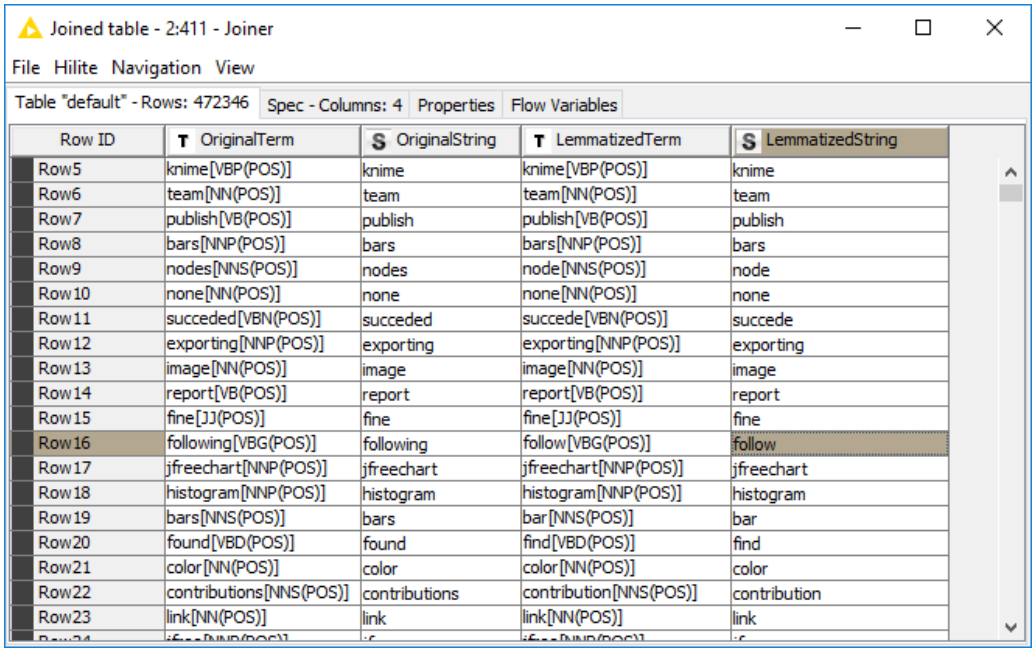


Figure 3.26. Output table of the last Joiner node in workflow Chapter3/04_BoW_and_Terms. See the lemmatization effect on the original word “following” reduced to “follow” by the Stanford Lemmatizer node.



Row ID	OriginalTerm	OriginalString	LemmatizedTerm	LemmatizedString
Row5	knime[VBP(POS)]	knime	knime[VBP(POS)]	knime
Row6	team[NN(POS)]	team	team[NN(POS)]	team
Row7	publish[VB(POS)]	publish	publish[VB(POS)]	publish
Row8	bars[NNP(POS)]	bars	bars[NNP(POS)]	bars
Row9	nodes[NNS(POS)]	nodes	node[NNS(POS)]	node
Row10	none[NN(POS)]	none	none[NN(POS)]	none
Row11	succeded[VBN(POS)]	succeded	succede[VBN(POS)]	succede
Row12	exporting[NNP(POS)]	exporting	exporting[NNP(POS)]	exporting
Row13	image[NN(POS)]	image	image[NN(POS)]	image
Row14	report[VB(POS)]	report	report[VB(POS)]	report
Row15	fine[JJ(POS)]	fine	fine[JJ(POS)]	fine
Row16	following[VBG(POS)]	following	follow[VBG(POS)]	follow
Row17	jfreechart[NNP(POS)]	jfreechart	jfreechart[NNP(POS)]	jfreechart
Row18	histogram[NNP(POS)]	histogram	histogram[NNP(POS)]	histogram
Row19	bars[NNS(POS)]	bars	bar[NNS(POS)]	bar
Row20	found[VBD(POS)]	found	find[VBD(POS)]	find
Row21	color[NN(POS)]	color	color[NN(POS)]	color
Row22	contributions[NNS(POS)]	contributions	contribution[NNS(POS)]	contribution
Row23	link[NN(POS)]	link	link[NN(POS)]	link

3.6. Helper Nodes

This last section describes a few additional helper nodes in the Text Processing category.

An useful helper node is the *Document Data Extractor* node. Do you remember that, when transforming a text from a String cell to a Document cell, we were adding extra information? Indeed, in the Strings to Document node we were adding the title, the authors, the content, the class, and more. Well, the Document Data Extractor node extracts this information back from the Document cell. From the input Document-type column, the node extracts for example Title, Text, and Author and appends them to the input data table.

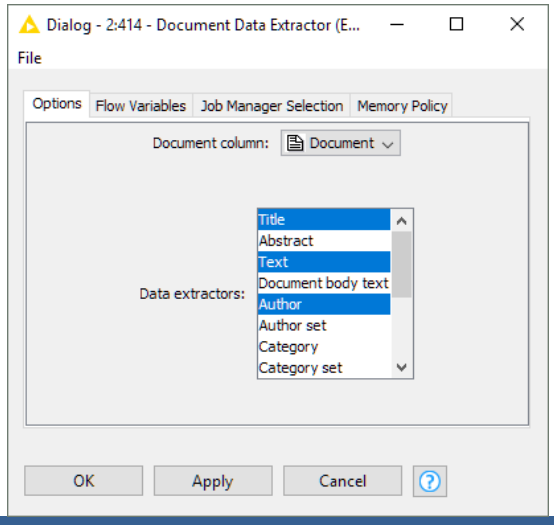
Document Data Extractor

This node extracts information from a Document cell, such as title, text content, author, etc …. Extracted data values are stored in new corresponding columns which are then appended to the input data table.

Options Tab

- *Document column* dropdown menu selects the Document-type column to serve as input.
- *Data extractors* multi-selection menu selects one or more values to extract from the input Documents.

Figure 3.27. Configuration window of the Document Data Extractor node



Another helper node is the *Sentence Extractor* node, which transforms detects sentences and outputs them as Strings. Parsing single sentences instead of full texts reduces the computational load and constrains the possible analysis results.

Documents can also be identified via (key, value) pairs, which are stored in the Document meta information. The *Meta Info Inserter* node associates a (key, value) pair to each Document and stores it in its meta-data structure.

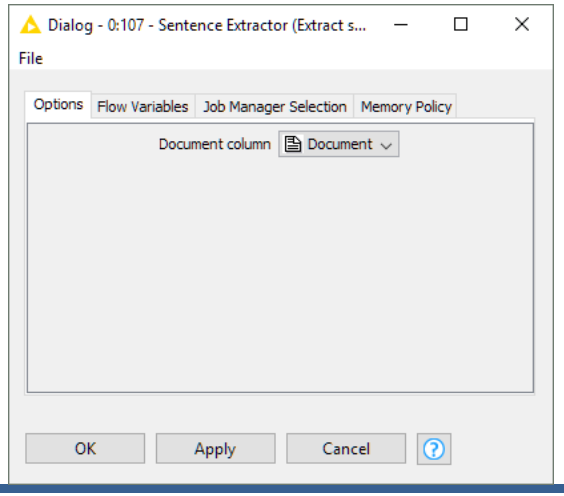
Sentence Extractor

This node extracts all sentences from the input Documents and outputs them in a String output column. The number of Terms contained in each sentence is output in an additional Integer column.

Options Tab

- *Document column* dropdown menu selects the Document-type column to use for sentence extraction.

Figure 3.28. Configuration window of the Sentence Extractor node



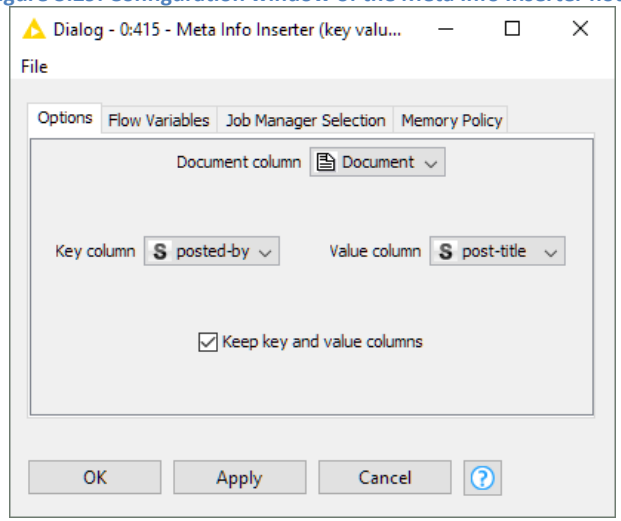
Meta Info Inserter

This node inserts (key, values) pairs in Document meta information.

Options Tab

- *Document column* dropdown menu selects the Document-type input column
- *Key column* dropdown menu selects the input column containing the keys for the pair.
- *Value column* dropdown menu selects the input column containing the values for the pair.
- Checkbox *Keep key and value column,* if enabled, outputs the key and the value columns also in the output table.

Figure 3.29. Configuration window of the Meta Info Inserter node



The (key, value) pair inserted in the Document meta-data with the Meta Info Inserter node can be extracted back by using the *Meta Info Extractor* node. Here, the user can decide whether to ignore duplicate input Documents. In this case the meta information is extracted only for distinct Document cells.

The last helper node for this section is the *Tags to String* node. This node transforms Term tags into Strings.

Workflow Chapter3/05_Sentence_Extractor summarizes the usage of some of the helper nodes described in this section.

Tag to String

This node converts terms' tags into Strings for one or more selected tag types. The node outputs a number of additional String columns containing the original tags. If a tag value is not available, a missing value will be generated for the output String column.

Options Tab

- *Term Column* dropdown menu selects the input Term column.
- *Tag types* multi-selection menu selects one or more tag types to be used to identify the tags to export as String.
- The *Missing tag value* text field defines the strategy to adopt in case of missing tags. If it is set to <MissingCell>, all missing tags will become missing values.

Figure 3.30. Configuration window of the String to Term node

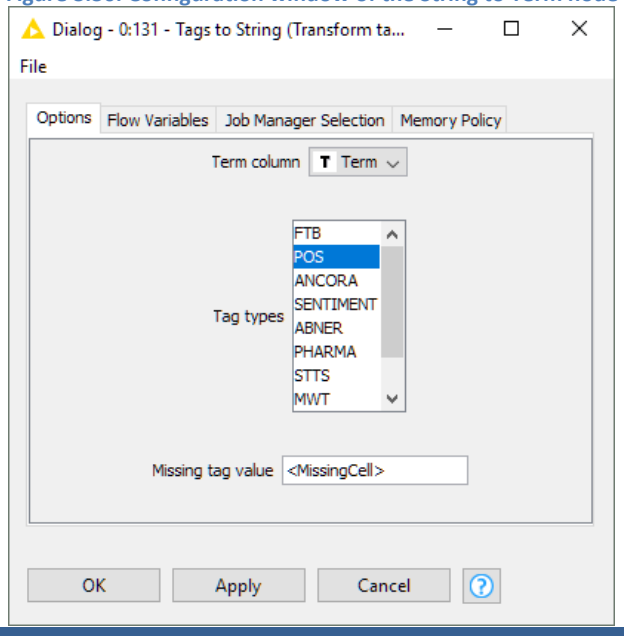
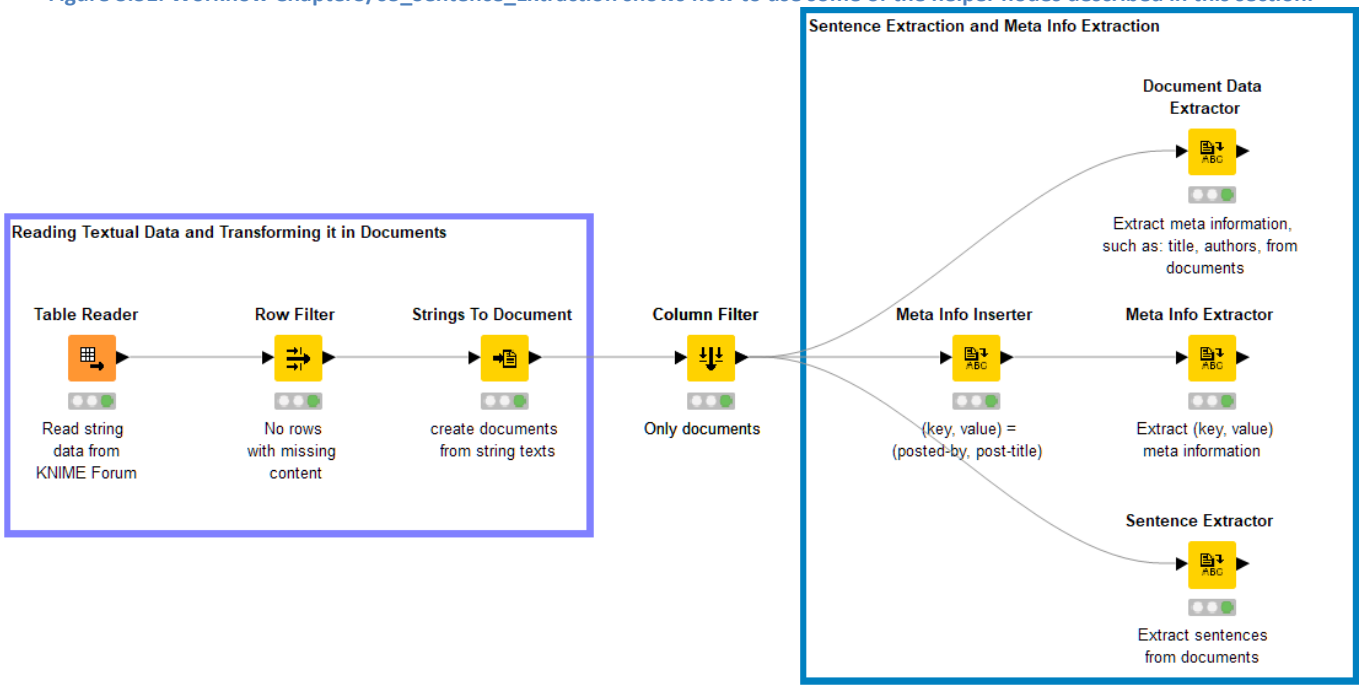


Figure 3.31. Workflow Chapter3/05_Sentence_Extraction shows how to use some of the helper nodes described in this section.



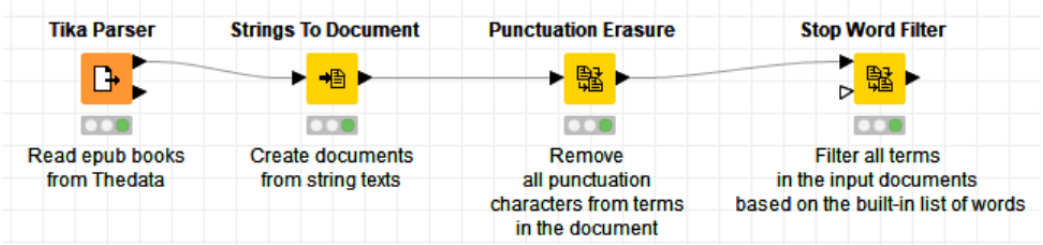
3.5. Exercises

Exercise 1

Read the content of the epub file Thedata\pg1513.epub containing the e-book of the tragedy "Romeo and Juliet". Remove punctuations and stop words.

Solution

Figure 3.32. Filtering - Punctuation Erasure and Stop Word Filter for the tragedy "Romeo and Juliet"



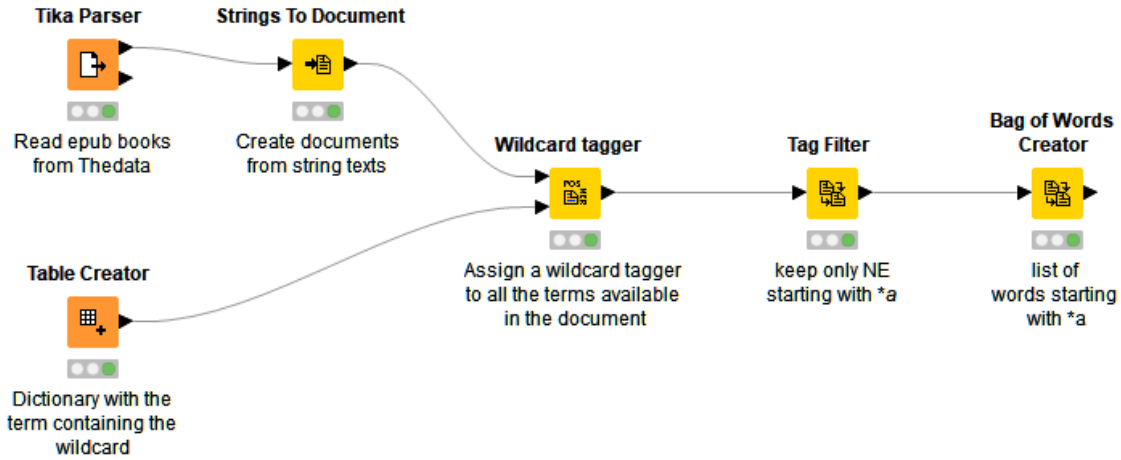
Exercise 2

Read the content of the epub file Thedata\pg1513.epub containing the e-book of the tragedy "Romeo and Juliet". Keep only words starting with “a” (case-sensitive) and then create the Bag of Words. How many unique words starting with “a” are there in the tragedy?

Solution

We have counted 210 words starting with “a” in the tragedy of “Romeo and Juliet” from the Thedata\pg1513.epub file.

Figure 3.33. A Wildcard Tagger extracts only the words in the "Romeo and Juliet" tragedy starting with “a”. The Bag Of Words Creator node makes them available at the output port.



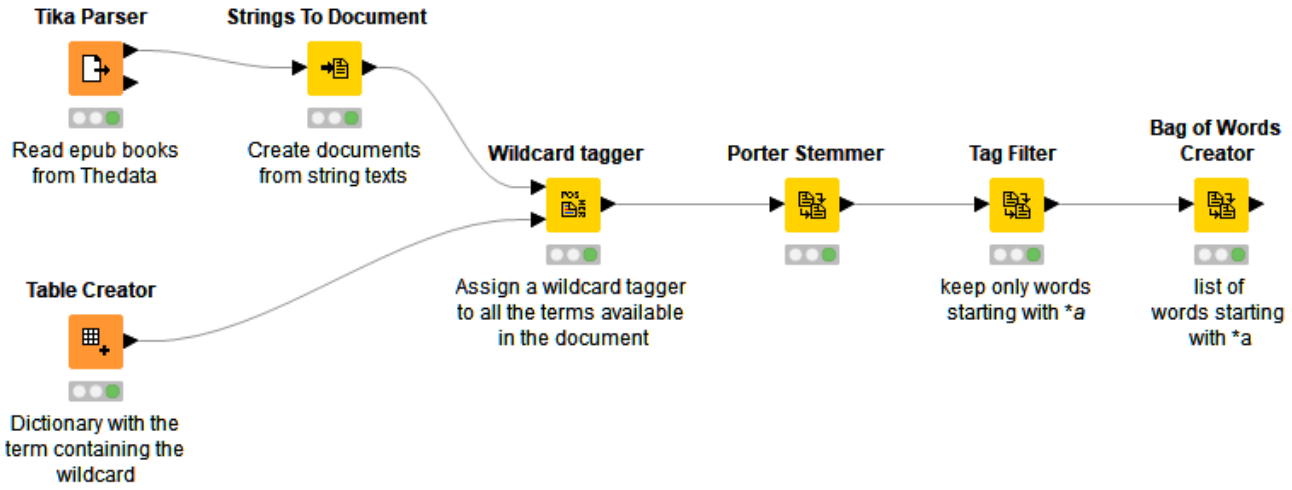
Exercise 3

Read the content of the epub file Thedata\pg1513.epub containing the e-book of the tragedy "Romeo and Juliet". Keep only words starting with “a” (case-sensitive). Apply stemming and then create the bag of words. Now, how many unique terms starting with “a” are there in the tragedy?

Solution

Using the Porter Stemmer node, we have counted still 210 words starting with “a” in the tragedy of “Romeo and Juliet” from the Thedata\pg1513.epub file.

Figure 3.34. The Porter Stemmer does not change the number of words in the "Romeo and Juliet" tragedy starting with “a”.



Chapter 4. Frequencies and Vectors

4.1. From Words to Numbers

Not all words in a Document are created equal. So far, we have cleaned up, enriched, and stemmed our words in a Document. At the end we have made a list of all the remaining words using the Bag Of Words Creator node. However, not all words carry the same amount of information. More frequent words, for example, are a clear indication of the text topic.

A review post, where the most recurring word is “food”, is probably talking about recipes or restaurants or some other food-related topic. If the post is on TripAdvisor then the choice is restricted to restaurants or food stores. If are reading reviews about restaurant though, the word “food” loses its descriptive importance, since all posts are probably talking about “food”. Thus, here other words, for example related to ethnicity and judgement, become more descriptive of the post topic.

All to say, that the descriptive power of a word in a text is usually based on some measure of its frequency. As you can imagine the scientific world has produced a number of different frequency measures, each one with its advantages and disadvantages. In this chapter we will describe some of the most commonly used word frequency measures, especially the ones that are also available within the KNIME Text Processing extension. We will then move from the Document representation based solely on the list of its words to a Document representation based on the pairs (word, frequency). The pair (word, frequency) should give more insights into the Document topic and sentiment.

The second effect of describing a text through its word frequencies consists of moving from a word based representation to a number based representation. A number based representation is where data scientists feel most comfortable. Indeed, once we are in the number space we can think of applying machine learning algorithms to classify and predict. At the end of this chapter, then we will show a few techniques that use such frequencies to prepare a suitable input data table for machine learning algorithms.

4.2. Word Frequencies

Let’s start with some measures of word frequency in a text. It is plausible to think that a term that occurs more often than others in a text is more descriptive and therefore should receive a higher frequency score. The problem now is how to define such term frequency scores.

The easiest way is, as usual, just counting. The number of occurrences of a term is its frequency score. The most often recurring term is also the most descriptive one. This is the **absolute frequency** and it works as long as we refer to same length texts. An example of same length texts are tweets. Given

the limitations in number of characters required by the Twitter platform, tweets cannot vary that much in length. In this case, the absolute frequency measures fits the data very well.

What about short vs. long texts? Review writers have each one their own style. Some especially like to write long texts, to repeat the same concept over and over or more reasonably to be very detailed in their review motivations. The same number of term occurrences in long texts cannot have the same weight as in short texts. In this case, we need to normalize the term absolute frequency by the total number of words in the text. This is the term **relative frequency**. Relative frequency fits well text corpora with variable text length.

Now let's suppose our text corpus includes only Documents about a specific topic, let's say restaurants. Term "food" will be frequent in each and every post. However, given the corpus theme, "food" would carry the relatively low information of an obvious word. Another frequency measure indeed takes into account how frequent a word is in the text and how infrequent the word is in the corpus. The idea is that a frequent word in the text, which at the same time is not much present in other texts, helps setting this Document apart from the others in the same corpus. Measures of this type are the **inverse document frequency** and the **inverse category frequency**.

We can then consider word co-occurrences (N-Grams) rather than single word (Uni-Grams) occurrences. We can consider other statistics based measures more elaborate than just counting. In this case, new frequency scores are defined. In the scientific literature, these frequency scores belong to the term weighting schemes, which have been widely used in information retrieval and text categorization models.

KNIME Analytics Platform provides a number of nodes to calculate a number of frequencies scores. We will describe them in this section, using the workflow stored in folder Chapter4/01_Frequencies.

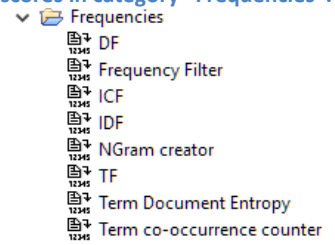


Figure 4.1. Nodes for frequency scores in category "Frequencies".

All nodes calculating frequency scores need an input Document-type column to work on. This column is usually selected in the tab named "Document Col" in the configuration window of all frequency nodes. This tab has only one setting: the dropdown menu of all input columns to select the Document type column to analyze.

Term absolute frequencies and relative frequencies are calculated by the TF node. The TF node outputs the original Document column from the input port, the bag of words for each Document cell, and the corresponding absolute or relative frequency for each one of the word in the Document cell. The configuration settings of the TF node does not allow to change the name of the output column containing the frequency measures. Default name for this column are "*TF abs*" for absolute frequencies and "*TF rel*" for relative frequencies.

Note. The frequency nodes require an input Term type column with the Document's Bag of Words, even though such column cannot be selected in the configuration window. This means that a Bag Of Words Creator node – or any other keyword generating node - must be applied before any frequency node.

TF

This node computes the absolute frequency or the relative frequencies of a term in each input Document cell.

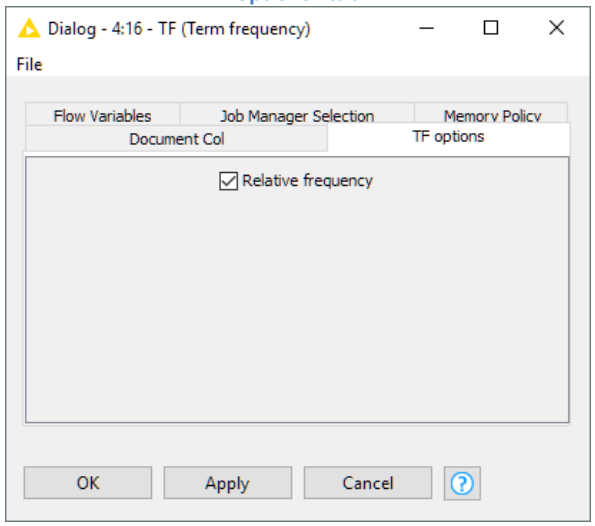
The absolute frequency counts the number of occurrences of a term in an input Document text. The relative frequency divides the absolute frequency by the total number of terms in the same Document text.

The configuration window has two tabs. The Document Col tab contains just a dropdown menu to select the input Document type column to work on.

„TF options“ Tab

- Checkbox *Relative frequency*, if checked, calculates term relative frequencies. If not checked, term absolute frequencies are calculated instead.

Figure 4.2. Configuration window of the TF node: “TF options” tab



Another group of frequency measures about the relevance of terms in Documents are the document frequencies. Document frequencies indicate the number of Documents all containing a specific term in the current Document group. The **DF** node performs such counting and outputs the input Document, its Bag of Words, and the DF frequency, i.e. the number of times a word appears in the input Document set.

In workflow *Chapter4/01_Frequencies*, after all required text preprocessing (Strings to Document, Punctuation Erasure, Number Filter, Stop Word Filter, Case Converter, Snowball Stemmer, Bag Of Words Creator), we applied a TF node to calculate the absolute frequencies for all terms in each Document bag of words. After the TF node we applied a DF node to count the number of occurrences of a term in the whole Document set. If we sort the output column TF abs and DF in descending order, we find that the most common word across all Documents is “node” (8442 times), while the most repeated word in a Document is “node” (2051 times) in the post with title “Opening node”. If we sort the relative frequencies in the same descending order, we find many words with highest relative frequency = 1.

Note. If we apply a POS Tagger, words might be split across different Terms in reason of different grammar roles and frequencies will be calculated for each one of these separated Terms. Thus, to fully appreciate the impact of an absolute frequency or a Document frequency, we avoided using POS tagging in this workflow.

However, just counting number of occurrences is often a too crude measure. Sparck Jones [12] proposed a more sophisticated measure to understand the term relevance for the single Document in a group of similar Documents: the **Inverse Document Frequency (IDF)**, aka as weighting function.

IDF counts the number of occurrences d_i of a given term t_i, being D the total number of Documents in the collection. Thus, according to [13], term t_i obtains a weight calculated as:

$$IDF(t_i) = log \frac{D}{d_i}.$$

Originally [12], the equation included an integer approximation of the logarithm in base 2 of the $\frac{D}{d_i}$ ratio, but it has been simplified to the current form over the years. The formula assigns a higher score, i.e. a stronger weight, to terms occurring less frequently across the Document collection (small d_i). The idea is that such words escaping the crowd better identify the Document they belong to.

Some other forms have been proposed over time to calculate the Inverse Document Frequency:

1) ***Normalized*** → $IDF(t_i) = log \frac{D}{d_i}$
2) ***Smooth*** → $IDF(t_i) = log\left(1 + \frac{D}{d_i}\right)$
3) ***Probabilistic*** → $IDF(t_i) = log \frac{(D - d_i)}{d_i}$

The smooth and the probabilistic forms of IDF avoid that a term is given a null weight just because it appeared in all Documents of the collection.

Continuing with the workflow Chapter4/01_Frequencies, we add an IDF after the TF node to calculate the normalized IDF measures. The IDF node produces an output column named IDF and containing the measures. If we sort the values in the IDF column in descending order, we find at the very top Terms like orgjfreechartapijavadocorgjfreechartrenderercategorystackedbarrender[], 881i[], or orgforumknime-usersparameter-iter[], which are obviously import mistakes and probably occur rarely across the Document collection. Scrolling down the sorted list of Terms, we find dropdown[] with IDF score 2.78 and much later on node[] with score 0.38. The score thus indicates the word “node” to occur much more often across the collection than the word “dropdown”.

However, from the presence of outlier words at the top of the list, we realize that IDF alone is also not a sufficiently descriptive measure. Common practice is then to use the product of IDF score and relative frequency from the TF node. The IDF score takes into account words that uniquely identify a Document, while the relative frequency rewards words occurring often within a text and therefore probably meaningful to describe its content. This new frequency measure is called **TF-IDF** or ***term frequency-inverse document frequency*** and it has proven to be more robust and reliable than the single measures alone. Formally for a Term *t* in Document *d:*

$$tf - idf_{t,d} = tf_d * idf_t$$

The formula produces:

- The highest value when Term *t* occurs often within a small number of Documents;
- lower values when Term *t* occurs fewer times within a Document or occurs in many Documents;
- the lowest value (which corresponds to zero) when *t* occurs in all the documents of the collection.

Many other forms of TF*IDF have been proposed. It is not the scope of this book to show all of them, but as reference we suggest to have a look at [14].

IDF

This node computes the Inverse Document Frequency of all Terms in a Document.

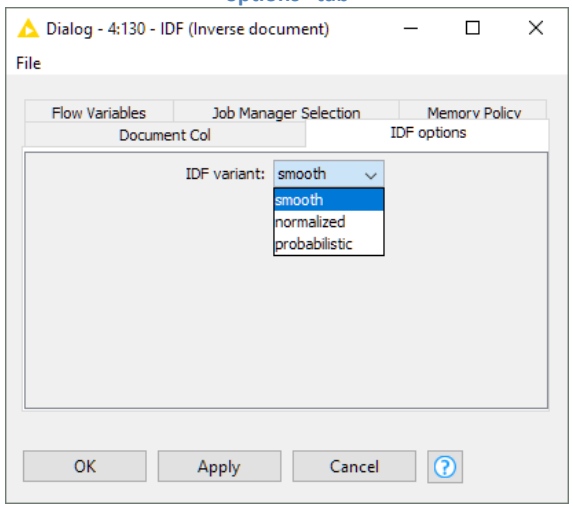
The configuration window has two tabs. The Document Col tab contains just a dropdown menu to select the input Document type column to work on.

„IDF options“ Tab

- *IDF variant* dropdown menu selects the variant of the Inverse Document Frequency to compute: normalized, smooth, or probabilistic according to the formula definitions given above.

The IDF node outputs the original Document column from the input port, the bag of words for each Document cell, and the corresponding Inverse Document Frequency value for each one of the Terms in the Document cell.

Figure 4.3. Configuration window of the IDF node: “IDF options” tab



In our example workflow, Chapeter4/01_Frequencies, we added a Math Formula node to calculate the TF-IDF product. Again sorting in descending order the TF*IDF measures in the output table uncovers *samaritan[],solución[],kilianthiel[],* and *god[]* as the top informative Terms at the top, identifying particularly useful contributions. A bit below, we find *login[]* and *unpivot[]* clearly identifying the problem described in the post.

After the Math Formula node we used a *Frequency Filter* node to extract those Terms with TF*IDF above 1. The output of the node contains only 679 Terms with relevant TF-IDF score.

Frequency Filter

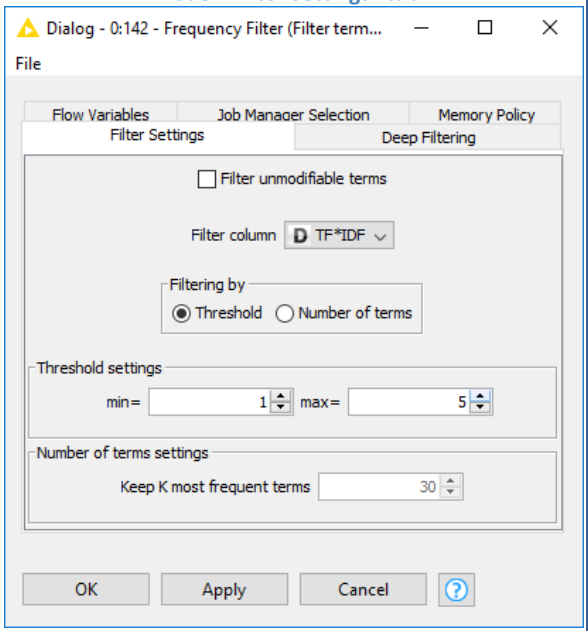
This node extracts those Terms with high frequency score from a collection of Documents.

The configuration window has two tabs. The Deep Filtering tab contains just a dropdown menu to select the input Document type column to work on.

„Filter Settings“ Tab

- *Filter unmodifiable terms* checkbox, if checked, allows to remove Terms that have been set as unmodifiable by previous nodes.
- *Filter column* selects the frequency score column to work on.
- *Filtering by* specifies whether Terms must filtered based on a frequency threshold or just by taking the top N items. Whatever option is chosen here activate one of the two next settings.
- *Threshold settings* define the interval within which to filter the input Terms
- *Number of terms settings* defines the number of top Terms to keep

Figure 4.4. Configuration window of the Frequency Filter node: “Filter Settings” tab



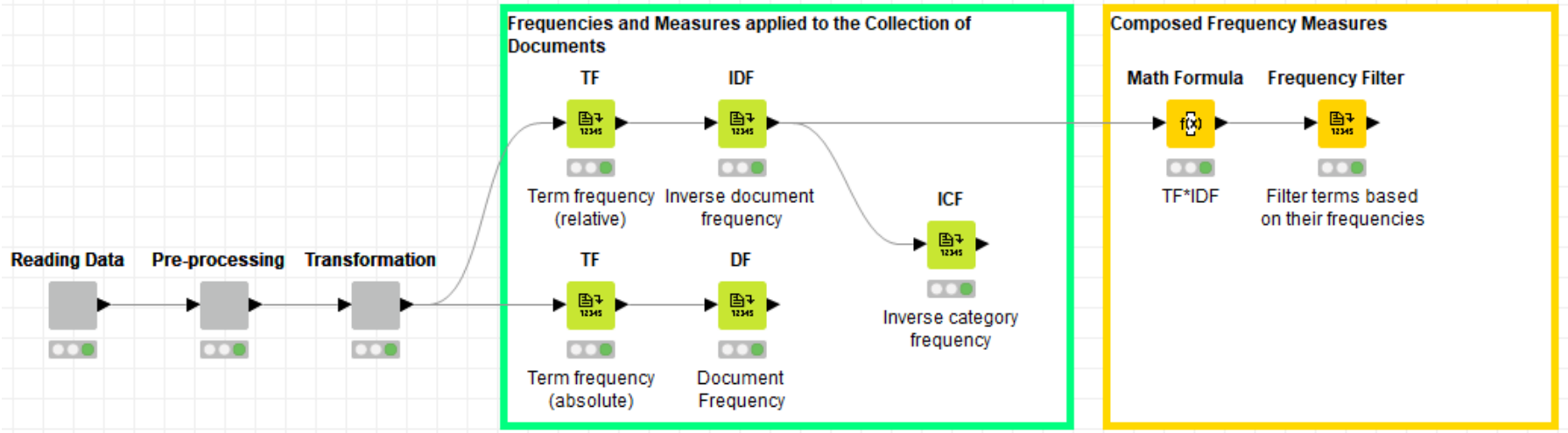
There is another node that calculates a similar frequency measure to the one produced by the IDF node: the **ICF** node. The ICF node calculates the **Inverse Category Frequency (ICF)**. ICF, introduced in [15], calculates the Term frequency relatively to a category / class of Documents rather than to a full set of Documents. The underlying concept is very similar however to the one behind the IDF measure.

More formally, the *Inverse Category Frequency* (ICF) is defined as follows:

$$icf_{t_i} = log\left(\frac{|C|}{cf(t_i)}\right)$$

where |C| denotes the number of categories in the collection of documents, and $cf(t_i)$ is the number of categories where the term t_i occurs. The node that calculates the ICF frequency measure is the ICF node. The ICF node has a minimal configuration window, consisting of solely the “Document Col” tab, where a dropdown menu allows to select the input Document type column to work on. The ICF node computes the Term frequencies with the aim to discriminate the Document category.

Figure 4.5. Example workflow stored in Chapter4/01_Frequencies, showing a number of nodes to calculate Term frequencies for a collection of Documents



4.3. Term Co-occurrences and N-Grams

So far we have described and quantified the importance of single words within a Document or a collection of Documents. However, often words make more sense when considered together rather than alone. Instead of considering the occurrences of single words, maybe we should move our attention to the co-occurrences of multiple words.

An evolution of frequency measures is brought by co-occurrence statistics. Co-occurrence statistics measures how often two or more terms occur together, therefore capturing the relationship between such words. Co-occurrence statistics can be computed at different levels: at Document level, at sentence level, and so on up to the highest level, i.e. the Document collection level.

There are many statistical measures for term co-occurrence. For instance, the ***word pair frequency*** counts how many times two words occur close together in a text (e.g. “Microsoft Word” vs. “Microsoft” and “Word”). For example, consider the following two sentences:

I built a KNIME workflow
The KNIME workflow computes the term frequencies

We can count term co-occurrences for neighboring words as “$w_{next}|w_{current}$”, which represents how many times the word w_{next} follows the word $w_{current}$. For the example above, co-occurrence statistics for the words “KNIME” and “frequencies” can be calculated across the whole data set and summarized as follows:

	a	built	computes	frequencies	I	term	the	KNIME	workflow
KNIME	1	0	0	0	0	0	1	0	2
frequencies	0	0	0	0	0	1	0	0	0

The table shows that the term “KNIME” occurs twice in pair with the term “workflow” and only once with “the” or “a”. The term “frequencies” is found in close proximity of the word “term” only once in this two sentence data set.

Note. Pair co-occurrence statistics does not take into account the term order in the sentence and refer to close neighbor words.

Since the term order is not considered, occurrence of *T1* followed by *T2* is equivalent to occurrence of *T2* followed by *T1*.

To compute term co-occurrences, the KNIME Text Processing extension provides the *Term co-occurrence counter* node. This node counts co-occurrences of term pairs in Documents at different levels - i.e. full Document, section, paragraph, sentence, and title – and returns the term pairs in alphabetical order with their co-occurrence number at the output port.

The co-occurrence level is ***hierarchical***. This means that Document includes section, which includes paragraph, which includes sentence, which includes neighbors, which includes title. This means that when the node is set to count co-occurrences at the paragraph level, it also outputs the co-occurrence numbers for its sub-levels: sentence, neighbor, and title.

Note. Higher levels of co-occurrence statistics result in larger output data tables and longer execution times.

Let’s take the workflow developed in the previous section, *Chapter4/01_Frequencies*, and let’s add a Term co-occurrence counter node, counting co-occurrences of terms in the same sentence. Having selected the level “Sentence” and being the co-occurrence levels hierarchical, we will also get the co-occurrences at the neighbor and title level in the output data table. Co-occurrences are calculated for each one of the input Document cells.

Let’s check now the most frequently associated words across all questions posted in the KNIME Forum data set. We must aggregate across all word pairs in Documents and sum up their co-occurrences with a GroupBy node. Then we sort the co-occurrence sum in descending order. The final output data table shows that the most frequently co-occurring words in questions about KNIME refers to “Eclipse” and “Java” and to all their different variations. Immediately after we find “knime” and “node”, followed by “debug” and “node”. The first pair is probably found in node development questions, while the second pair is probably found in questions about workflow building.

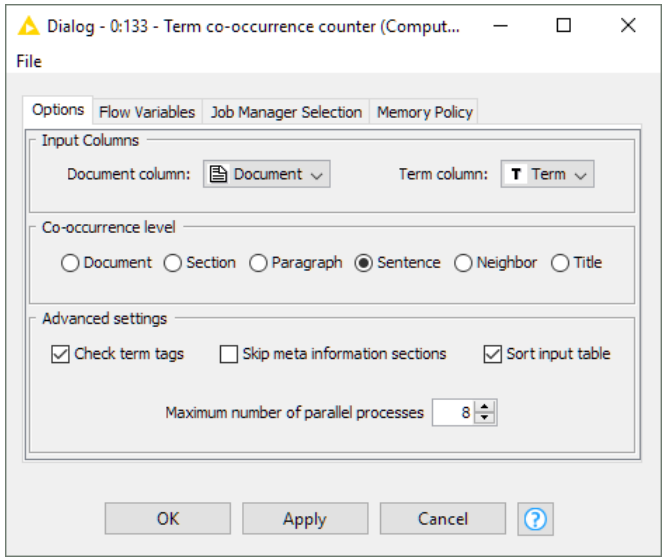
In general, term co-occurrence frequencies might be useful to associate concepts, like products, to judgements, like positive or negative, or to consequences, like bugs or malfunctioning.

Term co-occurrence counter

This node counts the number of co-occurrences of term pairs within selected Document parts, e.g. sentence, paragraph, section, and title. To make a term pair, words have to occur close to each other in the text. Configuration settings are the following:

- *Document column* dropdown menu selects the input Document column to use to compute the *term co-occurrence* number.
- *Term column* dropdown menu allows selecting the input Term column to compute the co-occurrences.
- Radio button menu *Co-occurrence level* defines the level for the calculation: Document, Section, Paragraph, Title, or just strictly neighboring words.

Figure 4.6. Configuration window of the *Term co-occurrence counter* node



- *Advanced settings* checkboxes allow to:
  - Consider the term tags as well, if option *Check term tags* is enabled. Terms with different tags are treated as different terms.
  - Ignore the meta information section, if *Skip meta information sections* checkbox is enabled;
  - Sort the input table by descending Term, if *Sort input table* option is checked. This pre-sorting might speed up calculation. However, it must be disabled if the input column comes already sorted.
- *Maximum number of parallel processes* text field controls the number of parallel processes to spawn.

So far, we have dealt with word pairs. Let us now move to N-Grams. N-Gram models, aka N-Grams, are sequences of *N* neighboring items, which can be phonemes, syllables, letters, words, or even DNA bases. An N-Gram with size N=1 is a *uni-gram*; with size N=2 it is a *bi-gram*; with size N=3 it is a *tri-gram*, and so on.

In order to be computed, N-Grams should respect two main assumptions:

Chain rule of probability [15]

The probability of a sequence of words *P(w_1, w_2, w_3, .., w_n)* - which can be synthetized as *P(w_1^n)* - can be calculated from the conditional probabilities of the previous words in the sentence, as follows:

$$P(w_1, w_2, w_3, .., w_n) = P(w_1^n) = P(w_1^0)P(w_2|w_1^1)P(w_3|w_1^2)P(w_4|w_1^3) \dots P(w_n|w_1^{n-1}) = \prod_{k=1}^{n} P(w_k|w_1^{k-1})$$

Where w_1^{k-1} indicates word w_1 in a word sequence of length *k*.

Markov assumption [16]

Since language might be really creative and different in every sentence or document, N-Grams take advantage of the Markov assumption, i.e. instead of computing the probability of a word given all the previous words, we can approximate the "word history" by considering just the few last words. This would also help with computational performance. The general model looks at *N-1* words. In formula, this can be expressed as follows:

$$P(w_1^n) \approx \prod_{k=1}^{n} P(w_k|w_{k-1})$$

Note. Unlike word pairs, for bi-grams, and N-Grams in general, the word order is important. So, T1 followed by T2 is a different bi-gram from T2 followed by T1.

The N-Gram Creator node detects N-Grams of length N in a Document and counts the number of its occurrences within the Document.

Again, we added an NGram creator node in the example workflow stored in Chapter4/01_Frequencies. The node was set to search the input Documents for tri-grams (N=3) words and to output the tri-gram, and its Document, corpus, and sentence frequency.

The corpus frequency considers the occurrences of the N-Grams in all input Documents, the Document frequency considers the occurrences of the N-Grams in each Document, and the sentence frequency considers the occurrences of the N-Grams overall at the sentence level.

The output data table of this NGram creator node, sorted by descending order of the sentence frequency, is shown in figure 4.8 and the final workflow in figure 4.9.

Note. It is possible to search N-Grams among the characters composing words. In this case, we would get the most common characters N-grams for the particular language under exam.

NGram creator

This node detects N-Grams in input Documents and counts their frequencies within each Document. The node configuration window allows to set:

NGram settings
- The number *N* of words in the N-Grams
- Whether the N-Grams have to be spotted on word or characters (*NGram type*)

Input / Output settings
- The Document type input column to use for the N-Gram search in *Document column* dropdown menu
- The type of output table through the *Output table* radio buttons.
    - *NGram frequencies* option produces the N-Gram followed by its frequency in the Document, in the sentence, in the corpus.
    - *NGram bag of words* option produces the input Document, the N-Gram and just its frequency within the Document.

Processes
- The N*umber of maximal parallel processes* to control execution speed
- The *Number of documents per process* for each parallel thread

Figure 4.7. Configuration window of the *NGram creator* node

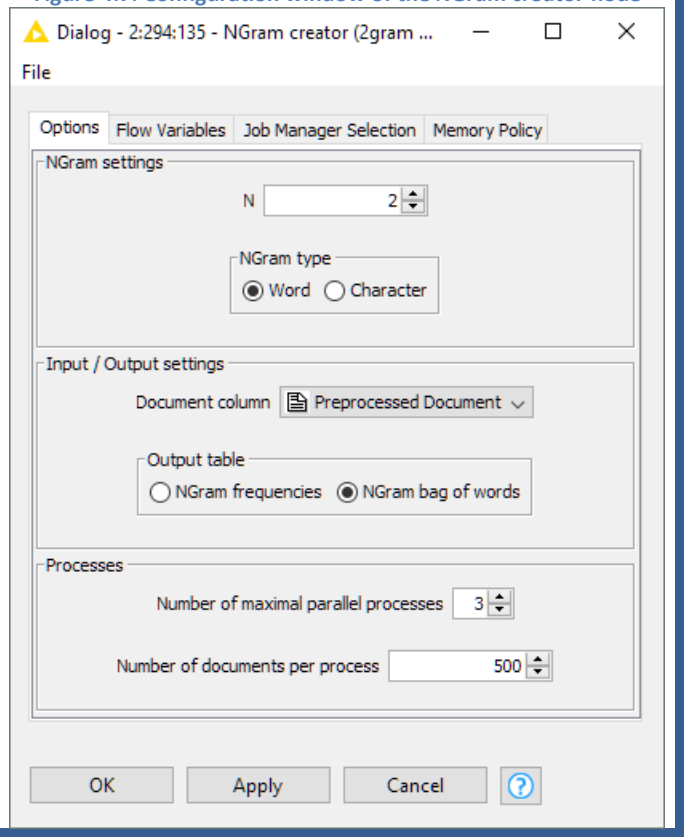
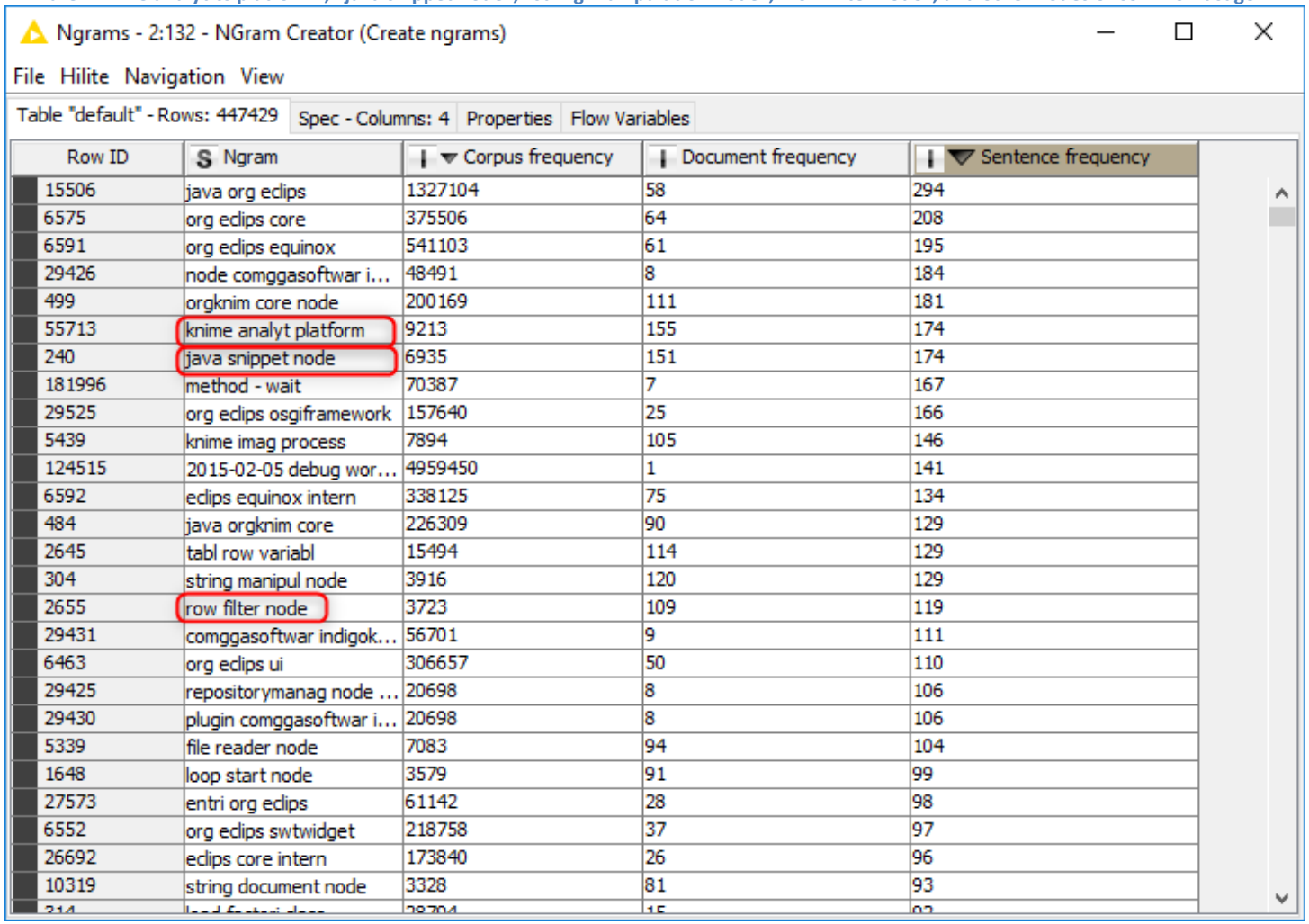
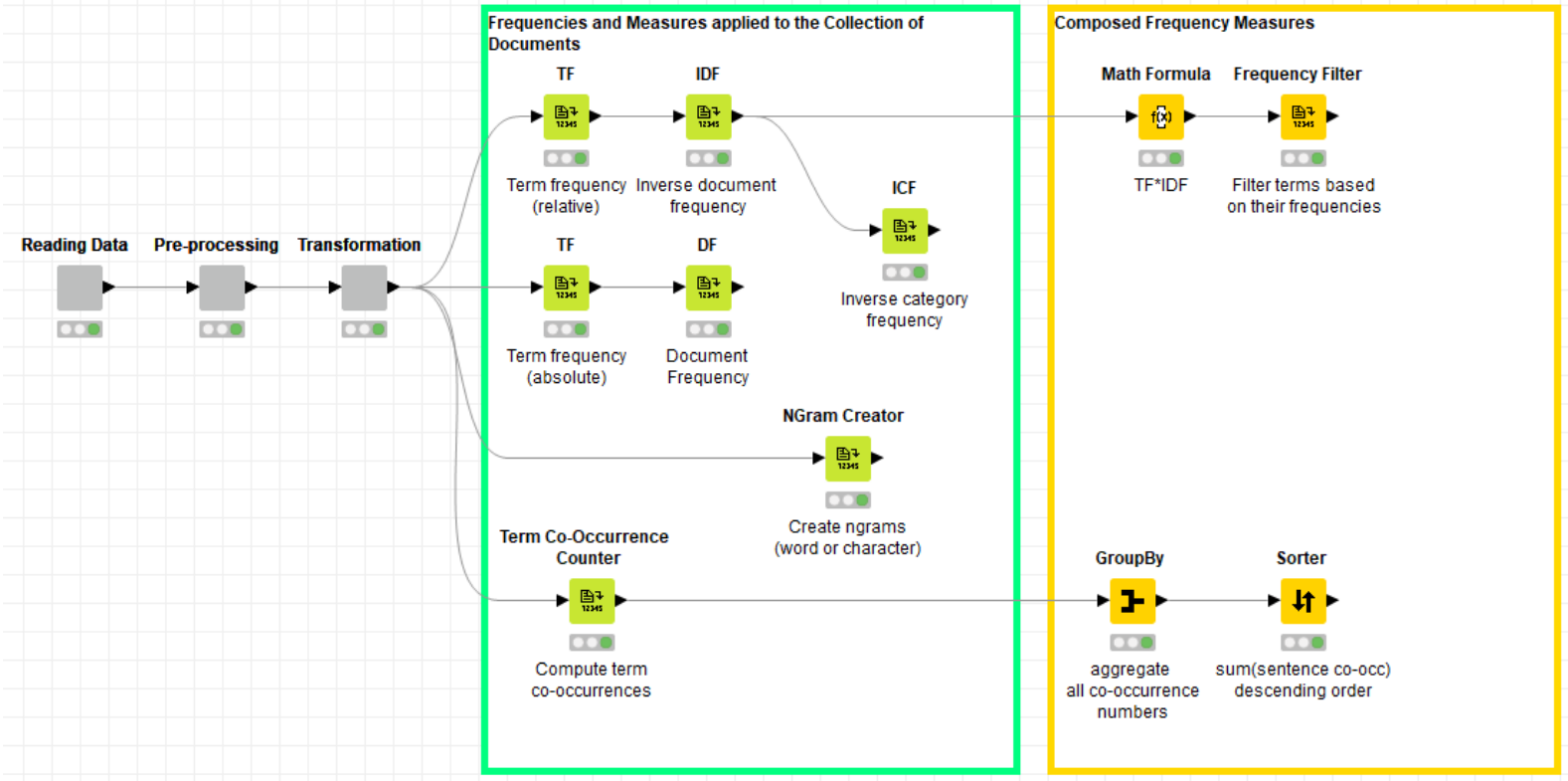


Figure 4.8. Output table of the NGram creator node in workflow Chapter4/01_Frequencies. The NGram creator node was set to look for word trigrams and to output frequencies. This data table is sorted by sentence frequency in descending order. Notice that the most frequently occurring word tri-grams at the sentence level over the whole Forum dataset are "knime analytics platform", "java snippet node", "string manipulation node", "row filter node", and other nodes of common usage.



Row ID	Ngram	Corpus frequency	Document frequency	Sentence frequency
15506	java org eclips	1327104	58	294
6575	org eclips core	375506	64	208
6591	org eclips equinox	541103	61	195
29426	node comggasoftwar i...	48491	8	184
499	orgknim core node	200169	111	181
55713	knime analyt platform	9213	155	174
240	java snippet node	6935	151	174
181996	method - wait	70387	7	167
29525	org eclips osgiframework	157640	25	166
5439	knime imag process	7894	105	146
124515	2015-02-05 debug wor...	4959450	1	141
6592	eclips equinox intern	338125	75	134
484	java orgknim core	226309	90	129
2645	tabl row variabl	15494	114	129
304	string manipul node	3916	120	129
2655	row filter node	3723	109	119
29431	comggasoftwar indigok...	56701	9	111
6463	org eclips ui	306657	50	110
29425	repositorymanag node ...	20698	8	106
29430	plugin comggasoftwar i...	20698	8	106
5339	file reader node	7083	94	104
1648	loop start node	3579	91	99
27573	entri org eclips	61142	28	98
6552	org eclips swtwidget	218758	37	97
26692	eclips core intern	173840	26	96
10319	string document node	3328	81	93

Figure 4.9. Workflow Chapter4/01_Frequencies expnaded with an NGram creator node and a Term co-occurrence counter node in its lower branches.



4.4. Document to Vector and Streaming Execution

We have finally reached the point where we are able to extract all single words (BoW) or N-Grams from a Document together with their frequencies. The final goal, however, has always been to deal with numbers rather than words; that is to transform this list of words into a numerical vector representation. This numerical representation is necessary for text classification, clustering, or similarity search.

The common set of words across all Documents in the collection is the space *vocabulary*. Each Document then can be represented by the vector of presence/absence (1/0) or frequency for each word in the vocabulary. The collection vocabulary then generates the *vector space model*.

The nodes responsible for transforming textual data into numerical vectors are the Document Vector node and its variations, all available in category *Text Processing / Transformation*.

Let's describe more formally the vectorization operation of a text Document. For a collection vocabulary of size t, Document d_j can be represented by the following word vector:

$$d_j = (w_{1,j}, w_{2,j}, w_{3,j}, .., w_{t,j})$$

Where d_j is a generic Document j, $w_{k,j}$ refers to the status of the k^{th} term in the vocabulary inside the j^{th} Document, and t is the dimension of the dataset vocabulary and therefore of the vector.

$w_{k,j}$ is equal to 0 if vocabulary word w_k is not present in j^{th} Document. Otherwise, $w_{k,j}$ is equal to 1 or to one of its frequency measures.

Let's take the following text collection as an example.

[Text 1] *I built a KNIME workflow*
[Text 2] *The KNIME workflow computes the term frequencies*

The collection vocabulary has size *t=9* and is: *{i, built, a, KNIME, workflow, the, computes, term, frequencies}.*

The corresponding vector representation for the two Document texts is:

	i	built	a	KNIME	workflow	the	computes	term	frequencies
Text 1	1	1	1	1	1	0	0	0	0
Text 2	0	0	0	1	1	1	1	1	1

The presence/absence binary encoding is called hot encoding. A variation of this hot encoding is to use frequency measure values instead of just 1s.

In this way, if a term occurs in the Document, its value in the vector is non-zero. The representation of a collection of Documents as vector space will be useful to measure the relationships and similarities among Documents. This means that Documents "close" to one another in the vector space are also "close" (similar) in content. For the purpose of creating a vector space from a collection of Documents, the KNIME Text Processing extension provides the *Document Vector* node.

The Document Vector node needs the vocabulary for the Document collection. Such vocabulary is the collection Bag of Words (BoW) and can be obtained with a Bag Of Words Creator node. If we want to use a frequency based variation of the Document hot encoding, term frequencies must also be available to the Document Vector node. This means that a Bag Of Words creator node and a set of nodes, such as TF, for frequency calculation must be applied previously to the Document Vector node.

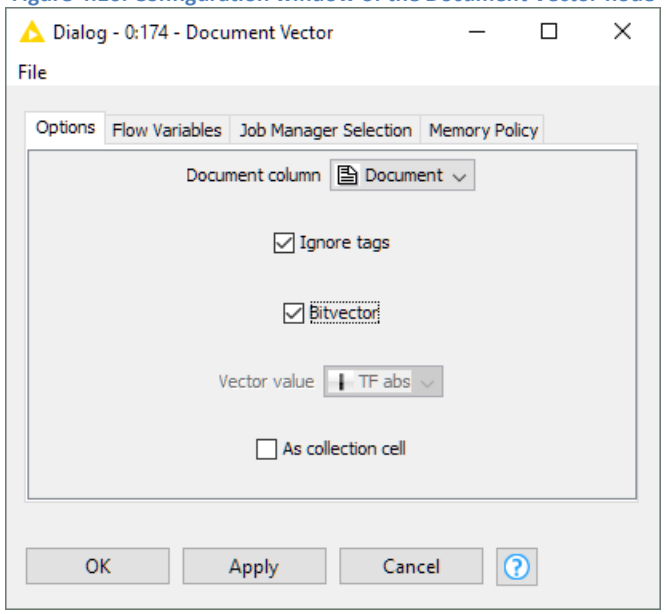
Document Vector

This node transforms all input Document texts into document vectors. The dimension of the output vectors is defined by the number of distinct terms in the vocabulary and the vocabulary can be generated for example by a previously applied *Bag of Words creator* node.

Configuration settings:

- *Document column* dropdown menu selects the Document type input column
- *Ignore Tags* checkbox controls whether to consider the same word as two terms if with different tags
- *Bitvector* checkbox generates a binary (if checked) or a frequency based (if unchecked) document vector. If a frequency based document vector has been chosen, the frequency measure to be used must be selected in *Vector value* dropdown menu.
- *As collection cell* checkbox controls the output format of the document vector: as a collection cell or as a data table.

Figure 4.10. Configuration window of the *Document Vector* node



Note. The Document Vector needs a Bag of Words and possibly frequency measures available at its input port.

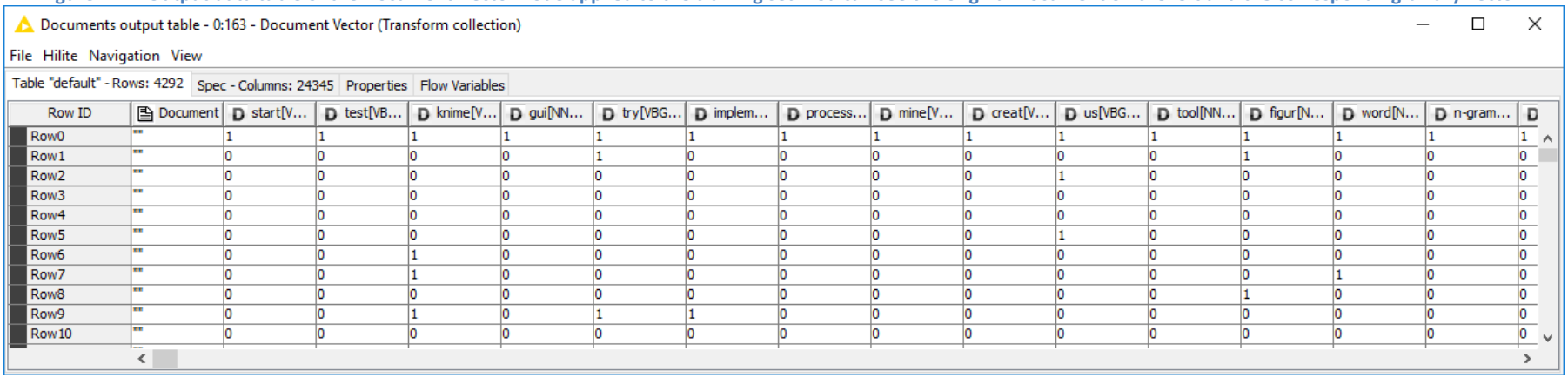
For this section we use the example workflow in *Chapter4/02_DocumentVector_Creation.* The dataset here is again the KNIME Forum Dataset. The first steps in the workflow are the usual ones: Reading the data and converting the Strings to Documents; term tagging for enrichment; cleaning up and stemming as pre-processing.

Thinking of the next classification operations, right before the generation of the dataset vocabulary via the Bag Of Words creator node, we split the Document set in training set and test set with a Partitioning node. We then create the vocabulary for the training set and the test set, with a Bag Of Words creator and a TF node on each workflow branch.

At this point of the workflow, we have a vocabulary and the corresponding term frequencies for each Document in the training set and in the test set. We can now apply the Document Vector node to the training set only. The goal is to define the vector transformation that gets binary document vectors

from input texts. The output data table with the document vectors generated by the Document Vector node applied to the training set is shown in figure 4.11.

Figure 4.11. Output data table of the Document Vector node applied to the training set. You can see the original Document on the left and the corresponding binary vector.



Row ID	Document	start[V...	test[VB...	knime[V...	gui[NN...	try[VBG...	implem...	process...	mine[V...	creat[V...	us[VBG...	tool[NN...	figur[N...	word[N...	n-gram...	D
Row0	""	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Row1	""	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0
Row2	""	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Row3	""	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Row4	""	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Row5	""	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Row6	""	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Row7	""	0	0	1	0	0	0	0	0	0	0	0	0	1	0	0
Row8	""	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
Row9	""	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0
Row10	""	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Document Vector model has been created on the training set. It must now be applied onto the test set. The node that applies the input Document Vector model onto the input data set is the Document Vector Apply node.

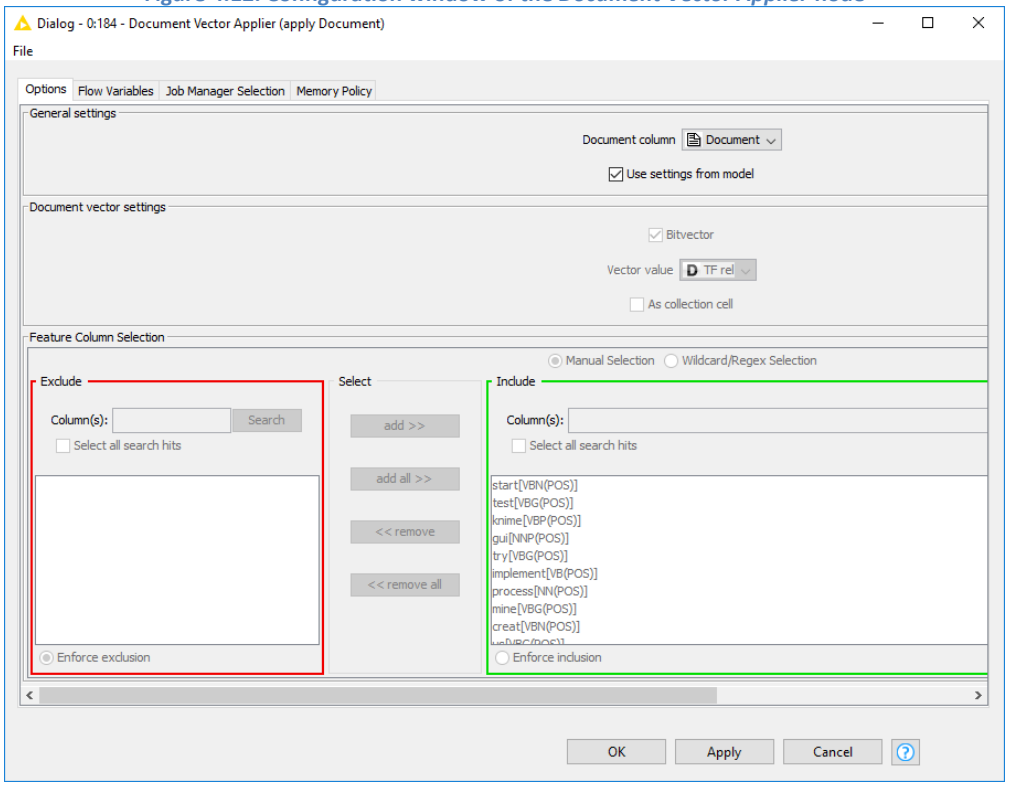
This node takes a Document Vector model at the first input port. This means: a vocabulary and a word vector schema for Document representation. The vocabulary has been built on a different data set, the training set. Thus, it is possible that some Documents in test set contain words not present in the training set vocabulary. Such words will not appear in the document vector representation of the test set.

Let's complete now the lower branch of the workflow in Chapter4/02_DocumentVector_Creation. This second branch originates from the second output port of the Partitioning node, i.e. from the test set. Here we apply the same Transformation wrapped metanode as for the training set to generate the BoW and frequencies for the test set as well. To such BoW and frequencies we apply the Document Vector Apply node, so obtaining a document vector representation according to the schema in figure 4.11.

Document Vector Applier

This node creates a document vector representation for the Documents in the input data table based on the document vector model at its first input port.

Figure 4.12. Configuration window of the *Document Vector Applier* node



Configuration settings:

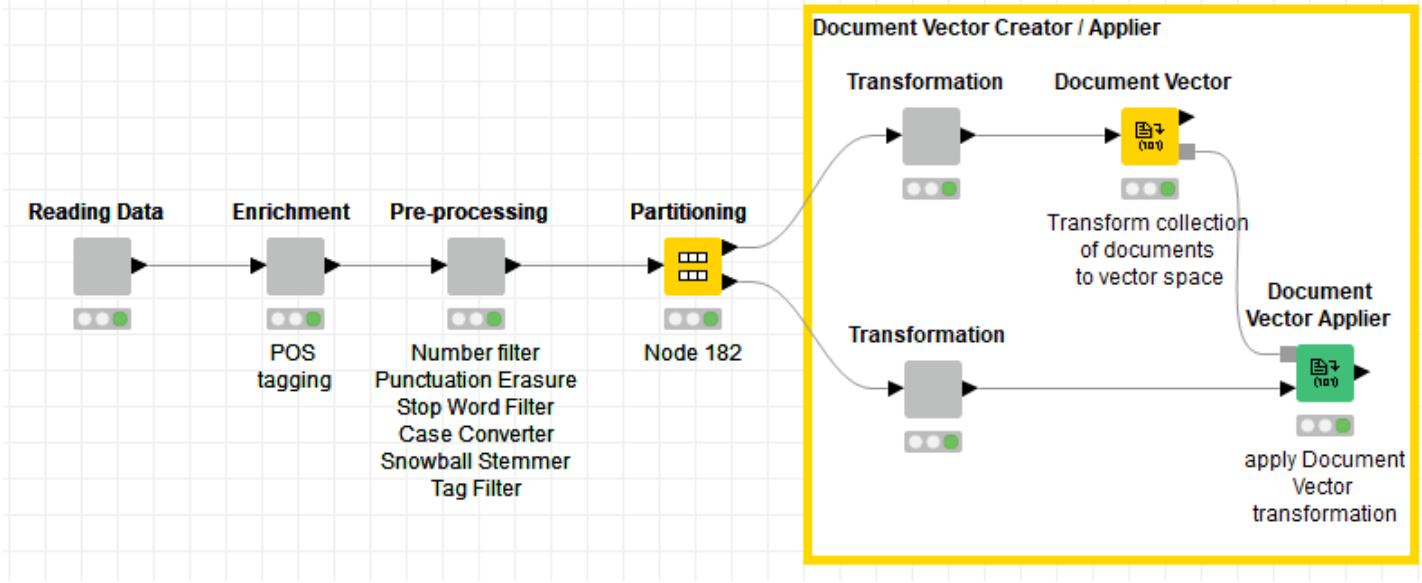
- *Document column* dropdown menu selects the input Document type column.

- *Use Settings from model* uses the vocabulary and the data table schema from the document vector input model. If unchecked, you can customize the vocabulary words and the word columns in the final document vector representation through the Include/Exclude framework in the lower part of the window.

- *Bitvector* checkbox creates a binary document vector based on presence/absence of word in Document. If unchecked, a frequency measure can be selected from the *Vector value* dropdown menu. This whole section is disabled if same settings as in input model have been selected.

- *As collection cell* checkbox defines the format of the document vectors in the output data table.

Figure 4.13. Workflow Chapter4/02_DocumentVector_Creation, after importing and cleaning up the posts from the KNIME Forum dataset, splits them into a training set and a test set with the Partitioning node, creates the BoW and corresponding frequencies for each Document in both sets, and then transforms the input Documents into document vectors by training a model on the training set with the Document Vector node and by applying this same model to the test set with the Document Vector Apply node.



Text Processing nodes involve computationally intensive operations. If you are planning to use them on large Document collections, you might experience long execution times. This is in the default node-after-node KNIME execution. However, KNIME Analytics Platform offers an alternative execution mode, via streaming.

When a node is executed in streaming mode, batches of data are passed one after the other to the node for execution. When the node has finished execution on one data batch, the batch is passed to the second node, while the next data batch comes in for execution. In this way, all nodes are active on some data batch at all times and execution becomes faster. On the other hand, the advantage of default execution mode is that you can see the node results at each execution step, which simplifies debugging considerably.

More information related to the *KNIME Streaming Executor* are available in the blogpost available at the following URL: https://www.knime.com/blog/streaming-data-in-knime .

Let's introduce here the streaming execution mode.

Streaming Mode Execution

In order for a node to execute in streaming mode, it must be encapsulated in a wrapped metanode. Indeed, streaming execution is only available for wrapped metanodes.

To create a wrapped metanode, select the group of nodes of interest, then right-click and select “Encapsulate into Wrapped Metanode” and give it a meaningful name.

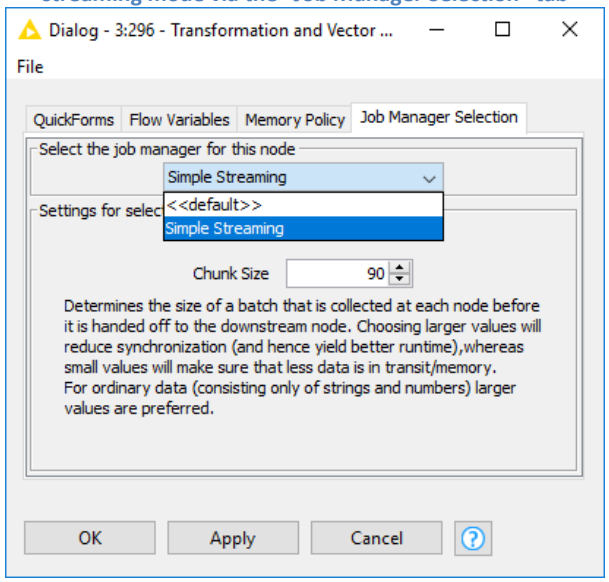
Double-click on the wrapped metanode; its configuration window opens. If the metanode does not contain Quickform nodes, such configuration window is empty.

Select the *“Job Manager Selection”* tab on the right, and select the *“Simple Streaming”* executor from the dropdown menu. The alternative to “Simple Streaming” executor is the *“<default>”* node-after-node execution mode.

Chunk Size finally defines the size of the data batch to pass from node to node.

Use Ctrl-Click to open the metanode content.

Figure 4.14. Setting a wrapped metanode to execute in streaming mode via the “Job Manager Selection” tab



For nodes to be executed in streaming mode, they must be *streamable*. Streamable means that they can operate on small batches of data independently. Not all transformations are streamable, since some of them requires the whole data set to operate, like for example a data table transposition.

In the Text Processing extension, most nodes are now streamable. The only node which still cannot be stremed is the Document Vector node. By the way this node is also quite computationally expensive when operating on large data sets and therefore on large vocabularies.

A faster version of the Document Vector node is the Document Vector Hashing node. This node implements the same task as the Document Vector node, i.e. transforms Documents into numerical vectors. In addition it uses a hash function to index the resulting columns. This makes the node execution much faster. Hashing functions work well for large data sets. For small data sets there is the chance of index collision.

Unlike the Document Vector node, the Document Vector Hashing node uses a fixed length for the document vectors. This does not allow vocabularies of variable length, but it makes the node streamable.

Feature hashing allows significant storage compression for the vectors created. In particular, feature hashing is extremely useful when large numbers of parameters with redundancies need to be stored within bounded memory capacity. The final result is an increased speed and reduced memory usage.

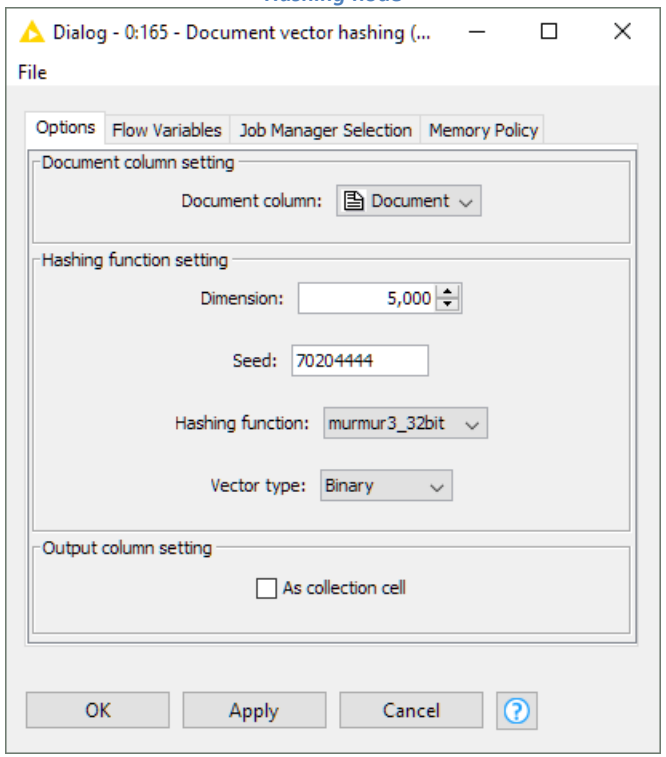
Document Vector Hashing

This node implements the same task as the Document Vector node, i.e. transforms Documents into numerical vectors. In addition, it uses a hash function to speed up execution.

Configuration settings:

- *Document column* dropdown menu selects the Document type input column to transform.
- *Dimension* integer box sets the dimension of the output document vector.
- The text field *Seed*[1] sets the seed for the hashing function. The same seed need to be used to reproduce exactly the same transformation.
- *Hashing Function* dropdown menu selects one of the many possible hashing functions,
- *Vector type* dropdown menu defines the output vectors as binary based on word presence/absence or as frequency based.
- *As collection cell* checkbox defines the format of the output vectors, as cell collections or as a data table

Figure 4.15. Configuration window of the *Document Vector Hashing* node



[1] A pseudorandom number generator (PRNG) is an algorithm for generating a sequence of numbers that approximates the properties of random numbers.

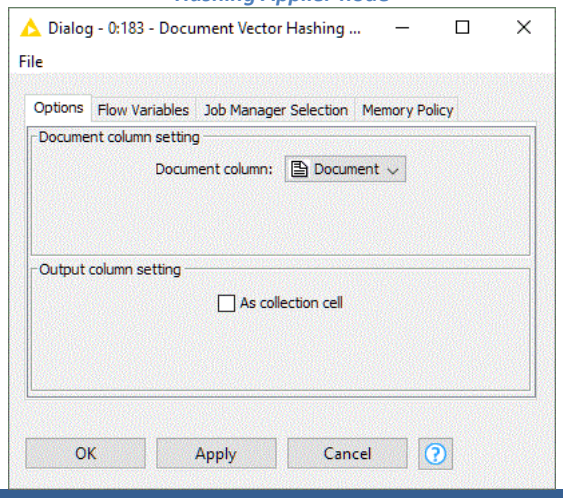
Document Vector Hashing Applier

This node transforms input Documents into document vector by using the hashed document vector model provided at its input port. This node uses the hash function of the model to speed up execution.

Configuration settings:

- *Document column* dropdown menu selects the input Document type column to transform.
- *As collection cell* checkbox defines the format of the output vectors, as data cells or as a cell collection.

Figure 4.16. Configuration window of the *Document Vector Hashing Applier* node



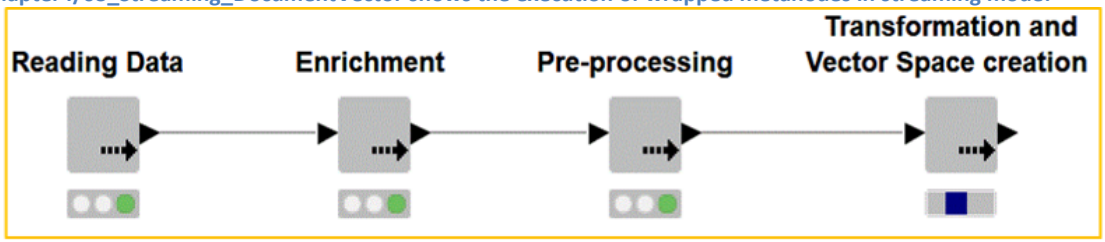
An example for the application of the Document Vector Hashing node and for streaming execution in general is available in workflow *Chapter4/03_Streaming_DocumentVector*. Figure 4.17 shows workflow *Chapter4/03_Streaming_DocumentVector* at the top and its wrapped metanodes below during execution. The last wrapped metanode, named “Transformation and Vector Space creation”, contains a Document Vector Hashing node.

Note. The dotted arrows in the lower right corner of each metanode indicate that the node has been set to execute in streaming mode.

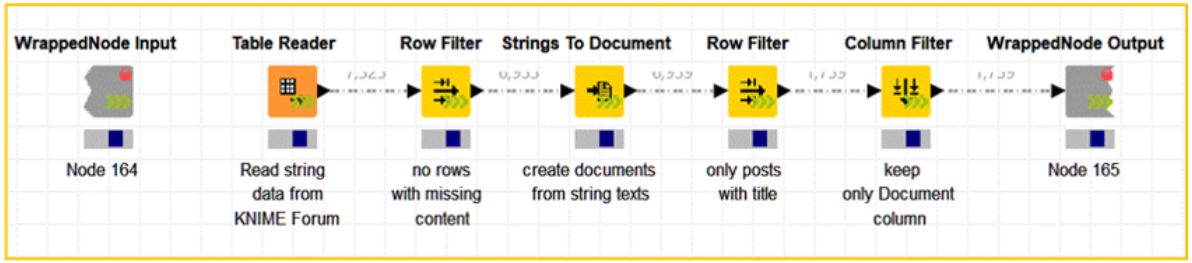
Note. The dotted connections in between nodes during execution show the streaming flow and the current number of processed data rows.

Figure 4.17. Workflow Chapter4/03_Streaming_DocumentVector shows the execution of wrapped metanodes in streaming mode.

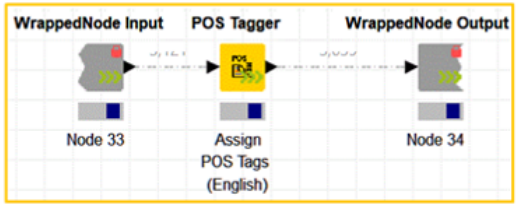
Workflow
Streaming_DocumentVector



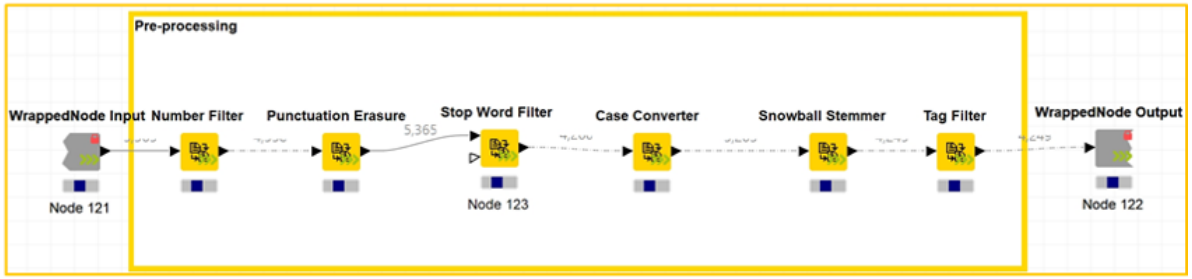
Wrapped Metanode
Reading Data



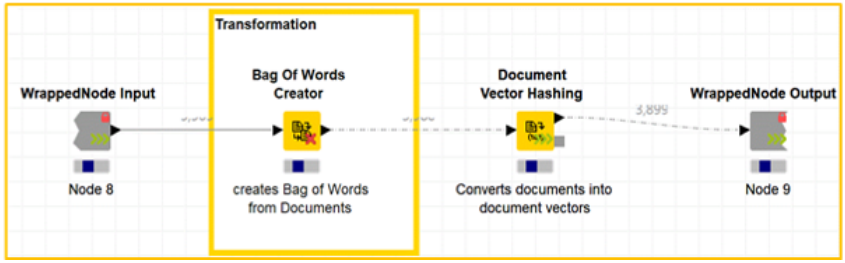
Wrapped Metanode
Enrichment



Wrapped Metanode
Pre-processing



Wrapped Metanode
Transformation and Vector Space creation



4.5. Keyword Extraction

An alternative way to speed up the execution of the Document Vector node, is of course to reduce the size of the model vocabulary. Indeed, even after cleaning the original text and removing all stop words and all non-informative yet grammatically necessary words, we will likely end up with a long list of words from the Bag Of Words creator node. Obviously, the longer the input texts and the collection, the longer the list of words in the vocabulary.

How can we reduce the collection vocabulary? We cannot of course just cut the list in an arbitrary point, since we might miss the most descriptive words for that text. In this section, we will deal exactly with this problem, i.e. how to reduce the number of words needed to describe a text, yet maintaining the most possible information about its content. Here is where the concept of keywords comes in.

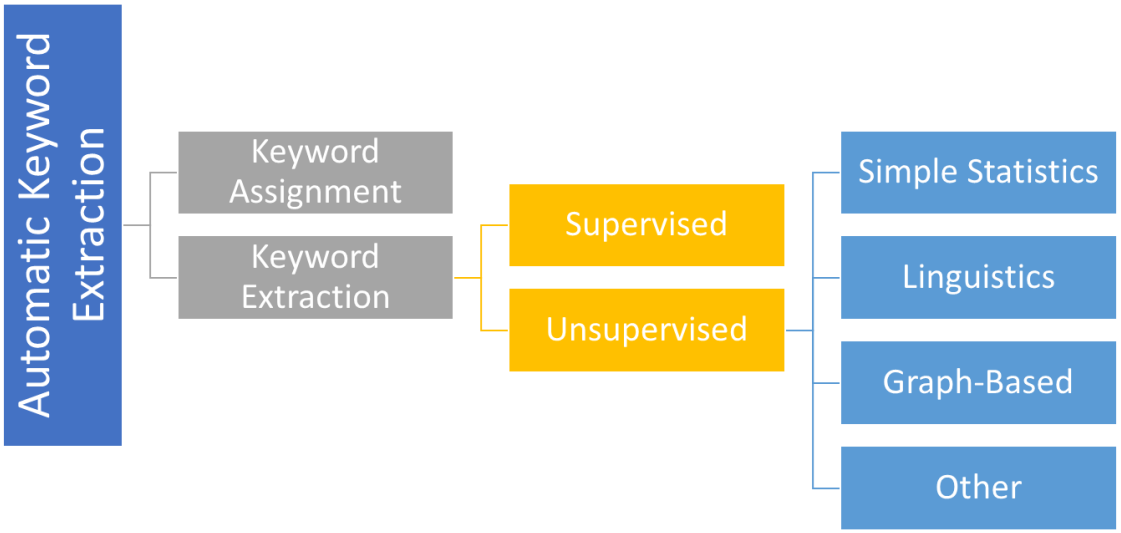
The terminology actually is wide-ranging and includes terms such as keyphrases, keysegment, keyterms, besides keywords. Those terms have slightly different meanings in the literature but they all define the phrases, segments, terms, and words summarizing the most relevant information in the Document or collection of Documents.

A keyword is a word that best summarizes the content or identifies the topic of its Document. A set of keywords is the set of the most descriptive words for that Document. Thus, appropriate keywords may provide a highly concise summary of a text and help us easily organize Documents. For this reason, automatic keyword extraction methods are widely used in Information Retrieval systems, NLP applications, and Text Mining.

The number of existing techniques for automatic keyword extraction is pretty vast, including supervised and unsupervised methods, which in tunr can be statistics based, linguistics based, graph based or heuristic based. However, since many of the proposed techniques encounter scalability and sparsity problems, new solutions are constantly being proposed.

In general, the taxonomy of the methods currently available to perform automatic keyword extraction can be represented as follows [17]:

Figure 4.18. Taxonomy representation of the different ways to extract keywords from a document or a collection of documents automatically



Following the taxonomy defined in [17], keyword assignment methods can be roughly divided into two categories: (1) keyword assignment and (2) keyword extraction. Both methods revolve around the same problem – selecting the best keywords.

In ***keyword assignment***, keywords are chosen from a controlled vocabulary of terms or predefined taxonomy. Then Documents are categorized into classes according to their word content. Keywords do not need to be explicitly mentioned in the text. Related words must suffice.

In ***keyword extraction*** keywords are detected in the Document and form the collection vocabulary.

The *supervised* approaches use a classifier to detect candidate words. These supervised methods require a labelled collection of Documents, which is not easy to have. When labels are not available for the Documents, the *unsupervised* methods come in handy.

In this chapter, we focus on unsupervised approaches for keyword extraction. The stages of an algorithm that performs keyword extraction can be summarized in three main phases:

1. *Candidate Keywords*: extract all possible words that can potentially be keywords.

2. *Feature Extraction*: for each candidate calculate features to measure its being a keyword.
3. *Evaluation*: verify the pertinence of the features proposed.

KNIME Text Processing extension provides a couple of nodes for automatic keyword extraction, i.e. ***Chi-Square Keyword Extractor*** and ***Keygraph Keyword Extractor***.

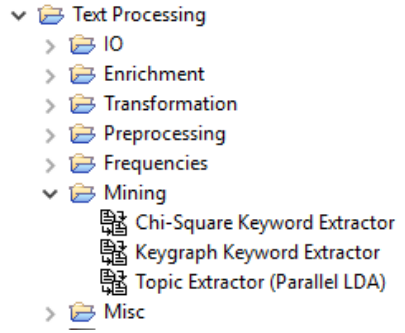


Figure 4.19. Keyword Extraction methods available in the KNIME Text Processing extension

The Chi-Square Keyword Extractor uses a chi-square measure to score the relevance of a word to a given text.

The Keygraph Keyword Extractor uses a graph representation of the Document collection to find the Document keywords.

The algorithm ***Chi-square Keyword Extractor*** detects the relevant keywords using a statistical measure of co-occurrence of words in a single Document. The algorithm behind this node has been developed by Matsuo and Ishizuka [18] and it follows these steps:

1. Extraction of most frequent terms
2. Counting co-occurrences of a term and frequent terms
3. Statistical significance evaluation of biases based on the chi-square test against the null hypothesis that "occurrence of frequent terms *G* is independent from occurrence of term *w*,", which we expect to reject.

Generally, terms with large χ^2 score (index of biases) are relatively important in the Document; terms with small χ^2 are relatively trivial. In other words, if a term appears selectively with a particular subset of frequent terms, the term is likely to have an important meaning.

The function used to calculate the χ^2 value takes into account two co-occurrence biases, due to sentence length and adjunct words.

Since the document consists of various sentences which are of different lengths, we know that a term that appears in a short sentence is less likely to co-occur with other terms. It is true even the opposite, i.e. a term that appears in long sentences is more likely to co-occur with other terms. For this reason, the function used to calculate the χ^2 value takes into account the expected frequency of term co-occurrence due to the sentence length.

The function for the χ^2 value also considers the case where terms are adjuncts of a generic term *g* and for this reason they co-occur more often than others. For instance, we might have higher χ^2 values for terms like *manipulation*, and *pre-processing* because they are adjuncts of the term *data*.

Now, let's correct another problem. Suppose that you have two terms t_1 and t_2 frequently appearing together. Co-occurrence of terms q and t_1 might imply the co-occurrence of q and t_2. This implies that term q has also a high χ^2 value, which could lead q to become a keyword. In order to prevent this and to avoid handling too sparse matrices, we cluster the terms using a similarity-based approach and a pairwise approach [17].

Let's see now the algorithm steps in the Chi-Square Keyword Extractor node:

1- *Selection of frequent terms*. By default, the top 30% of Document terms is selected.

2- *Clustering frequent terms*. Terms are clustered using two measures: distribution similarity and mutual information. For distribution similarity, the Chi-Square Keyword Extractor node computes the L1 norm - normalized in [0,1] - instead of the Jensen-Shannon divergence suggested in [17]. This means that all terms whose normalized L1 norm score is greater than or equal to a set threshold are considered similar. As mutual information the pointwise mutual information is used, to cluster terms that co-occur frequently. The terms whose pointwise mutual information is greater than or equal to the set threshold are considered similar, and thus clustered together.

3- *Calculation of expected probability*. Here the algorithm counts the number of terms co-occurring and computes the expected probability.

4- *Computation of the χ^2 value*. The χ^2 value is computed following the formula:

$$\chi'^2(w) = \chi^2(w) - \max_{g \in G} \left\{ \frac{(freq(w,g) - n_w p_g)^2}{n_w p_g} \right\}$$

Where:

$\chi^2(w) = \frac{(freq(w,g) - n_w p_g)^2}{n_w p_g}$, $n_w p_g$ is the expected frequency of the co-occurrence, and $(freq(w,g) - n_w p_g)$ is the difference between expected and observed frequencies. This computation function of the χ^2 value is more robust against bias values.

Chi-Square Keyword Extractor

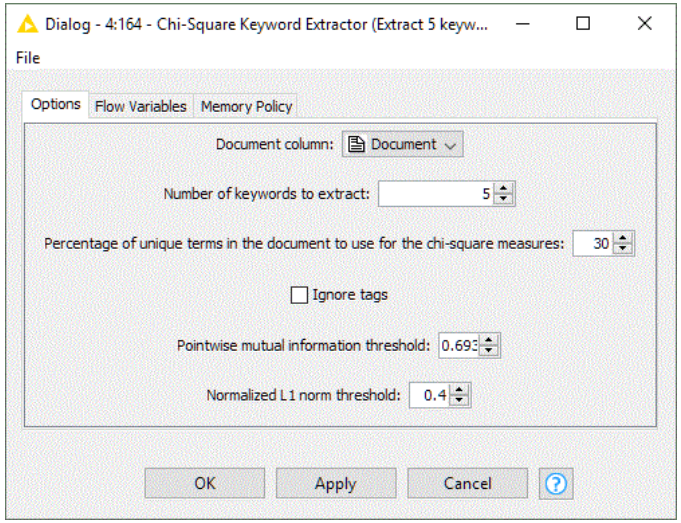
This node automatically extracts keywords using the algorithm proposed in [17].

Configuration settings:

- *Document column* dropdown menu selects the input Document type column to work on.
- *Number of keywords to extract* sets the final number of keywords for each Document.
- *Percentage of unique terms in the document to use for the chi/square measures* field text sets the percentage of Document's top frequent terms to use for the clustering phase. The default value is 30%.
- *Ignore tags* checkbox treats all terms equally, independent of their tags, if checked.
- *Pointwise information threshold* text box sets the threshold for the pointwise mutual information during the clustering phase.
- *Normalized L1 norm threshold* text box sets the for the distribution similarity measure.

The output data table includes the keywords and their Chi-Square score.

Figure 4.20. Configuration window of the *Chi-Square Keyword Extractor* node



The other keyword extraction node available within the kNIME Text Processing extension is the ***Keygraph Keyword Extractor*** node.

The Keygraph Keyword Extractor node relies on analyzing the graph of term co-occurrences. The algorithm is based on three major phases as illustrated by Oshawa, Benson, and Yachida in [19]:

1. Create clusters based on term co-occurrences inside the Document. In doing so, we build a co-occurrence graph.
2. Extract information about connected terms (edges in the graph).
3. Extract keywords.

Terms are graph nodes. Term co-occurrences are graph edges.

In **phase 1**, a graph *G* for a document *D* is created using Document terms as nodes. Connection between two nodes (edges) is given by the number of co-occurrences of these two terms. Only high-frequency terms are taken into account to build the graph *G*. Thus, such terms/nodes are clustered as follows:

- Terms are sorted by their frequency and a fraction is extracted. By default the top 30% terms are used to create the nodes.
- Edges are created if the association between the two terms is strong.
  The association measure used in [19] is:

$$assoc(w_i, w_j) = \sum_{s \in D} min(|w_i|_s, |w_j|_s)$$

  Where: $|w|_s$ represents the occurrence count of generic term *w* in sentence *s.*

  Once the association measure is calculated, all pairs of high frequency terms are sorted by association. This allows to appropriately segment the document and so to create the clusters.

We won't go into the details with the steps showed by Benson, and Yachida [19], concerning **phase 2**. We will only show the formula that measures the key factor of a word:

$$key(w) = 1 - \prod_{g \subset G} \left(1 - \frac{based(w, g)}{neighbors(g)}\right)$$

$based(w, g)$ counts the word *w*'s occurrences in the same sentence, as term in cluster g.

$neighbors(g)$ counts the number of occurrences of terms in cluster *g (neighbor terms)*.

$\frac{based(w,g)}{neighbors(g)}$ corresponds to the rate of occurrence of term *w* in the term neighborhood in cluster *g*.

$key(w)$ denotes the conditional probability that word *w* is used when keeping all word clusters in mind.

We can now extract N *high key terms,* i.e. the N terms with highest *key(w)*. The default threshold for high key term extraction, suggested by Oshawa, Benson, and Yachida in [19], was N=12. The *high key terms* extracted here are used to create new nodes in the graph, if those are not part of the graph, yet.

Finally, in **phase 3**, the final key score is calculated as the sum of *key(w)* value across edges and sentences.

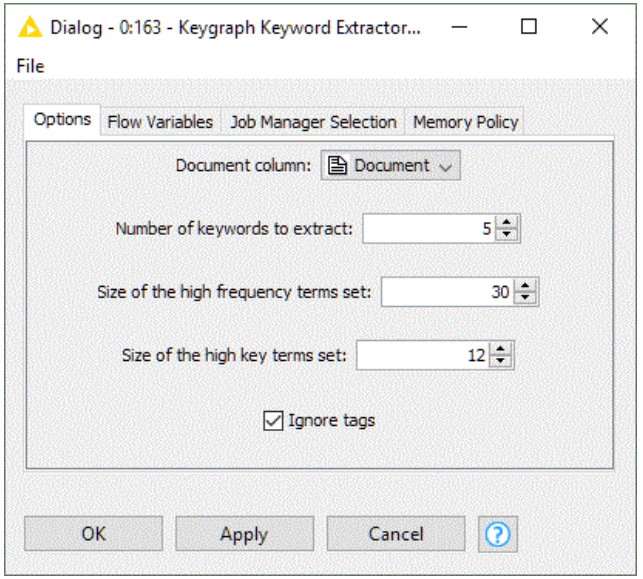
Keygraph Keyword Extractor

This node automatically extracts keywords by using the graph-based approach described in [19].

Configuration settings.

- *Document column* dropdown menu identifies the Document type input column to use for keyword extraction.
- *Number of keyword to extract* sets the number of keywords for each Document.
- *Size of the high frequency terms set* sets the number of high frequency terms to build the clusters. By default, the top frequent 30% terms are used.
- *Size of the high key terms set* defines the size of high key terms set. By default the high key terms set contains the 12 top terms.
- *Ignore tags* checkbox, if checked, treats all words equally, independent of possible tags.

Figure 4.21. Configuration window of the *Keygraph Keyword Extractor* node



The output data table includes the keywords and their total key score.
If the console's output level in the Preferences page is set to DEBUG, the console will display the contents of the clusters after the pruning phase, during node execution.

The example workflow, that we have prepared to illustrate keyword extraction, is named *04_Keyword_Extraction* and is located in the Chapter4 folder. In this workflow, first we read the KNIME Forum posts and we convert them into Documents. In addition a Meta Info Inserter node adds the Forum Topic to the Documents. Those steps are encapsulated in the wrapped metanode "Reading Data". Next, we enrich the documents by applying the *POS Tagger* node. The *Preprocessing* wrapped metanode implements some of the classical pre-processing steps. The Meta Info Extractor node here extracts the Forum topic that we had added earlier on and the next Row Filter node keeps only the posts in the Big Data extension forum.

Now we create two branches. The first branch applies a *Chi-Square Keyword Extractor* node, while the second branch applies a *Keygraph Keyword Extractor* node. Both nodes are set to extract 5 keywords per each Document. Finally, a *Sorter* node sorts the Chi-square values in one branch and the key scores in the other branch in descending order and a *Row Filter* node extracts only the Documents containing "**hdinsight compat**".

The output keywords and their scores are visible respectively in figure 4.22 and 4.23.

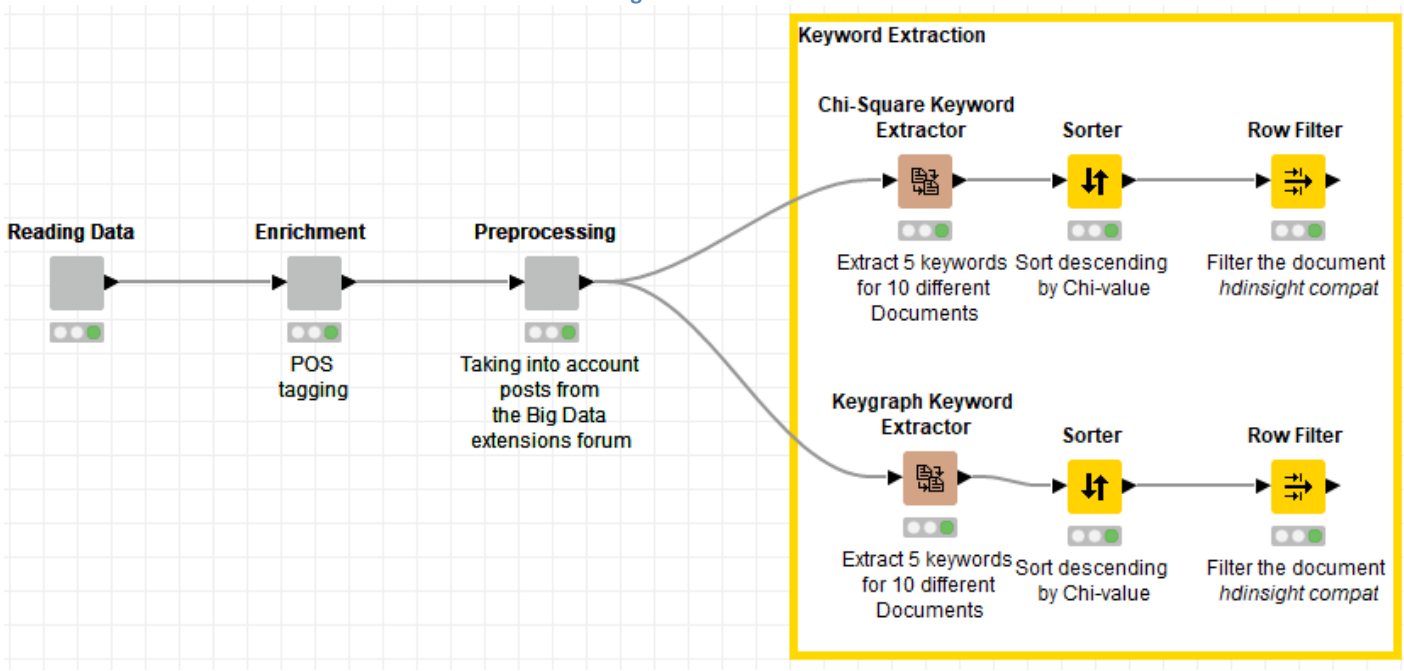
Figure 4.22. 5 Chi-square keywords and their Chi scores for Documents from the Big Data Extension Forum containing "*hdinsight compat*".

Row ID	Keyword	Chi value	Document
0	compat[JJ(POS)]	2.981	"hdinsight compat"
1	hdinsight[NNP(POS)]	2.34	"hdinsight compat"
2	particular[JJ(POS)]	1.011	"hdinsight compat"
3	platform[NN(POS)]	1.011	"hdinsight compat"
4	azur[NNP(POS)]	1.011	"hdinsight compat"

Figure 4.23. 5 Keygraph keywords and their Key scores for Documents from the Big Data Extension Forum containing "*hdinsight compat*".

Row ID	Keyword	Score	Document
0	hdinsight[NNP(POS)]	16	"hdinsight compat"
1	purchas[NN(POS)]	14	"hdinsight compat"
2	data[NNS(POS)]	9	"hdinsight compat"
3	extens[NNS(POS)]	9	"hdinsight compat"
4	spark[NN(POS)]	9	"hdinsight compat"

Figure 4.24. Workflow Chapter4/04_Keyword_Extraction implements keyword extraction using the Chi-Square algorithm and the Keygraph algorithm. Result keywords are shown in figure 4.22 and 4.23.



4.6. Exercises

Exercise 1

Read the Large Movie Review Dataset[2] (sampled) available at the following path *Thedata/MoviereviewDataset_sampled.table*. The dataset contains sentiment labeled reviews as positive or negative, as well as unlabelled reviews.

Use the *Strings to Document* node to transforms the Strings into Documents.

Pre-process the Documents by filtering the numbers, remove punctuation signs, remove the stop words, convert to lower case, reduce the words to their stem and keep only nouns and verbs.

Create a bag of words for the remaining terms and keep only the terms appearing in at least 5 Documents.

Compute the terms' absolute TF and relative TF frequencies.

Bin the relative frequencies with an Auto-Binner node in sample quantiles (0.0, 0.25, 0.5, 0.75, 1.0). Then, with a *Row Filter* node, keep only Bin 4.

Of the remaining terms keep only those with TF relative > 0.2.

Finally, get the list of the most frequent terms in Documents with a *GroupBy* node.

Solution

[2] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. (2011). Learning Word Vectors for Sentiment Analysis. The 49th Annual Meeting of the Association for Computational Linguistics (ACL 2011)

Figure 4.25. Solution workflow for exercise 1. Terms and Frequencies for the Movie Review Dataset.

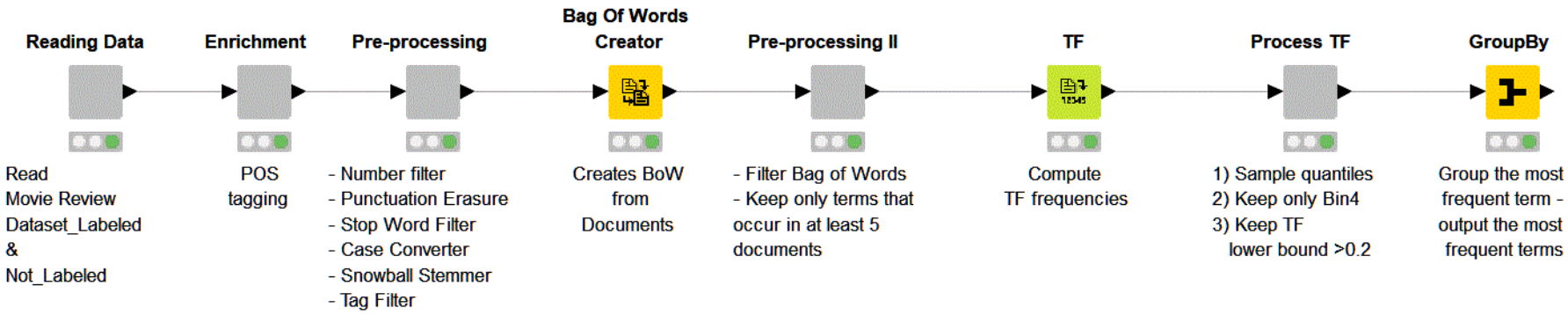


Figure 4.26. Sub-workflow in wrapped metanode „Pre-processing II“

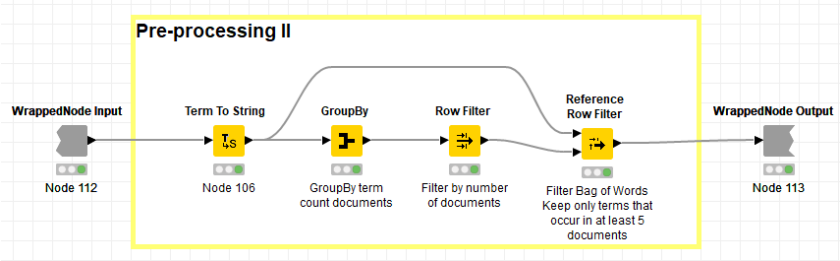


Figure 4.27. Sub-workflow in wrapped metanode „Process TF“

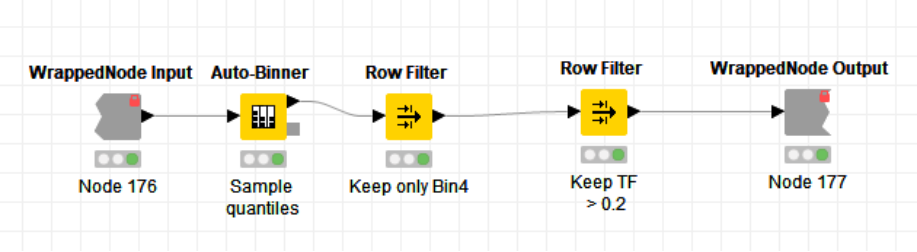


Figure 4.28. Final output data table showing the most frequent terms as grouped by the GroupBy node

Row ID	S Term a...
Row0	action
Row1	actor
Row2	br
Row3	camp
Row4	charact
Row5	danc
Row6	eddi
Row7	film
Row8	fun
Row9	game
Row10	godzilla
Row11	joke
Row12	look
Row13	love
Row14	make
Row15	movi
Row16	murphi
Row17	pictur
Row18	plot
Row19	rent
Row20	stori
Row21	tragedi
Row22	version

Exercise 2

Read the Large Movie Review Dataset (sampled) available at the following path Thedata/MoviereviewDataset_sampled.table. The dataset contains labeled reviews as positive or negative, as well as unlabelled reviews.

Use the *Strings to Document* node to transforms the strings into documents.

With a Document Data Extractor node extract the category for each Document and then keep only Documents with category = ”POS”, i.e. positive sentiment. Of those Documents keep only the first 3.

Pre-process them by filtering numbers, erase punctuations, filter stop words, convert to lower case, and stem terms.

Finally, use the *Chi-Square Keyword Extractor* node to extract 5 keywords for each document. Keep only keywords with Chi Value > 10.

What are the most valuable keywords (chi value > 10)?

Solution

Figure 4.29. Solution workflow for exercise 2. Keyword Extraction on Movie Review Dataset.

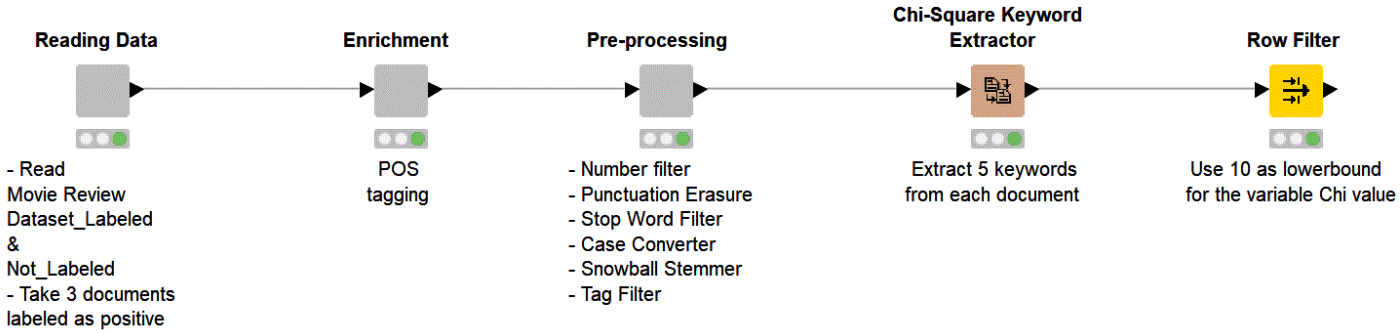
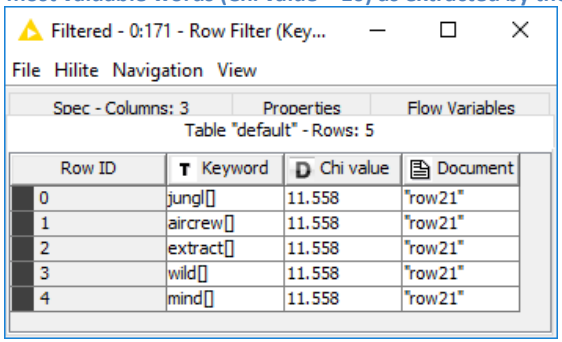


Figure 4.30. Top 5 most valuable words (Chi value > 10) as extracted by the Row Filter node.



Row ID	Keyword	Chi value	Document
0	jungl[]	11.558	"row21"
1	aircrew[]	11.558	"row21"
2	extract[]	11.558	"row21"
3	wild[]	11.558	"row21"
4	mind[]	11.558	"row21"

Exercise 3

Read the Large Movie Review Dataset (sampled) available at the following path *Thedata/MoviereviewDataset_sampled.table*. The dataset contains labeled reviews as positive or negative, as well as unlabelled reviews.

Use the Strings to Document node to transforms the strings into documents. Keep only documents categorized as positive reviews.

POS Tag the terms. Pre-process the Documents by filtering numbers, erase punctuations, filter stop words, convert words to lower case, stem them, and keep only nouns, adjectives, and verbs. Create the bag of words for the remaining terms.

Filter the Bag of Words to keep only terms occurring in at least 5 Documents.

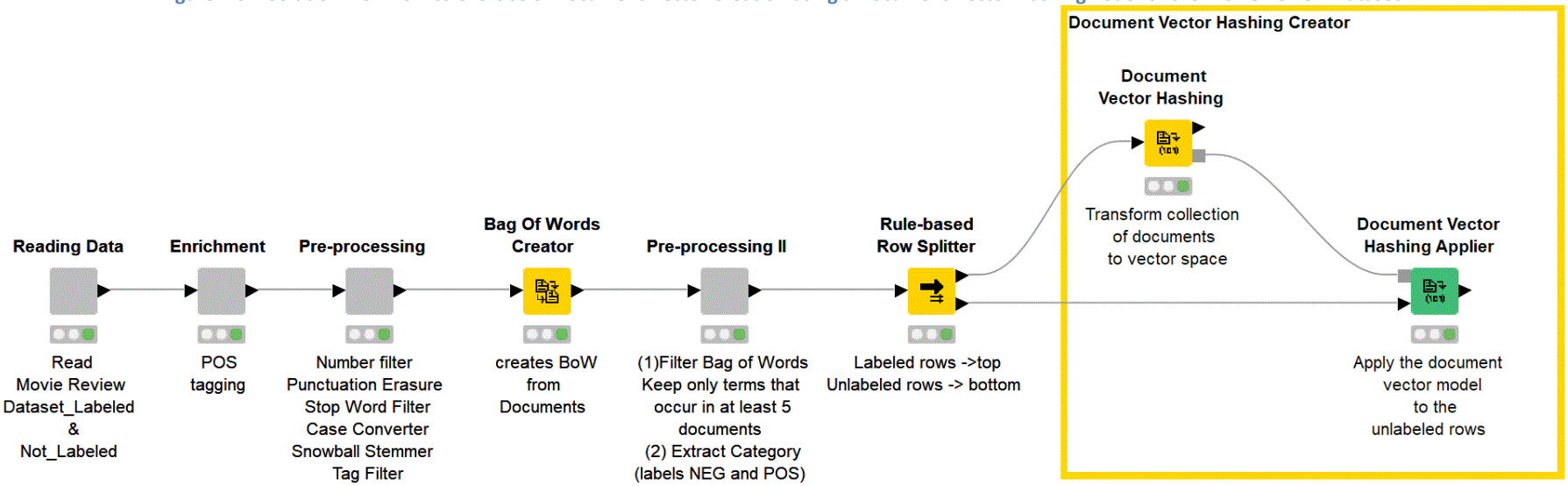
Split the collection of documents in two different sets of data: labelled (category NEG or POS) and unlabeled (missing category). Use the *Rule-based Row Splitter* node to do that.

Build a Document Vector for each labelled Document using a Document Vector Hashing node.

Apply the Document Vector model to each unlabeled Document, using a Document Vector Hashing Applier node.

Solution

Figure 4.31. Solution workflow to exercise 3. Document Vector Creation using a Document Vector Hashing node for the Movie Review Dataset



Chapter 5. Visualization

5.1. View Document Details

It is often said that “one image is better than a 1000 words” to underline the capability of pictures to describe a concept more intuitively than just words or, worse, numbers. This is true also for quantitative text mining. Although the word “quantitative” refers to measured quantities in a text, even in this discipline a picture can help summarize the analysis conclusions. For this reason, KNIME Analytics Platform has devoted a few additional nodes to text visualization. The first step consists of visualization of pure texts, possibly with the added tags. Text visualization of can be implemented with the Document Viewer node.

Document Viewer

The Document Viewer node loads all Documents in the input data table into its View. After execution, right-click the node and select “View: Document View” option from its context menu. The View of the Document Viewer node opens, lists all Documents from the input data table, and allows to select one Document as starting point. A “Quick Search” option allows to isolate the Document of interest, searching by title, authors, source, and category (Fig. 5.1).

After a Document is selected with a double-click from the list, a second view opens to show the Document details, including text, related information, author, and metadata. In this second view, the Document content is displayed on the right, while all other details are displayed on the left under tabs: Search, Document Information, Authors, and Meta-Information.

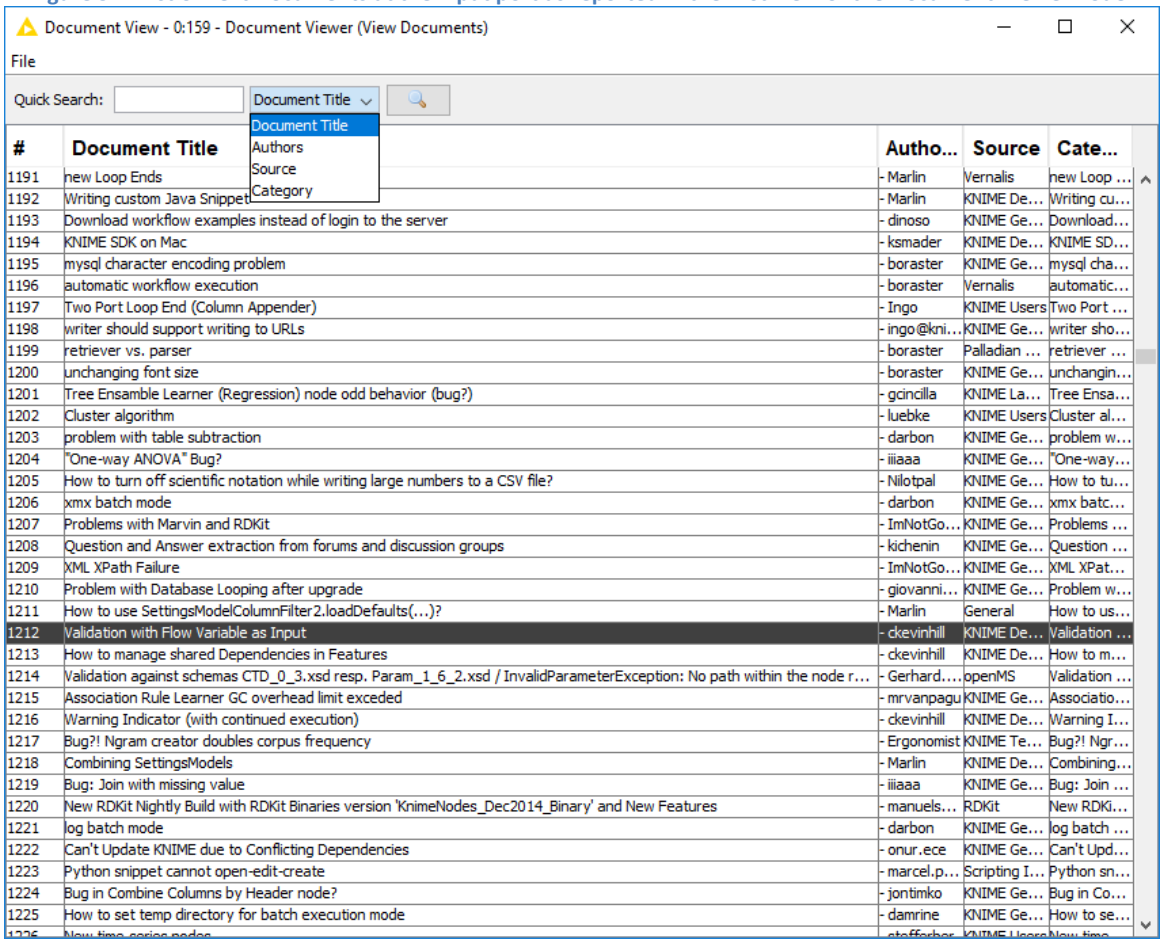
Tab Search is the most interesting one. It allows for word search according to a regular expression and for a few visualization settings. The pen option allows to color words in the text, according to a tag category and a color, both selected nearby. Under tab Search, text mining tags and HTML tags (if any) can also be displayed throughout the text.

Selecting and right-clicking a word In the text opens a list of search engines and web information repositories, such as Google or Wikipedia. Selecting one of them, runs the search or displays the page for that word. The default search engine or web information repository can be set under “Search” on the left of the view.

No configuration settings are required for this node.

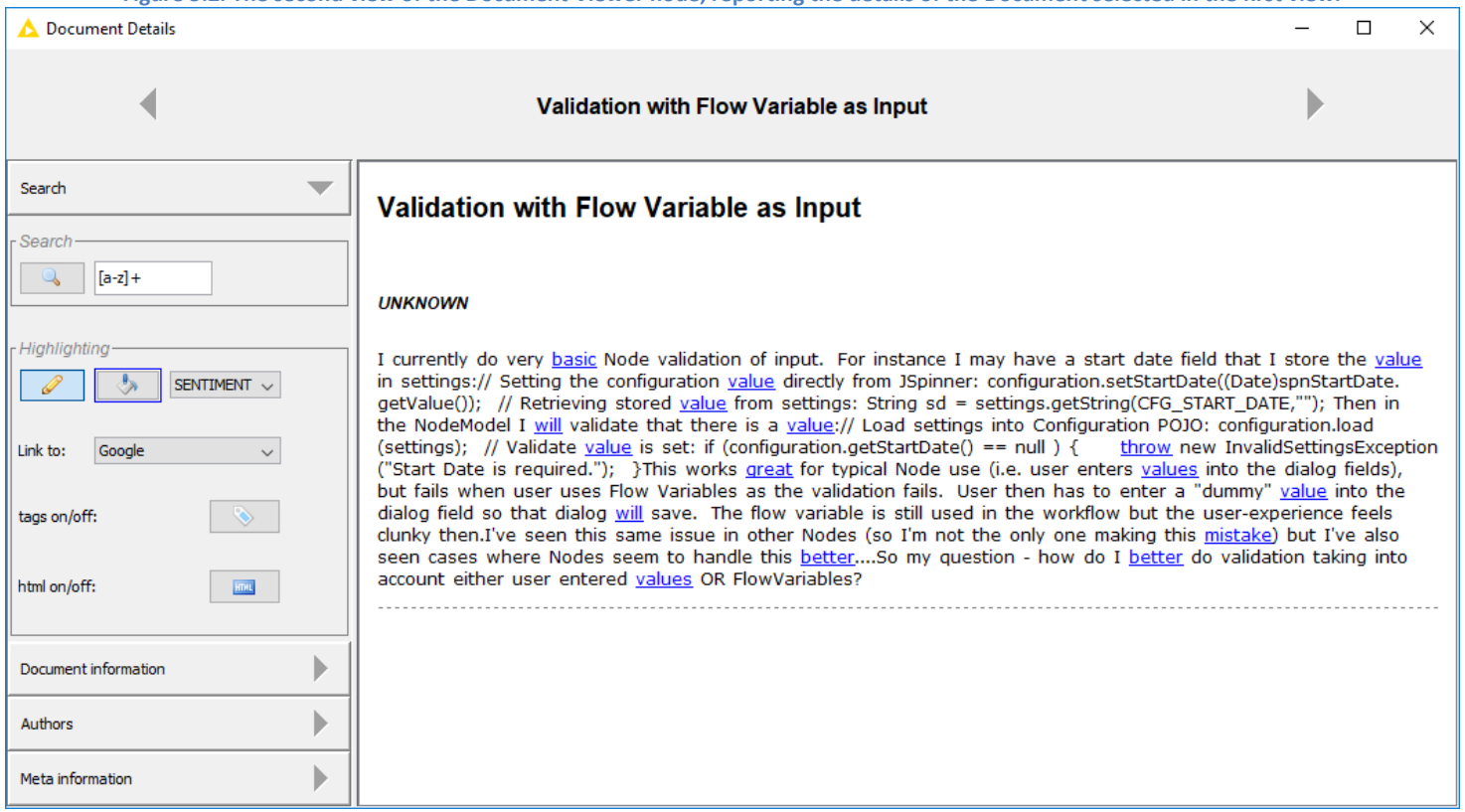
Note. First and second views of the Document Viewer node are connected. Changing the selection in the first view changes the details of the Document displayed in the second view.

Figure 5.1. List of Text Documents at the input port as reported in the first view of the Document Viewer node.



#	Document Title	Autho...	Source	Cate...
1191	new Loop Ends	- Marlin	Vernalis	new Loop ...
1192	Writing custom Java Snippet	- Marlin	KNIME De...	Writing cu...
1193	Download workflow examples instead of login to the server	- dinoso	KNIME Ge...	Download...
1194	KNIME SDK on Mac	- ksmader	KNIME De...	KNIME SD...
1195	mysql character encoding problem	- boraster	KNIME Ge...	mysql cha...
1196	automatic workflow execution	- boraster	Vernalis	automatic...
1197	Two Port Loop End (Column Appender)	- Ingo	KNIME Users	Two Port ...
1198	writer should support writing to URLs	- ingo@kni...	KNIME Ge...	writer sho...
1199	retriever vs. parser	- boraster	Palladian ...	retriever ...
1200	unchanging font size	- boraster	KNIME Ge...	unchangin...
1201	Tree Ensamble Learner (Regression) node odd behavior (bug?)	- gcincilla	KNIME La...	Tree Ensa...
1202	Cluster algorithm	- luebke	KNIME Users	Cluster al...
1203	problem with table subtraction	- darbon	KNIME Ge...	problem w...
1204	"One-way ANOVA" Bug?	- iiiaaa	KNIME Ge...	"One-way...
1205	How to turn off scientific notation while writing large numbers to a CSV file?	- Nilotpal	KNIME Ge...	How to tu...
1206	xmx batch mode	- darbon	KNIME Ge...	xmx batc...
1207	Problems with Marvin and RDKit	- ImNotGo...	KNIME Ge...	Problems ...
1208	Question and Answer extraction from forums and discussion groups	- kichenin	KNIME Ge...	Question ...
1209	XML XPath Failure	- ImNotGo...	KNIME Ge...	XML XPat...
1210	Problem with Database Looping after upgrade	- giovanni...	KNIME Ge...	Problem w...
1211	How to use SettingsModelColumnFilter2.loadDefaults(...)?	- Marlin	General	How to us...
1212	Validation with Flow Variable as Input	- ckevinhill	KNIME De...	Validation ...
1213	How to manage shared Dependencies in Features	- ckevinhill	KNIME De...	How to m...
1214	Validation against schemas CTD_0_3.xsd resp. Param_1_6_2.xsd / InvalidParameterException: No path within the node r...	- Gerhard....	openMS	Validation ...
1215	Association Rule Learner GC overhead limit exceded	- mrvanpagu	KNIME Ge...	Associatio...
1216	Warning Indicator (with continued execution)	- ckevinhill	KNIME De...	Warning I...
1217	Bug?! Ngram creator doubles corpus frequency	- Ergonomist	KNIME Te...	Bug?! Ngr...
1218	Combining SettingsModels	- Marlin	KNIME De...	Combining...
1219	Bug: Join with missing value	- iiiaaa	KNIME Ge...	Bug: Join ...
1220	New RDKit Nightly Build with RDKit Binaries version 'KnimeNodes_Dec2014_Binary' and New Features	- manuels...	RDKit	New RDKi...
1221	log batch mode	- darbon	KNIME Ge...	log batch ...
1222	Can't Update KNIME due to Conflicting Dependencies	- onur.ece	KNIME Ge...	Can't Upd...
1223	Python snippet cannot open-edit-create	- marcel.p...	Scripting I...	Python sn...
1224	Bug in Combine Columns by Header node?	- jontimko	KNIME Ge...	Bug in Co...
1225	How to set temp directory for batch execution mode	- damrine	KNIME Ge...	How to se...

Figure 5.2. The second view of the Document Viewer node, reporting the details of the Document selected in the first view.



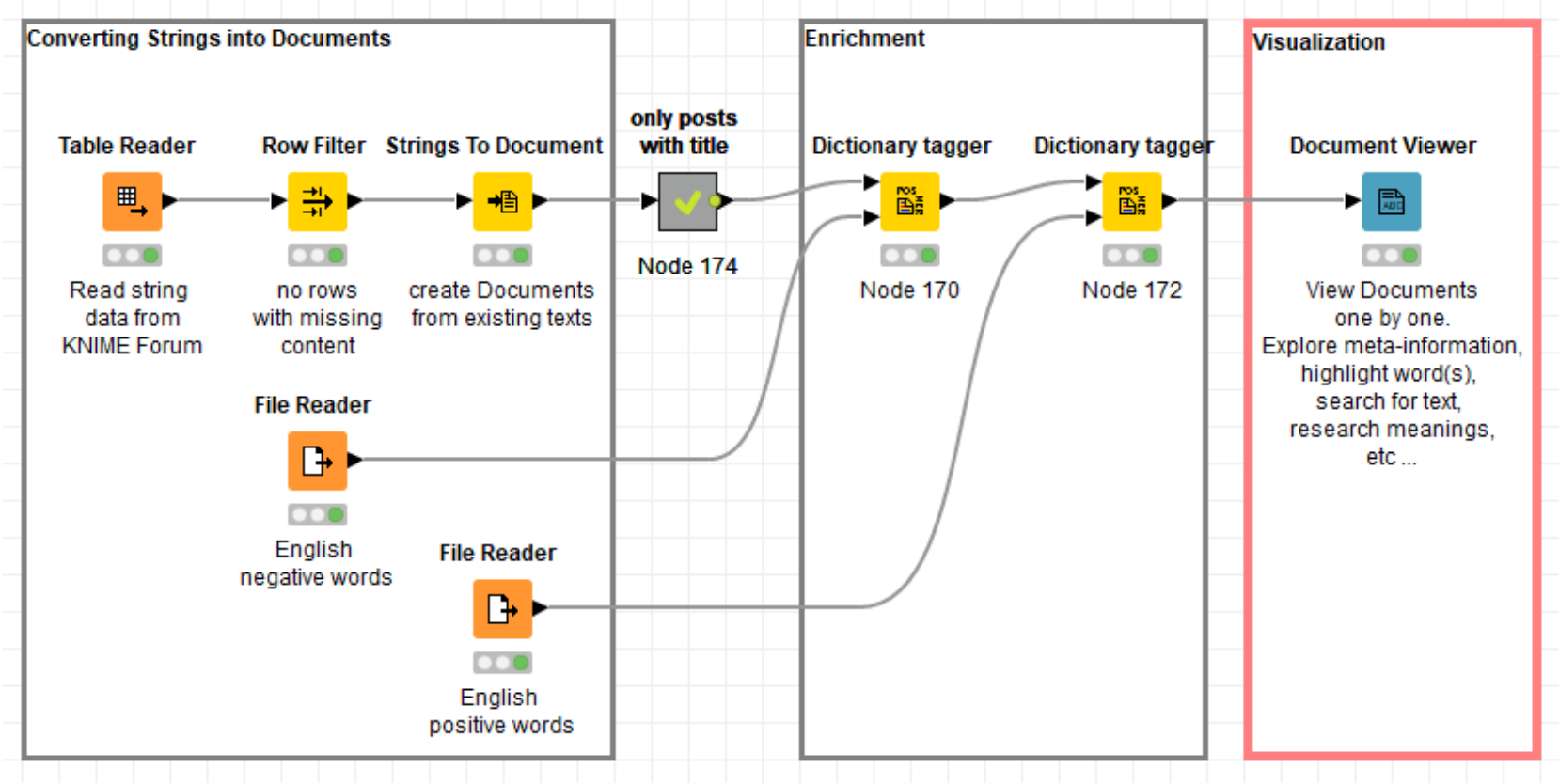
The example workflow, that we have prepared to illustrate how this node works, is named 01_Explore_Document and it is located in Chapter5 folder. Here we build on top of the workflows developed in the previous chapters.

First, we read the KNIME Forum dataset. The dataset contains the posts from the KNIME Forum, i.e. posts and related comments. Posts and comments can be distinguished through the post-title field. Unlike the original posts, comments have a missing post-title value. However, comments and posts in the same thread share the same topic_title value. After importing the forum texts, we decided to work only on the original posts and avoid the comments. Thus, we extract only those rows with not missing post-title value.

We then enrich our Documents with a few sentiment tags. We feed two lists of words, one for negative and one for positive words, from the MPQA Opinion Corpus into a Dictionary Tagger node, to tag the matching words as NEGATIVE SENTIMENT and as POSITIVE SENTIMENT respectively.

Now we have a list of Documents, with some tagged words in the SENTIMENT tag category, ready for visual inspection with the Document Viewer node. From the list of Documents we chose Document #1212 with title “Validation with Flow Variable as Input” by user ckevinhill, the details of which are displayed in the second view of the node (Fig. 5.1 and 5.2).

Figure 5.3. This workflow (Chapter5/01_Explore_Document) imports the KNIME Forum data set, isolates posts, and tags words as POSITIVE or NEGATIVE according to the MPQA Opinion Corpus



5.2. Word Cloud

Now the question is: what are people actually discussing in the KNIME Forum? What is the most discussed topic and why? An easy visual way to get an overview of contents in large text collections is the word cloud. Exactly those word clouds that you see on newspapers summarizing politicians’ talks can also be built using the KNIME Text Processing Extension.

The word cloud of all posts, without comments, available in the KNIME Forum dataset can be seen in Fig. 5.4. Now how did we get to that? The necessary node here is the Tag Cloud (Javascript) node.

A tag (or word) cloud is a representation of words from a text, where the importance of the word in the text is directly translated into its visual properties in the image. Usually font size and color are the visual properties used for this purpose and the word frequency is taken as the measure of the word importance.

Figure 5.4. Word Cloud for all threads in the KNIME Forum Dataset, colored using the default color map. Top 1000 most frequent Terms only.

Tag Cloud (Javascript)

The Tag Cloud (Javascript) node needs a list of Terms and optionally some measure of their frequency at the input port. It then displays all terms in a word cloud. The final word cloud is available as a view from the context menu or as an image at the node output port. In the configuration window, three tabs set all required parameters for this view.

Options. Here you can set the Term column ("*Tag column*") and, optionally, the column with associated frequencies ("*Size column*") for the word size. "Size column" can also be set to "none". In this case, all words are displayed with the same size.

Frequencies in "Size column" can be aggregated as sum of all frequencies for the same word (flag "*Aggregate tags*") and/or displayed independently of the associated Tag (flag "*Ignore term tags*"). If frequency is not available, but a size graphical property has been previously associated with each row, flag "*Use row size property*" will use that to size the words in the cloud.

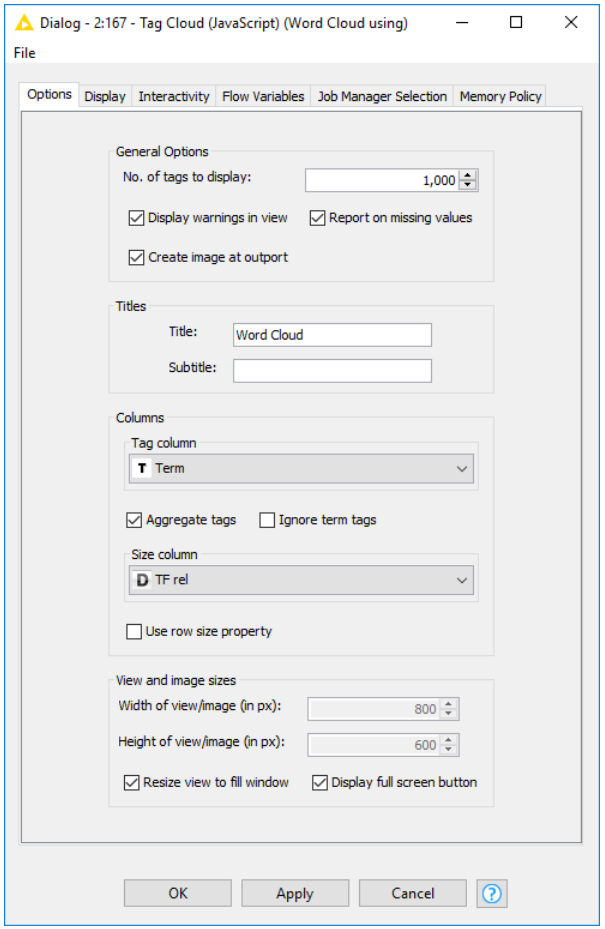
Option ***"No. of tags to display"*** limits the words to be displayed to the top N. This is for better readability.

Flag ***"Create image at output port"*** creates a static image object at speed execution expenses. Image size options are displayed in the lower part of this tab. A few additional flags are available to report warnings in case of missing values or other problems in the input data.

Display. This tab contains options about the visualization itself, such as color, fonts, and cloud orientation. Words in the cloud can be colored according to the default color map (Fig. 5.4) with disabled flag "Use row color property" or according to the color scheme set by a previous Color Manager node with enabled flag "Use row color property" (Fig. 5.7).

Interactivity. This tab contains control options for the interactive view, such as word selection.

Figure 5.5. Configuration window of the Tag Cloud (Javascript) node



The Tag Cloud (Javascript) view displays the generated word cloud. Here font size, transparency, and boldness indicate the most important words, according to the adopted frequency measure. The most frequently recurring words in the KNIME Forum are "knime", "node", "column", and to a lesser extent "file", "table", "row", and "workflow". All of them, besides "file", belong to the KNIME Analytics Platform data and work structure.

In the current workflow, we have produced two word clouds. Word cloud in figure 5.4 disabled option “Use row color property” and relied on the default coloring scheme for the top 1000 most frequent Terms. Word Cloud in figure 5.7 relies on the color heatmap defined by a Color Manager node, where darker green indicates less frequent words and brighter green indicates more frequent words. The brightest words correspond to the most frequent words; that is again: “knime”, “node”, and “column”.

Figure 5.6. Green shades heatmap for word frequencies

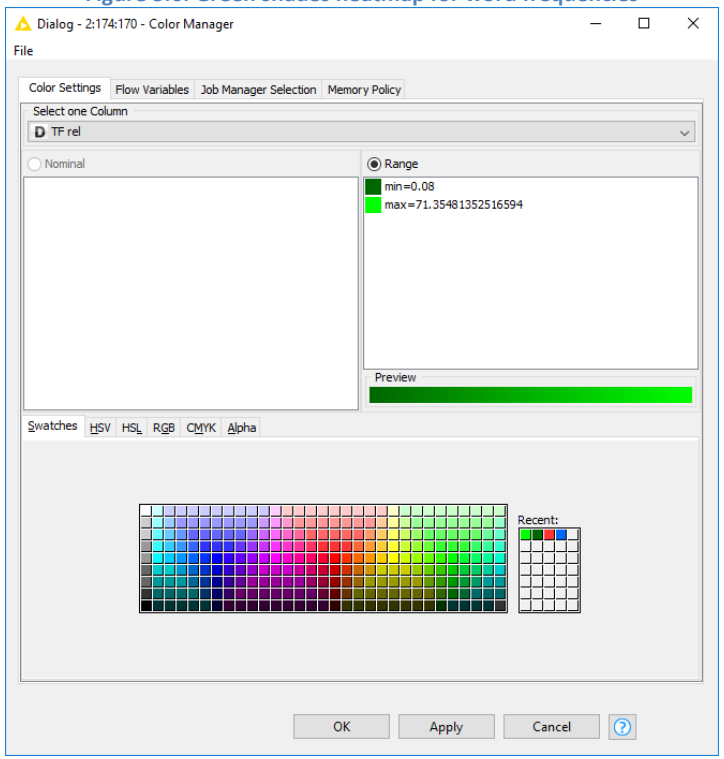
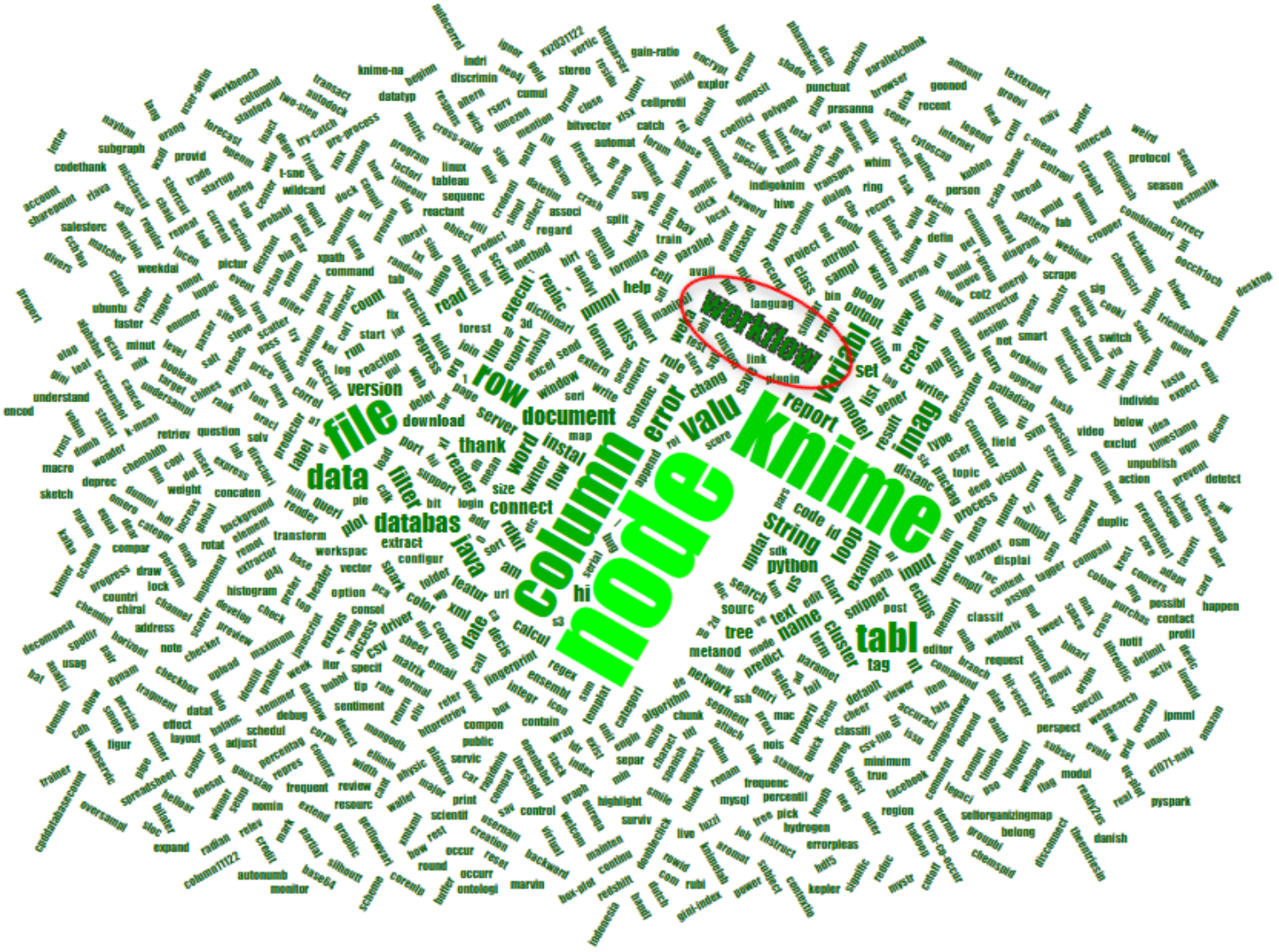
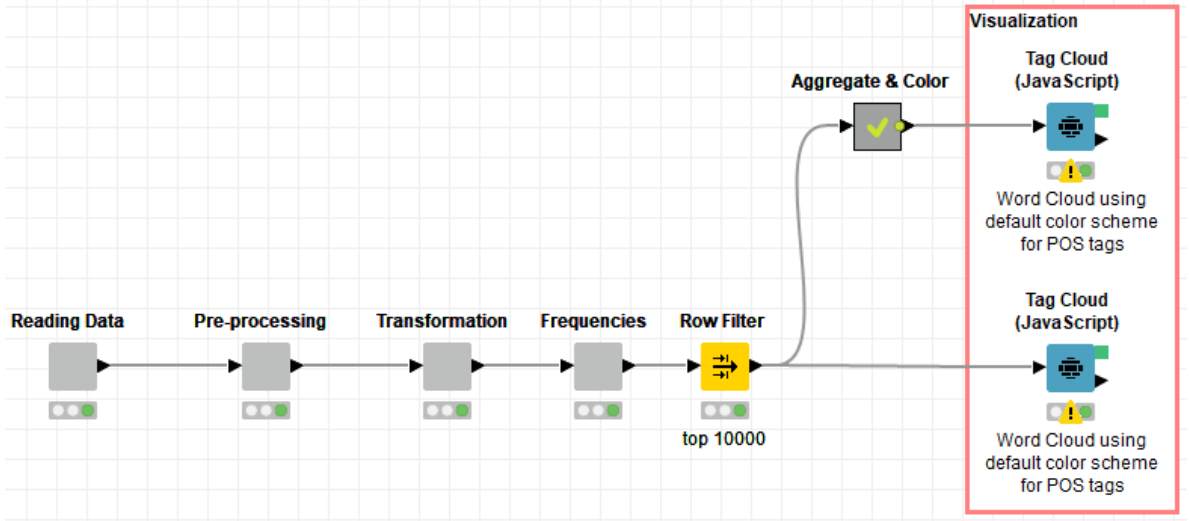


Figure 5.7. Word Cloud of same input data as in figure 5.4, but colored with the color scheme defined in figure 5.6



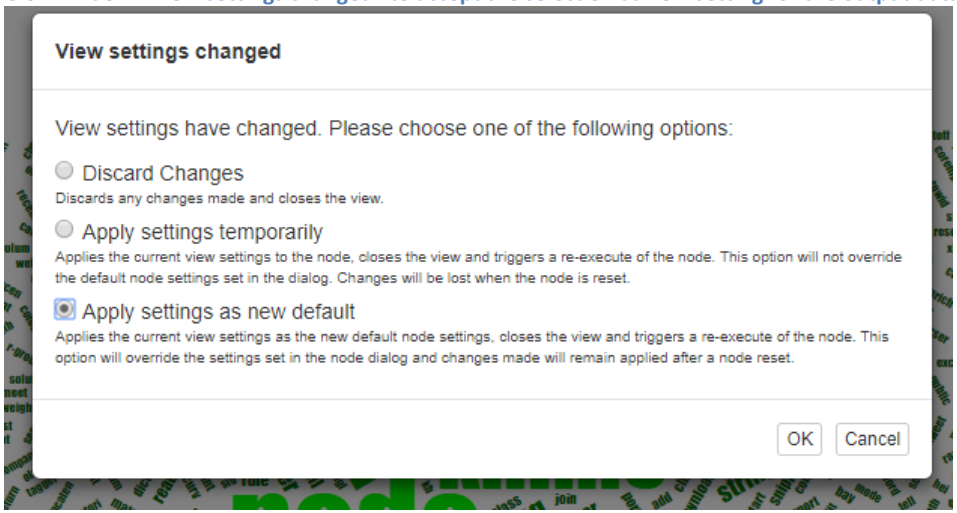
The full workflow displaying the word cloud using the default color scheme and the word cloud using the customized color scheme is reported in figure 5.8.

Figure 5.8. Workflow Chapter5/02_Word_Cloud generating the word clouds in figure 5.4 and in figure 5.7 both with Tag Cloud (Javascript) nodes



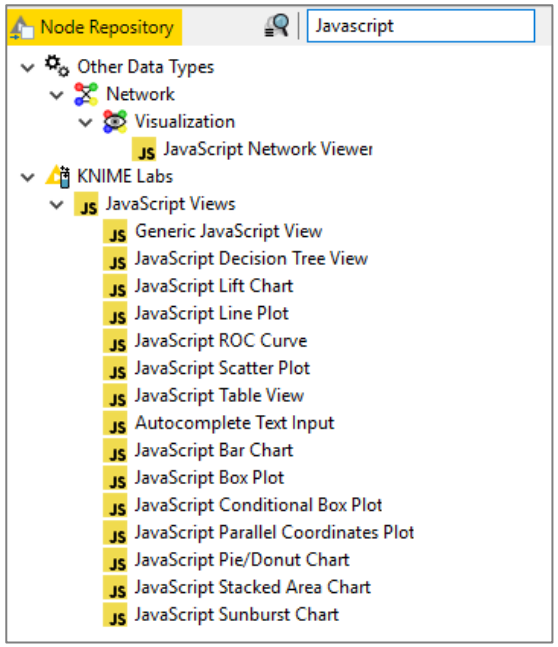
If you paid close attention, the word “workflow” – circled in red in the word cloud in figure 5.7 - has been selected. Indeed, the Tag Cloud (Javascript) view is interactive and words can be selected with a click. After clicking button “Close” in the lower right corner of the view, you will be presented with the option of keeping the new selection as your new setting for the output data table. If you accept that, by selecting “Apply settings as new default” and then “OK”, all data rows with Term “workflow” will take value “true” in column “Selected (Javascript Tag Cloud)” in the output data table.

Figure 5.9. Window “View settings changed” to accept the selection as new setting for the output data table



5.3. Other Javascript based Nodes

Figure 5.10. Javascript based nodes for data visualization



Visualization of text Documents data does not necessarily need custom views. Traditional views will do just as well. In this section we would like to display the frequency values of the top 20 words on a classic bar chart. Classic chart and plot views are produced in by a number of Javascript based nodes. The Javascript Bar Chart node is what we need in this case.

After reading the input Documents, cleaning them up, and transforming them into bags of words, we calculate some frequency measures for each Term in each Document: relative term frequency (TF rel), Inverse Document Frequency (IDF), and the combined frequency TF-IDF. Notice that we do not perform any tagging of the words in the texts.

The goal here is to find the top 20 words with highest frequency across all Documents. Thus, we need to aggregate the frequency measures across all Documents. For that, we use a GroupBy node, grouping by Term and calculating: Sum(TFrel), Mean(TF-IDF), Sum(TF-IDF).

After sorting by frequency in descending order, the top 20 words - according to each one of those frequency measures - are extracted. To display these top 20 words and their frequencies for the three frequency measures, we use three Javascript Bar Chart nodes.

Figure 5.11. Top 20 words with highest Sum(TFrel)

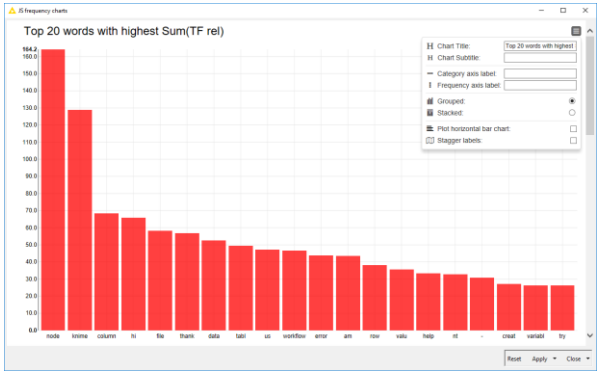


Figure 5.12. Top 20 words with highest average TF-IDF

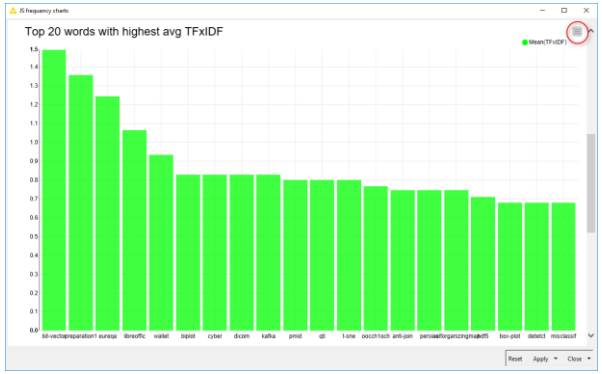
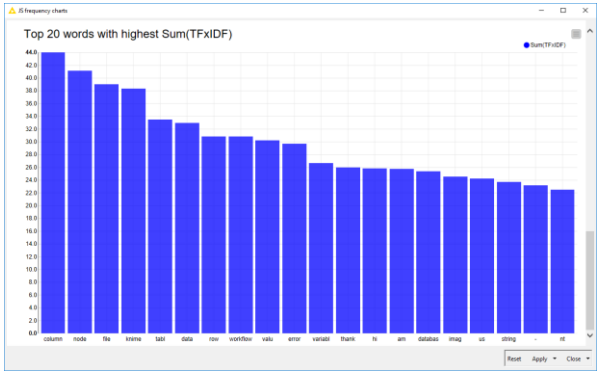


Figure 5.13. Top 20 words with highest total TF-IDF



Bar Chart (Javascript)

The Bar Chart (Javascript) node displays one or more columns in the input data table in the form of a bar chart. It produces an interactive view and an image. The view can be opened via the option “Interactive View: Group Bar Chart” in the context menu.

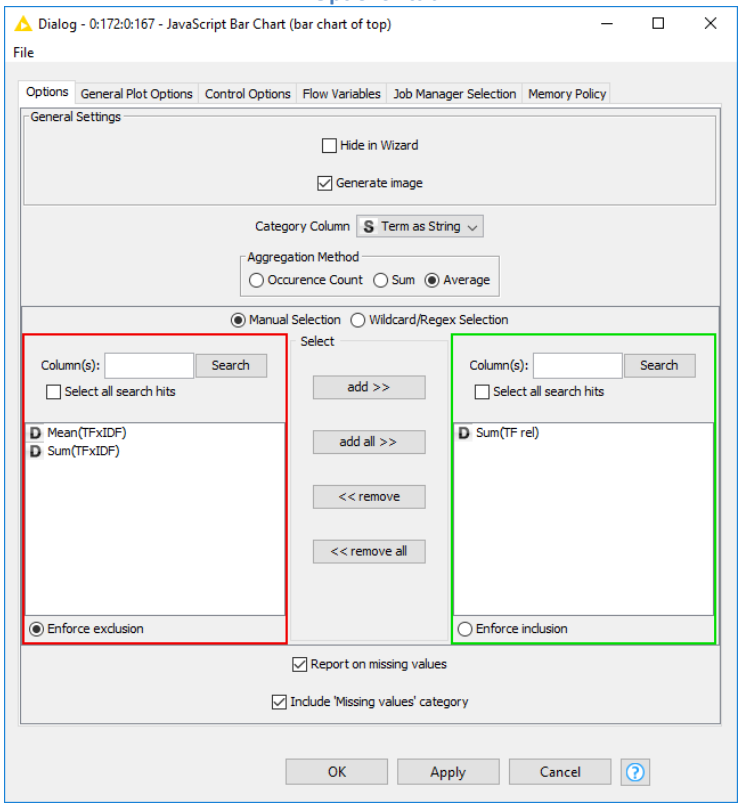
Its parameters can be set in 3 of the configuration window tabs: “Options”, “General Plot Opions”, “Control Options”.

- Tab “Options” defines the x-axis via the Category Column and the y-axis via the Include/Exclude framework as well as a few additional parameters, like whether to generate the image

- Tab “General Plot Options” contains settings specific to the plot area, like title, subtitle, labels, legend, etc …

- Tab “Control Options” defines which interactive controls will be enabled in the final view, like editing title and/or subtitle and changing axis labels.

Figure 5.14. Configuration window of the Javascript Bar Chart node: “Options” tab



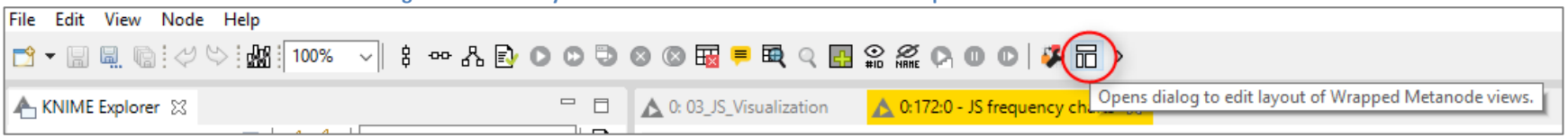
Note. All Javascript nodes produce an interactive view and for all of them the degree of interactivity can be set in the configuration window in Control Options tab.

In figure 5.12, you can see, circled in red in the top right corner, a list symbol. This button opens the interactivity menu allowing to change some of the settings in the plot, like for example title, axis, orientation, etc… (see effect in Fig. 5.11).

The three bar charts are actually part of a single view: the view of the wrapped metanode containing all three Javascript Bar Chart nodes. Indeed, when one or more Javascript based nodes are collapsed together inside a wrapped metanode, the wrapped metanode acquires a view itself as the

combination of the views of all underlying Javascript nodes. The layout of this combined view can be arranged via the layout editor button on the right end of the tool bar at the top of the KNIME workbench (Fig. 5.15).

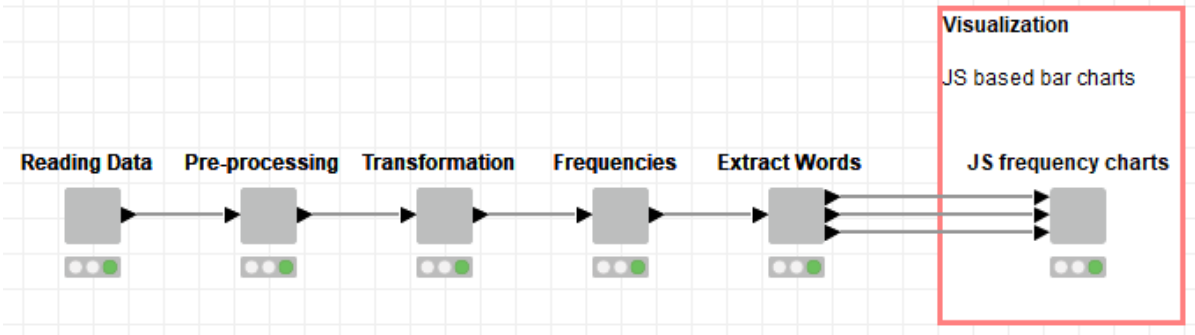
Figure 5.15. The layout editor button in the tool bar at the top of the KNIME workbench.



The top 20 words change if we consider different frequency measures, as we can see from the Javascript Bar Chart nodes' views. The sum of the relative frequencies reflects the word cloud representation of the word relative frequency across Documents. "node" and "knime" are still the most frequently used words in the whole forum dataset. A similar frequency distribution, with the addition of the word "file", is found using sum(TF-IDF) for words across all Documents. The importance rank changes drastically when averaging rather than summing TF-IDF values across Documents. In this case, the words best identifying the question formulated in a post are "bit-vector", "eureqa", "preparation", "wallet", "cyber", etc … So, beware of the frequency measure you use. It will change the whole word importance/information concept.

This workflow is located in Chapter5/03_JS_Visualization (Fig. 5.16). Here the standard pipeline of data reading, pre-processing, transformation into bags of words, frequency calculation is clearly visible. The last two wrapped metanodes contain the frequency aggregations across Documents and the Javascript based bar chart nodes, respectively.

Figure 5.16. Workflow Chapter5/03_JS_Visualization visualizing the words with highest frequency on a Javascript based bar chart

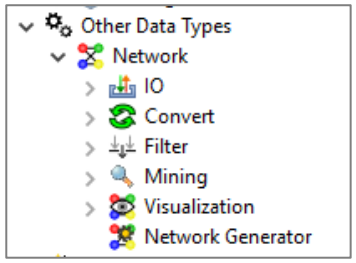


5.4. Interaction Graph

In this section we present a visualization technique which is not strictly designed for texts and words. However, it is often used to visualize interactions, and particularly word-based interactions, among social media users, such as forum users for example. Such a visualization technique is the network

graph. In this case, the social media users become the nodes of the network and the text exchanges, i.e. the interactions, become the edges connecting the two user-nodes. For more details, you can check the Graph Theory page on Wikipedia.

Figure 5.17. The Network category in the Node Repository panel



KNIME Analytics Platform includes a Network category with nodes for network analysis and visualization. The Network category can be found in the Node Repository under Other Data Types.

Before visualizing the network graph, we need to build the network; and before building the network we need to prepare the map of interactions between each pair of users. The interaction map takes the following form:

```
<Thread ID> <post-text> <comment-text> <comment-author> <post-author>
```

Following this format we have one post and a list of comments for each forum thread.

In the KNIME Forum dataset, <Thread ID> is found in column named topic-title, <comment-author> and <post-author> are both in column posted-by, and <comment-text> and <post-text> are in column post-content. The column that helps recognizing comments vs. post in the same thread is the column named post-title. post-title value is missing for comments.

Let's focus on the visualization of one of the small forums in the dataset, i.e. the CDK forum. The small number of users, and consequently of interactions, will make for a clearer picture without altering the preparation and visualization steps.

After reading the data and filtering in all data rows with column forum=CDK, we need to transform the original data structure in the form described above. To obtain that, first comments are joined with the post and their authors for each thread; then the number of occurrences of each pair comment-author (commenter) and post-author (author) are counted. Such number of occurrences is the number of interactions exchanged throughout the whole dataset. On the side, the number of comments and posts are also counted for each author and commenter. Based on his/her total number of posts and his/her total number of comments a forum user becomes an author (blue) or a commenter (red) and his/her degree of activity (role) is defined as the total number of his/her posts or comments. We have built the interaction map of the CDK forum users (Fig. 5.18).

Now, let's build a network object on the interaction map of the forum users, using nodes from the Network category. Authors and commenters will be the network nodes; no_exchanges - as the number of interactions between author and commenter - will be the network edge between two nodes; and the role value defines the node size. First the Object Inserter node creates the network object from the interaction matrix. Then the three following Feature Inserter nodes add the properties Node Size from the poster activity degree (role), Node Color from the author or commenter feature, and Font Size fixed to 40pt for each graph node. The sequence of Network nodes is shown in figure 5.19.

Note. The output port of Network nodes is a light green square. This indicates a network object.

Figure 5.18. Interaction Map of CDK Forum users

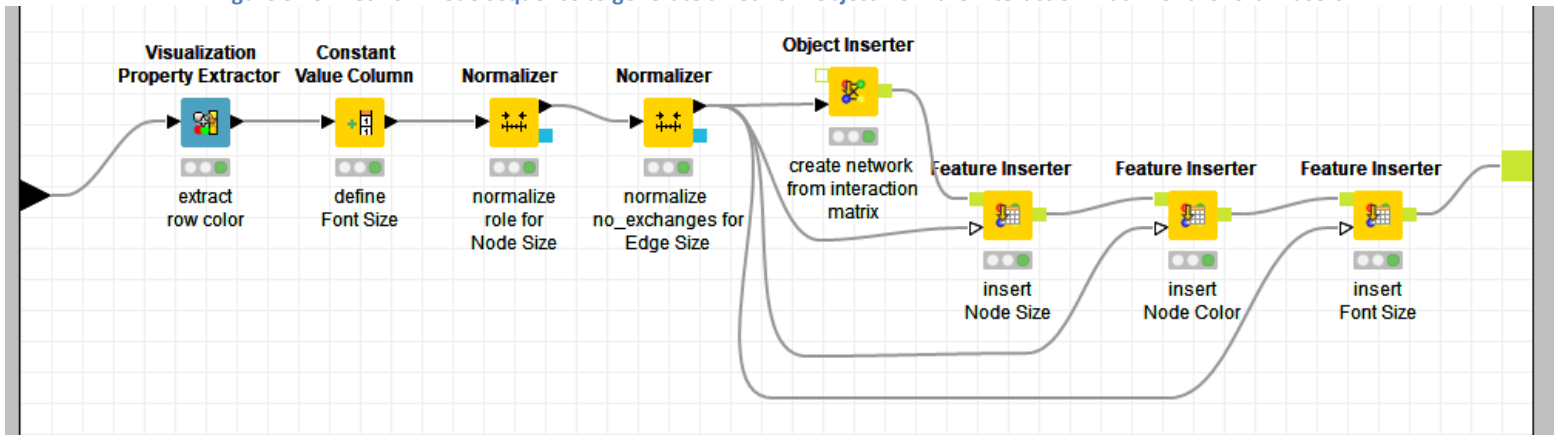
Table with Colors - 0:211:189 - Color Manager (commenters -> red)

File Hilite Navigation View

Table "default" - Rows: 47 Spec - Columns: 7 Properties Flow Variables

Row ID	commenter	no_exchanges	author	no_posts	no_answers	role	prediction
Row0_Row0_...	Stephan	1	Aaron Hart	1	0	1	author
Row1_Row1_...	Stephan	2	ChEMBLCur...	1	0	1	author
Row2_Row2_...	Stephan	1	Grenix	1	0	1	author
Row3_Row3_...	Stephan	4	INAKI	1	0	1	author
Row4_Row4_...	Stephan	8	InsilicoCons...	2	0	2	author
Row4_Row5_...	richards99	1	InsilicoCons...	2	0	2	author
Row4_Row6_...	thor	1	InsilicoCons...	2	0	2	author
Row5_Row7_...	Stephan	7	Jojo007	2	0	2	author
Row6_Row8_...	Stephan	7	K4A	2	0	2	author
Row7_Row9_...	Marlin	1	LM	1	0	1	author
Row7_Row10...	fabienc	3	LM	1	0	1	author
Row7_Row11...	s.roughley	1	LM	1	0	1	author
Row8_Row12...	Stephan	2	Rinker	1	0	1	author
Row9_Row13...	Stephan	7	SOH979	2	0	2	author
Row10_Row1...	eduece99	2	Stephan	1	111	112	commenter
Row11_Row1...	thor	2	akosgmbh	1	0	1	author
Row12_Row1...	richards99	2	dischiessel	1	0	1	author
Row13_Row1...	Stephan	7	eduece99	4	2	6	author
Row13_Row1...	richards99	2	eduece99	4	2	6	author

Figure 5.19. Network node sequence to generate a network object from the interaction matrix of the forum users



Object Inserter

This is the node that takes a matrix as input and produces a network object as output. In order to build a network, it needs at least two of the following inputs: source node, destination node, and connections.

Tab “Options”

Here the node accepts the basic information to build the network: node 1 (source), node 2 (destination), edge (connection). Each node and edge can display an explanatory label, which can be the same as the node ID and edge ID (default) or can come from another column in the input data table.

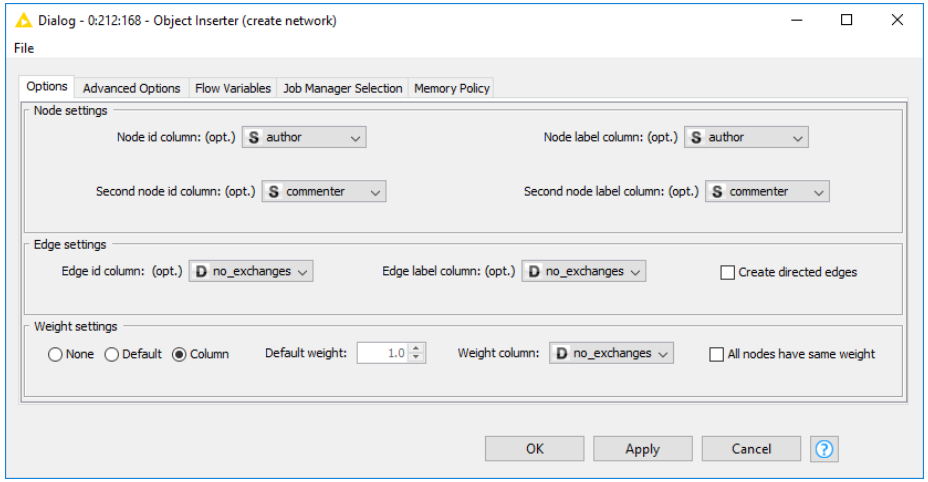
Edges can show the flow direction (directed edges) or just the connection. The flag “Create directed edges” allows to display the flow direction.

Edges and nodes can carry a weight, which will result in thickness and size in the graph image. In the lower part of the configuration window you can define the edge weight as: none, a default fixed value, or a dynamic value from an input column. There you can also force all nodes to have the same size.

Tab “Advanced Options”

It is possible to input the node/edge information through a single String column (node partition column). Column and separation character can be set in the “Advanced Options” tab.

Figure 5.20. Tab “Options” in configuration window of Object Inserter node



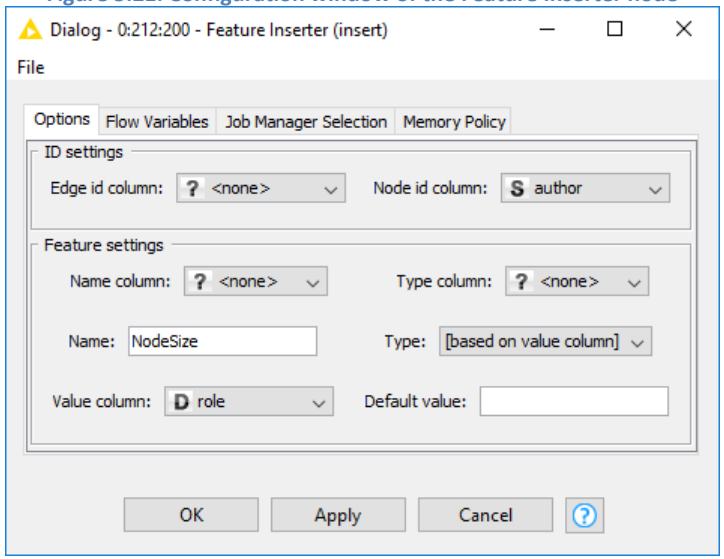
Feature Inserter

Nodes and edges in a network object can own features. Classic features are color and size. Such features cannot be defined at network creation, but need to be added later with a Feature Inserter node.

Each network feature has a name, a type, and a value, and it is associated to either nodes or edges. Thus, the configuration window of the Feature Inserter node requires:

- The edge ID and/or node ID column associated with this feature
- The feature name either from an input column or as fixed value
- The feature value either from an input column or as a fixed value
- The feature type either from an input column or automatically detected from the feature value column

Figure 5.21. Configuration window of the Feature Inserter node



The node that takes a network object as input, transforms it into a graph, and visualizes it in the form of an interactive view or of an image is the Javascript Network Viewer node.

For the output graph, in our example workflow, we used a concentric layout. Other layouts are available. Some layouts display some network better than others. We set the visual node size and color, label size and edge width from the corresponding network features. We also enabled all interactivity options in the View Control tab to make the final view as interactive as possible.

The final workflow is shown in figure 5.23 and the resulting graph representing the interactions of the CDK forum users in figure 5.24.

In the network graph, we can see that user “Stephan” is the central piece of this forum. He is extremely active (largest node), he is a commenter (red), and he answers all users equally (central position). We guess he qualifies as the expert of the forum. There are other commenter users (in red) throughout the graph, but they are less central, which means they are more dedicated to some posts and topics, and less active. In general, there are more poster authors than commenters, as it should be, since there are more users asking questions than users answering. Finally, notice the thick exchange between user “Stephan” and user “gcincilla”. This denotes a few post and comment exchanges between the two.

Network Viewer (Javascript)

This node creates the graph for the input network object and displays it in an interactive view and in an output image.

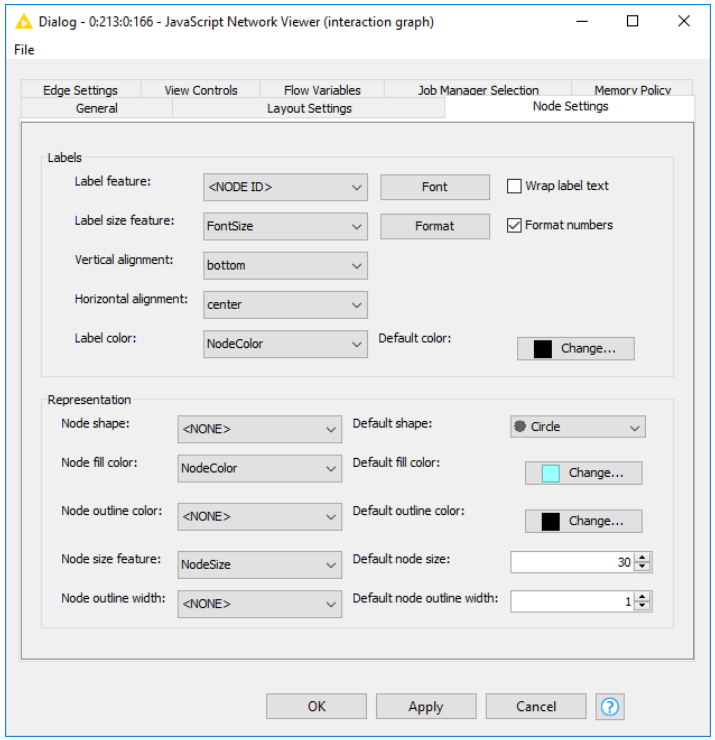
The graph view is controlled by many parameters: the layout, the labels, the node visualization settings, the edge visualization settings, the interactivity levels. Therefore, the node configuration window includes 5 tabs.

General. In this tab, settings cover the list of available graph layouts - the list will be shown in the interactive view - the output image size, the chart title and subtitle.

Layout Settings. Here the default graph layout is selected. It can always be overwritten within the interactive view. A few graphical Javascript based options are available and can be set for the selected layout.

Node Settings. Here are all the node settings, covering the label and the representation of the node itself. For the node labels, you can select: text ("label feature"), size, alignment, color, font, and format. Text, size, and color can come from previously inserted node features. If none is selected, default values are applied.

Figure 5.22. Configuration window of the Javascript Network Viewer



For node representation, you can define shape, color, size, and outline color and width. All those settings can be assigned from previously inserted node features. If none is selected, default values are applied.

Edge Settings. As in the previous tab, possible options for the edges are: label text, size, and color, as well as edge width and color. All those settings can be assigned from previously inserted edge features. If none is selected, default values are applied.

View Controls. This tab is common to all Javascript based visualization nodes. It contains the enabling flags for the interactivity of the various components in the node view. In this case: zooming, title and subtitle editing, node and edge property controls, actions related to selection and filtering events.

Figure 5.23. Workflow Chapter5/04_Interaction_Graph builds the CDK forum interaction matrix, transforms it into a network object, and visualizes the network graph.

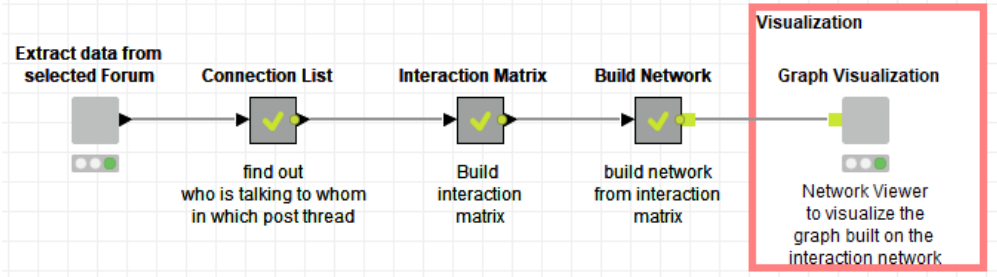
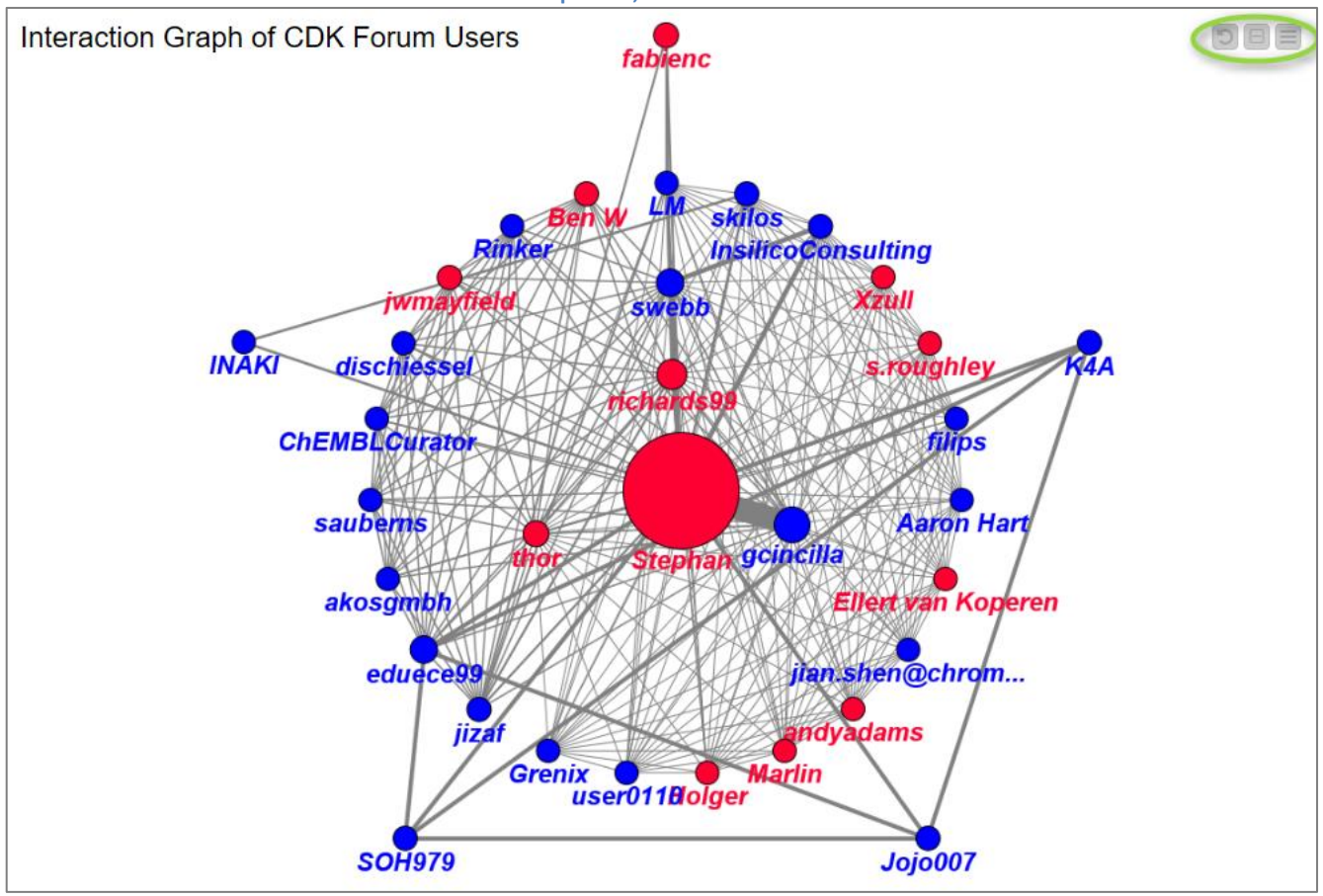


Figure 5.24. Network graph showing the interactions between CDK forum users. Thicker edges indicate a higher number of interactions; larger nodes indicate more active users. Blue users are posters, red users are commenters.



5.5. Exercises

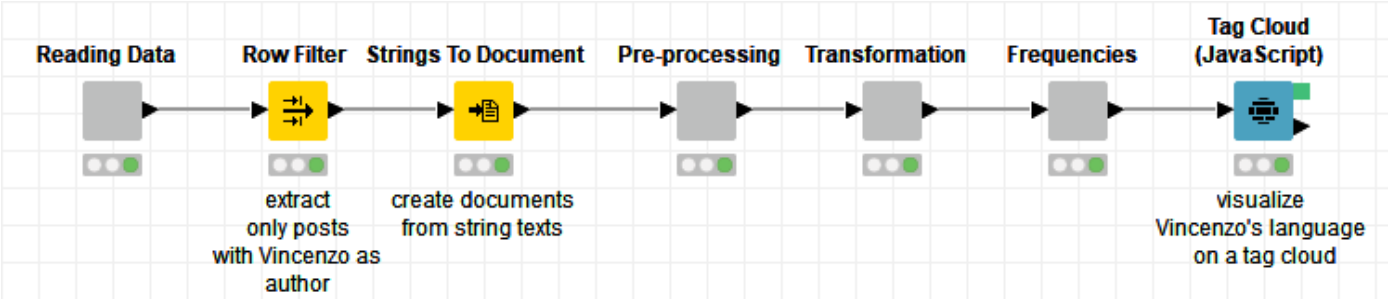
Exercise 1

From the KNIME forum data set, build the word cloud for all posts and comments of a single forum user. As an example, you can use posted-by=vincenzo, where Vincenzo is the author of this book.

Solution

Figure 5.25 Word cloud of Vincenzo's posts and comments in the KNIME forum data set. As you can see from the size of the words "thank" and "pleas" Vincenzo is a very polite person. Also the biggest and most central word here is "Vincenzo" due to his habit of signing posts and comments.

Figure 5.26 Solution workflow of this exercise available under Chapter5/Exercises/Exercise1_WordCloud_of_a_ForumUser



Exercise 2

From the text of "Romeo and Juliet" in file "pg1513.epub" in folder "Thedata",

- tag the tragedy characters as PERSON using a Wildcard Tagger node
- extract them using the Tag Filter node
- run the "Term co-occurrence counter" node to count the isolated terms (PERSON) occurring together in a sentence (the speaker and the addressee)
- Build a network using the tragedy characters as nodes and the number of co-occurrences as edges
- Calculate the authority score of each character by means of the Network Analyzer node and use this value for the node size
- Visualize the network graph

Solution

After reading the data with the Tika Parser node, we tag the characters as PERSON with a Wildcard Tagger node in the "Tag Characters" metanode. In the same metanode, we also filter out all non-PERSON words.

Now we have sentences made of only character names. Since this is a theater play, the first word is always the speaking character. The second name, if any, is the character spoken to. So, counting the number of co-occurrences of character names in a sentence is a measure of their interactivity. We will use the character names as nodes and the number of co-occurrences as edges when building the network object.

The edge width is then the number of co-occurrences. The node size could be the number of times a character name appears in the text. Alternatively, and more sophisticatedly, we could calculate the authority score – as it is done for the analysis of modern social media – of the character in the play. The "Network Analyzer" node calculates such and other similar scores. The character with the highest authority score should be represented in the graph with a bigger node. Let's also color the nodes by family: the Montague in blue and the Capulet in red, church in green and prince in orange.

Using an Object Inserter node we build the network using the co-occurring terms as nodes and the number of co-occurrences as edges. Then we insert a number of additional features with a Multi-Feature Inserter node: number of co-occurrences for edge width, authority score for node size, family color for node color.

Finally, the Javascript Network Viewer visualizes the network graph. The final workflow is shown in figure 5.27 and the final graph in figure 5.30.

Figure 5.27. Workflow to tag characters, calculate character occurrences and co-occurrences, build network, and display character interaction graph for "Romeo and Juliet"

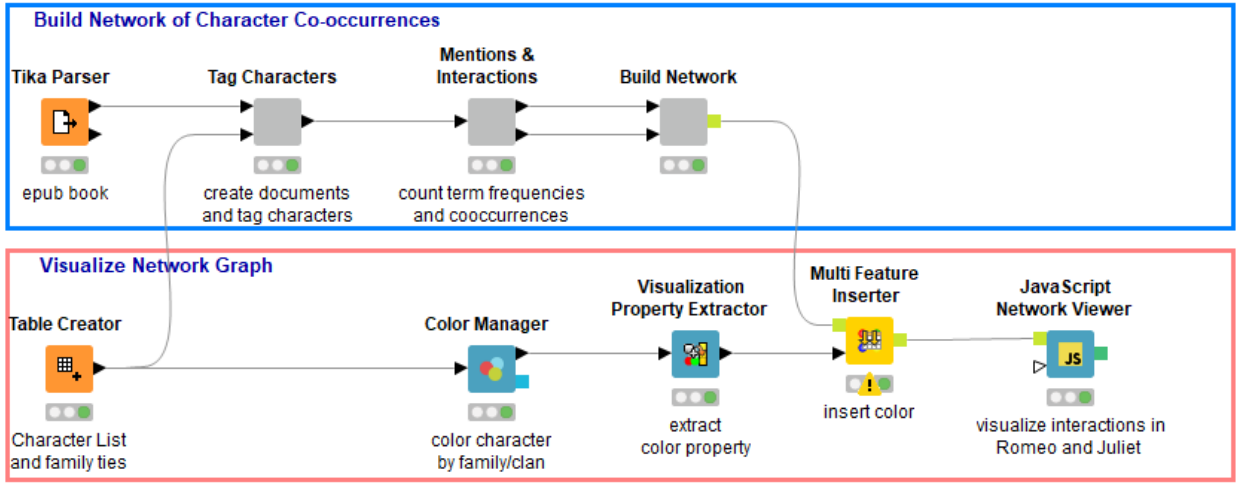


Figure 5.28. Content of the "Mentions and Interactions" wrapped metanode

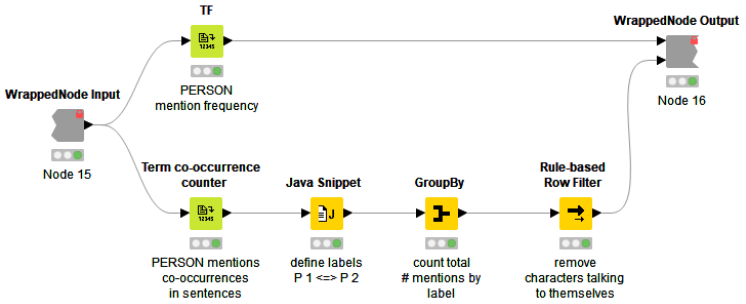


Figure 5.29. Content of the "Build Network" wrapped metanode

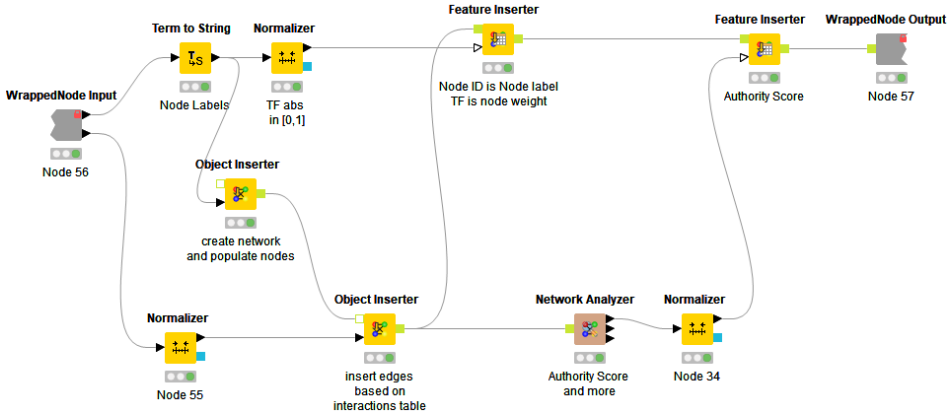
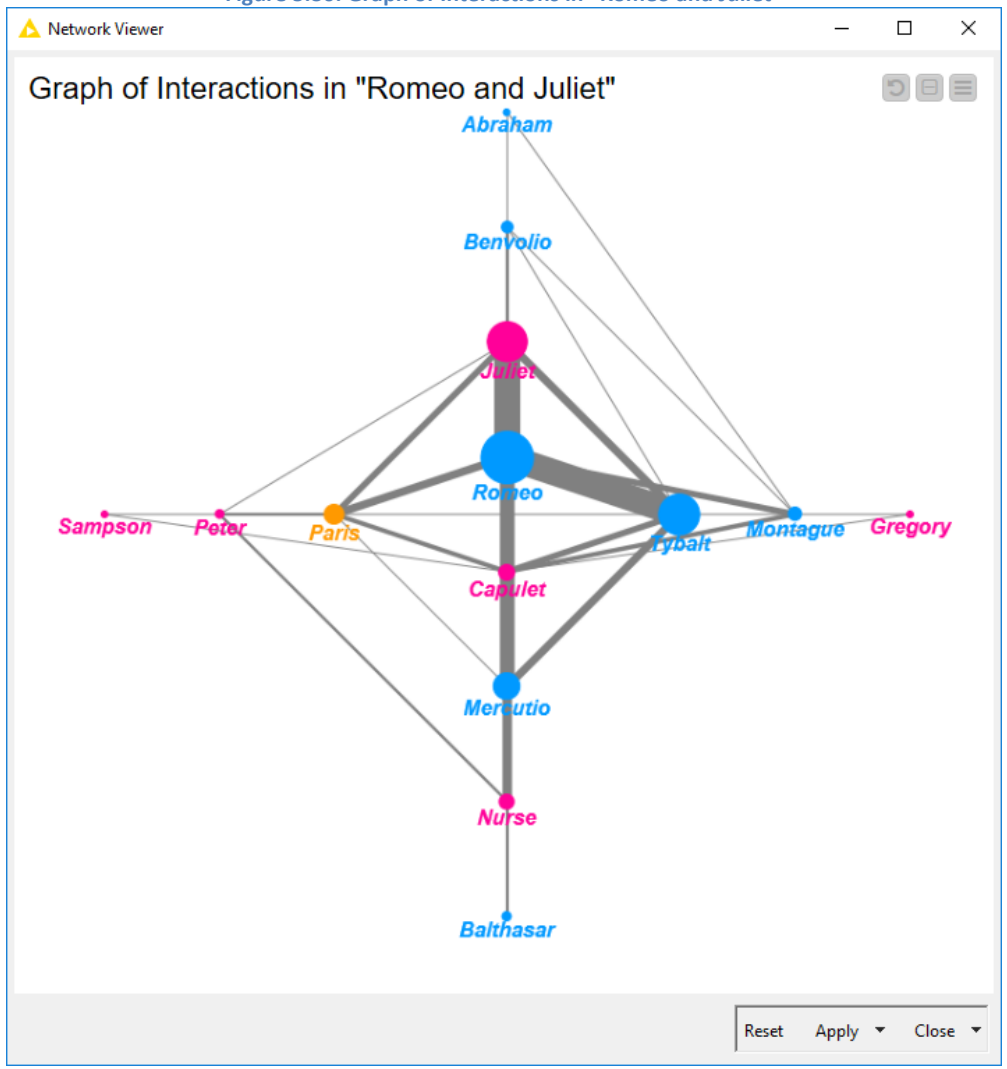


Figure 5.30. Graph of Interactions in “Romeo and Juliet”



Chapter 6. Topic Detection and Classification

6.1. Searching for Topics

The most common tasks in text mining are topic detection and sentiment analysis. Topic detection tries to identify the topic(s) of a post, comment, or even just a conversation. Sentiment analysis tries to identify the tone of the conversation, post, or comment, weather positive, negative, neutral, and all the shades in between. In this chapter, we will focus on topic detection.

Topic detection, like all other data analytics tasks, can follow two approaches: unsupervised and supervised. For unsupervised topic detection, the goal is to identify a pre-defined number of topics together with the words (or keywords) that describe them. For the supervised approach, we talk about topic classification. In this case, topic classes are defined and a training set with labelled texts is made available. The goal then becomes to predict which text can be associated to which topic class.

In the previous chapters, we have imported texts, cleaned them up from unimportant words and punctuation, enriched each word with additional information (tagging), removed inflections from the words, and finally mapped each text into its matrix of word's presence/absence or frequency. Once the text has been mapped into the document vector, we are now dealing with just a numerical data table, where each row is a number vector, possibly labelled with one of the pre-defined classes. This last transformation, from text to vector, was the keystone to move from text interpretation to text analytics. Now that a text is purely a vector of numbers, any machine learning and statistical algorithms can be applied for clustering and prediction.

6.2. Document Clustering

Let's start with Document clustering. The goal here is to group together Documents on the basis of some kind of distance / similarity measure and to represent each group (cluster) with the most representative words in that cluster.

There are many ways to define the distance, perform the grouping, and choose the representative words. In this section, we will explore general Machine Learning (ML) based clustering techniques and the text-specific Latent Dirichlet Association (LDA) algorithm. While ML based clustering techniques applies to all numerical data sets, LDA is expressly dedicated to text analytics.

Machine Learning Clustering Techniques

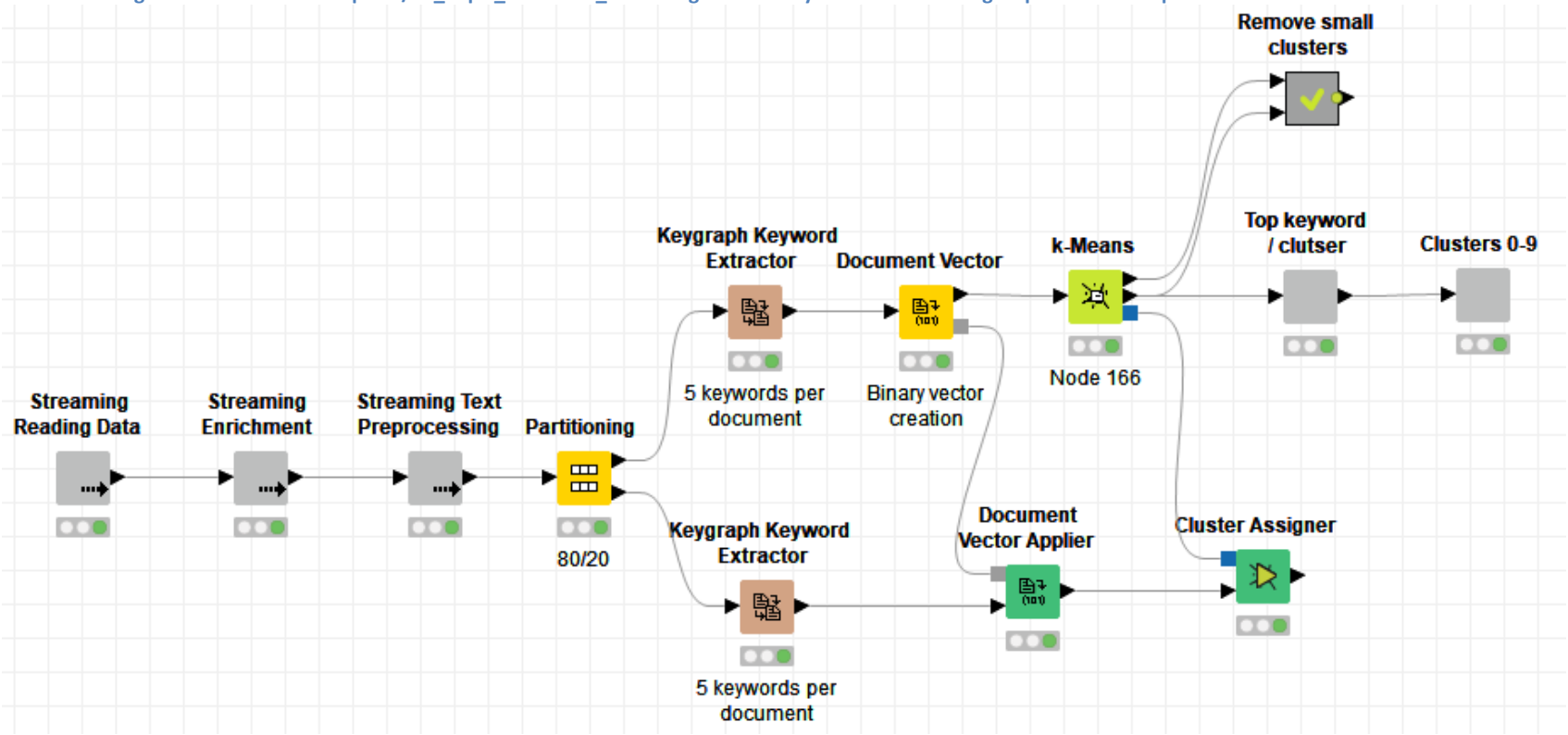
In the previous example workflows, we have imported texts, converted them into Document objects, enriched them using a Part Of Speech (POS) tagging algorithm for English, kept only nouns, adjectives, and verbs, cleaned up the text from numbers, punctuation signs, and stop words, and reduced

each word to its stem. The workflow prepared for this section, in Chapter6/01_Topic_Detection_Clustering (Fig. 6.1), all these pre-processing steps are implemented in the first three wrapped metanodes: Streaming Reading Data, Streaming Enrichment, and Streaming Text Processing. You might have noticed that all these metanodes are executed in streaming mode for faster execution.

Next, after partitioning the data set, we asked a Keygraph Keyword Extractor node to generate 5 keywords to represent each Document. All Documents are then transformed, with the Document Vector node, into vectors of 0/1s according to the absence/presence of a given keyword in the text.

At this point, we have a series of numerical data rows, each one representing a Document in the original data set. We will group them together into clusters using a many Machine Learning clustering technique. KNIME Analytics Platform offers a number of clustering algorithms: k-Means, hierarchical clustering, DBSCAN, and a few more. They are all located in Analytics/Mining/Clustering in Node Repository. In our example workflow, we have used the k-Means algorithm to cluster the training set in 10 clusters and the Cluster Assigner node to assign a cluster to each Document vector.

Figure 6.1. Workflow Chapter6/01_Topic_Detection_Clustering clusters keywords to discover groups of similar topics across a set of Documents.



The advantage of the k-Means node with respect to other clustering nodes consists of the prototype rows at its second output port. These prototype rows are generated as the average of all Document vectors in a cluster; that is, they average numbers between 0 and 1, measuring the presence of a given keyword in the cluster. What if, for each cluster, we displayed the keywords in a word cloud? Keywords could be sized by their average presence within the cluster, as calculated in the cluster prototype.

In the "Top keyword / cluster" wrapped metanode, the first step is to transpose the data table containing the prototypes, as to have the word vector for each cluster in a column. We then loop on all columns, i.e. on all clusters, one by one to extract the top keywords, i.e. the keywords with average presence higher than 0.005. Finally, for each cluster, the top keywords are displayed in a word cloud using the Tag Cloud (Javascript) node.

Not all clusters are created equal. Some of them cover many patterns, while some of them cover just a few. We could of course remove the least populated clusters, by counting the number of elements in each cluster, and removing the clusters with less than N elements, with N being an arbitrary number. This has been implemented in the metanode "Remove small clusters", but it has not been used for this example.

Note. We fed the k-Means node with binary vectors. If you use another non-normalized score in the Document vectors, you will need to normalize the vector set before passing it to the k-Means node.

The workflow is shown in figure 6.1 and the word clouds of the 10 clusters are displayed in figures 6.2 to 6.11.

Cluster 0 is the biggest cluster with 697 Documents and therefore with the most various questions and comments. The word "KNIME", "file", and "table" seem to be most frequently occurring. Such words are quite general, but this is not surprising since we are on the biggest cluster for the KNIME forum. It is hard to detect a single topic there. Probably, this cluster would benefit from a sub-clustering operation.

On the opposite, cluster 7 is the smallest cluster with only 4 Documents. From its word cloud it seems that all 4 Documents are concerned about loops and loop nodes.

Cluster 4 covers mostly Documents related to database operations, while Cluster 5 Documents on questions about accessing a database.

Cluster 3 has all questions about the basics of KNIME products: installation, download, redistribution, and licenses.

Cluster 1 collects questions on model evaluation and A/B testing.

Cluster 2 includes Documents with questions on node and workflow execution on the KNIME Analytics Platform; while Cluster 6 Documents about workflow execution on the KNIME Server.

Questions on reporting and data manipulation can be found respectively in cluster 8 and cluster 9.

Figure 6.2. Cluster 0 is the biggest cluster of all with 1151 Documents describing a variety of different problems. The associated topic could be “Generic Questions”. This cluster probably needs sub-clusters.

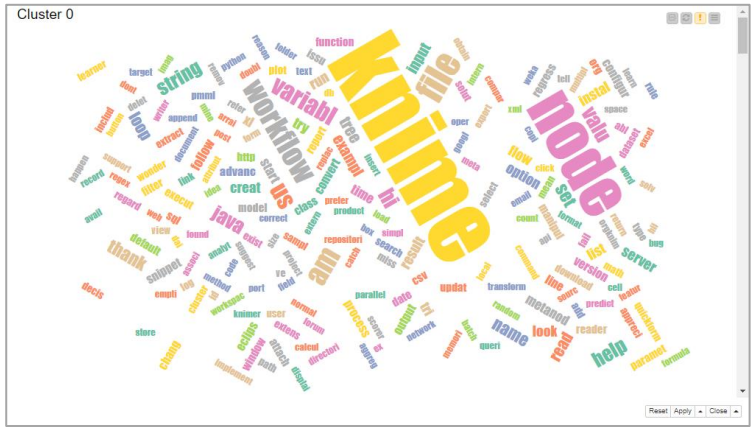


Figure 6.3. Cluster 1 collects 195 Documents including a number of row related questions, from filtering to loop. Topic could be: “Data Row Operations”.

Figure 6.4. Cluster 2 collects 153 Documents with column related questions. The topic could be “Data Column Operations”.

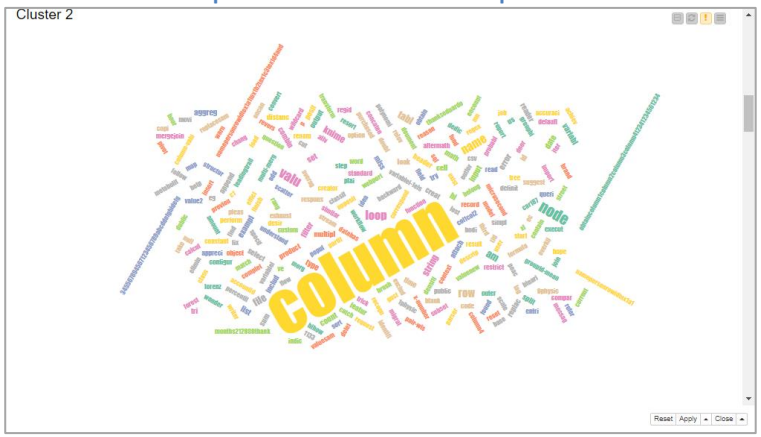


Figure 6.5. Cluster 3 collects 107 Documents reporting execution errors. Topic: “Execution Error”.

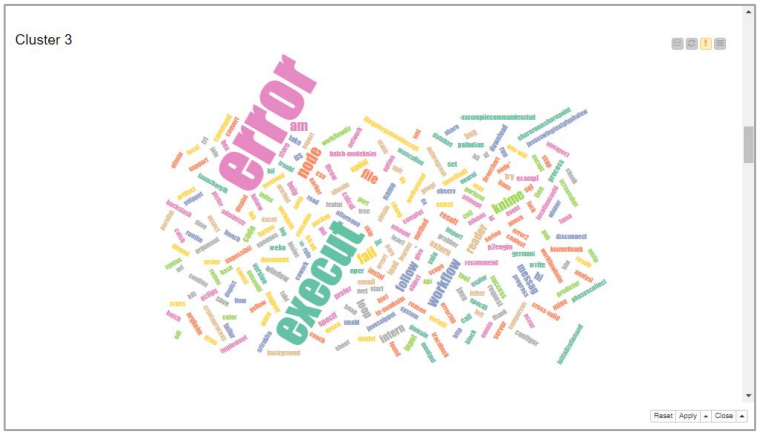


Figure 6.6. Cluster 4 collects only 10 Documents test questions. Topic could be: “Testing”.

Figure 6.7. Cluster 5 has 105 Documents mainly on data access. Topic: “Data Access”.

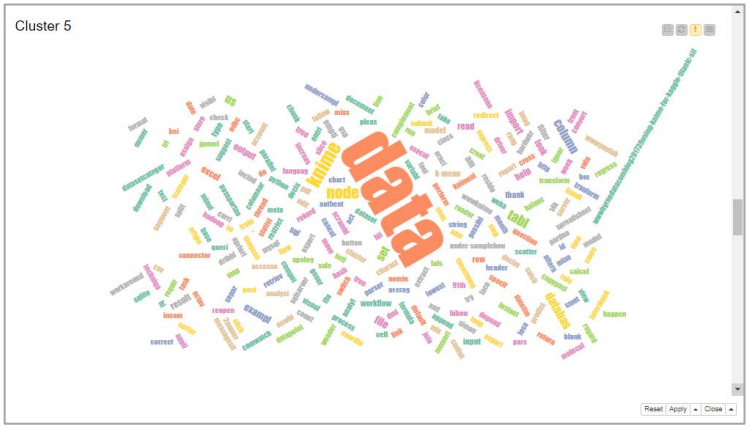


Figure 6.8. Cluster 6 collects 22 Documents with installation questions. Topic could be “Installation”.

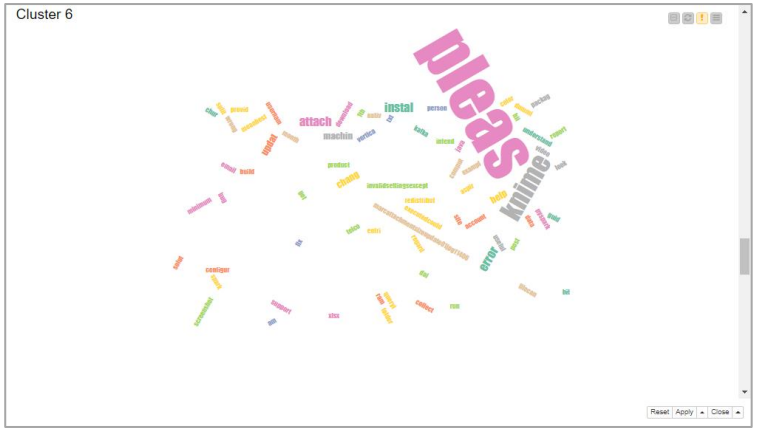


Figure 6.9. Cluster 7 covers 41 Documents asking about connections to specific databases. The associated topic could be “Database Connection”.

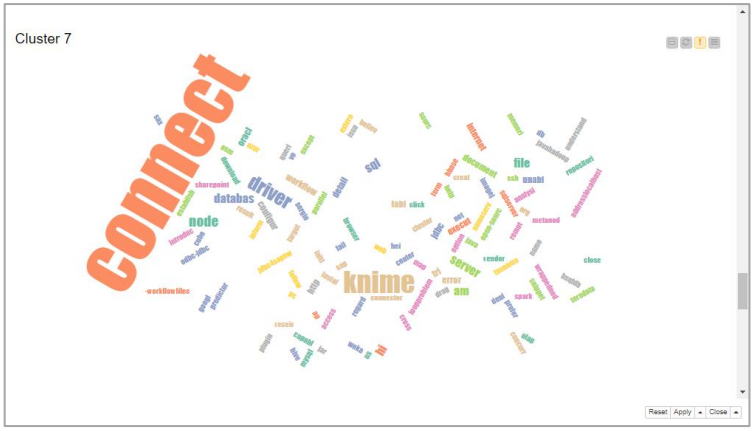
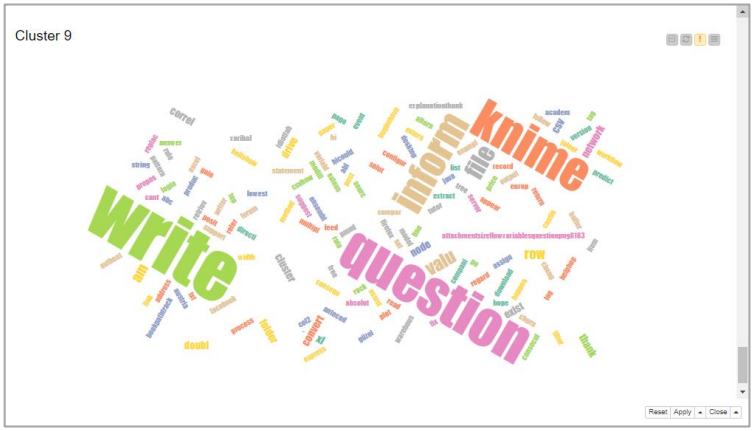


Figure 6.10. Cluster 8 collects 89 questions about database operations. Topic could be "Database Operations".

Figure 6.11. Cluster 9 contains 43 Documents about data and model writing. Topic: "Writing".



Latent Dirichlet Allocation (LDA)

General Machine Learning clustering techniques are not the only way to extract topics from a text data set. Text mining literature has proposed a number of statistical models, known as *probabilistic topic models,* to detect topics from an unlabeled set of Documents. One of the most popular model is the ***Latent Dirichlet Allocation***, or ***LDA***, developed by Blei, Ng, and Jordan [20].

LDA is a generative unsupervised probabilistic algorithm that finds out the top *K* topics in a dataset as described by the most relevant *N* keywords. In other words, the Documents in the data set are represented as random mixtures of *latent* topics, where each topic is characterized by a Dirichlet distribution over a fixed vocabulary. "*Latent"* means that we need to infer the topics, rather than directly observe them.

The algorithm is defined as generative model[3], which means that we make some *a priori* statistical assumptions [21], i.e:

- Word order in Documents is not important.
- Document order in the data set is not important.
- The number of topics has to be known in advance.
- The same word can belong to multiple topics.
- Each Document, in the total collection of *D* Documents, is seen as a mixture of *K* latent topics.
- Each topic has a multinomial distribution over a vocabulary of words *w*.

[3] A generative model describes how data is generated, in terms of a probabilistic model (https://www.ee.columbia.edu/~dpwe/e6820/lectures/L03-ml.pdf)

The generative process for LDA is given by:

$$\vartheta_j \sim D[\alpha],\ \Phi_k \sim D[\beta],\ z_{ij} \sim \vartheta_j,\ x_{ij} \sim \Phi_{zij}$$

where:

- ϑ_j is the mixing proportion of topics for Document *j* and it is modelled by a Dirichlet distribution with parameter α;
- a Dirichlet prior with parameter β is placed on the word-topic distributions Φ_k ;
- z_{ij}= *k* is the k topic drawn for the i^{th} word in Document *j* with probability $\vartheta_{k|j}$;
- word x_{ij} is drawn from topic z_{ij}, with x_{ij} taking on value *w* with probability $\Phi_{w|zij}$;

Let's see the learning phase (or posterior computation). We have selected a fixed number *K* of topics to discover and we want to learn the topic representation for each document and the words associated to each topic. Without going into much details, the algorithm operates as follows:

- At start, the algorithm randomly assigns each word in each Document to one of the *K* topics. This assignment produces an initial topic representation for all Documents and an initial word distribution for all K topics.
- Then, for each word *w* in Document *d* and for each topic *t*, compute:
    - *p(topic t | document d),* which is equal to the proportion of words in Document *d* currently assigned to topic *t*;
    - *p(word w | topic t)* which corresponds to the proportion of assignments to topic *t* over all Documents containing this word *w*.
- Reassign word *w* to a new topic *t*, based on probability *p(topic t | document d) * p(word w | topic t)*
- It then repeats the previous two steps iteratively until convergence.

There exist several variations and extensions of the LDA algorithm, some of them focusing on better algorithm performance. The simple parallel threaded LDA algorithm [21] followed by the Sparse LDA sampling scheme [22] is implemented in the *Topic Extractor (Parallel LDA)* node in the KNIME Text Processing extension.

Workflow Chapter6/Topic_Detection_LDA uses the Topic Extractor (Parallel LDA) node to discover topics in the KNIME Forum dataset. After data reading, pre-processing, stemming, stop word filtering, number filtering, etc., the *Topic Extractor (Parallel LDA)* node is applied, to extract 6 topics and 20 words to describe each one of them. At the end a tag cloud is created using the 20 words extracted for each topic. As a frequency measure for the word size, topic word occurrences are calculated in the wrapped metanode named "Term Count". Here, the *Dictionary Tagger* node tags only topic terms. All other terms, not related to topics, are filtered out by the *Modifiable Term Filter* node. After that, the BoW is created and the occurrences of each term are counted using a *GroupBy* node.

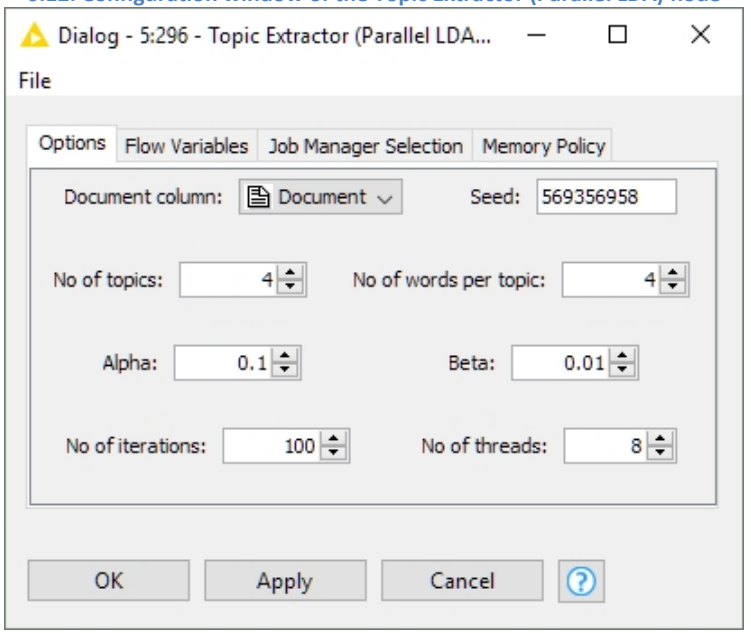
Topic Extractor (Parallel LDA)

LDA is a generative unsupervised probabilistic model that discovers the top *K* topics in a text data set and describes them with the most relevant *N* words. The node integrates "MALLET: A Machine Learning for Language Toolkit" topic modeling library.

Configuration settings require:

- *Document column* dropdown menu selects the input Document type column to use for topic detection.
- *Seed* text field defines the seed for random number drawing. Algorithm runs with different seeds produce different results.
- *Number of topics* sets the number K of topics to extract.
- *Number of words per topic* sets the number of representative words to describe the topics.
- *Alpha* and *Beta* sets the alpha and beta values in the a priori distributions. There is no theory-based method to choose the best alpha and beta values. Usually, the following relation is considered: $\alpha = 50/K$. Lower α values generate less topics for each Document, till the limiting case where α is zero. In this case each Document involves a single topic, which is randomly chosen from the set of all topics. Default value for β is 0.01.
- *No of Iterations* sets the maximum number of iterations for the algorithm to run
- *No of threads* sets the number of parallel threads to spawn

6.12. Configuration window of the Topic Extractor (Parallel LDA) node



A frequently asked question is “How many topics shall I extract to have a full minimal coverage of the data set?” As for all clustering optimization questions, there is no theory based answer. A few attempts have been proposed. Here we link to the KNIME blog post “Topic Extraction. Optimizing the Number of Topics with the Elbow Method” by K. Thiel and A. Dewi (2017).

Figure 6.13. Workflow Chapter6/Topic_Detection_LDA extracts topics from the KNIME Forum dataset using the Topic >Extractor (Parallel LDA) (Latent Dirichlet Allocation) node

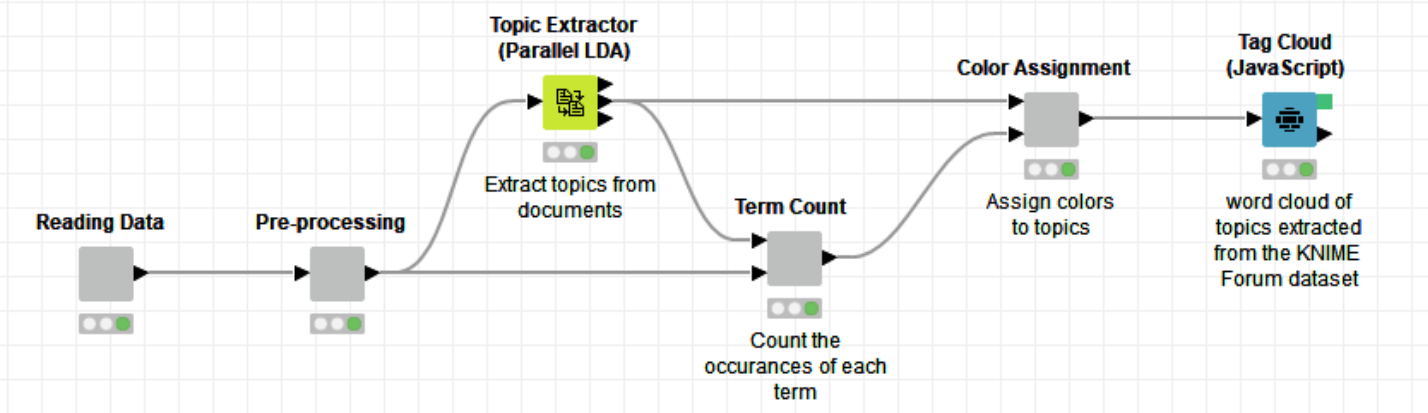
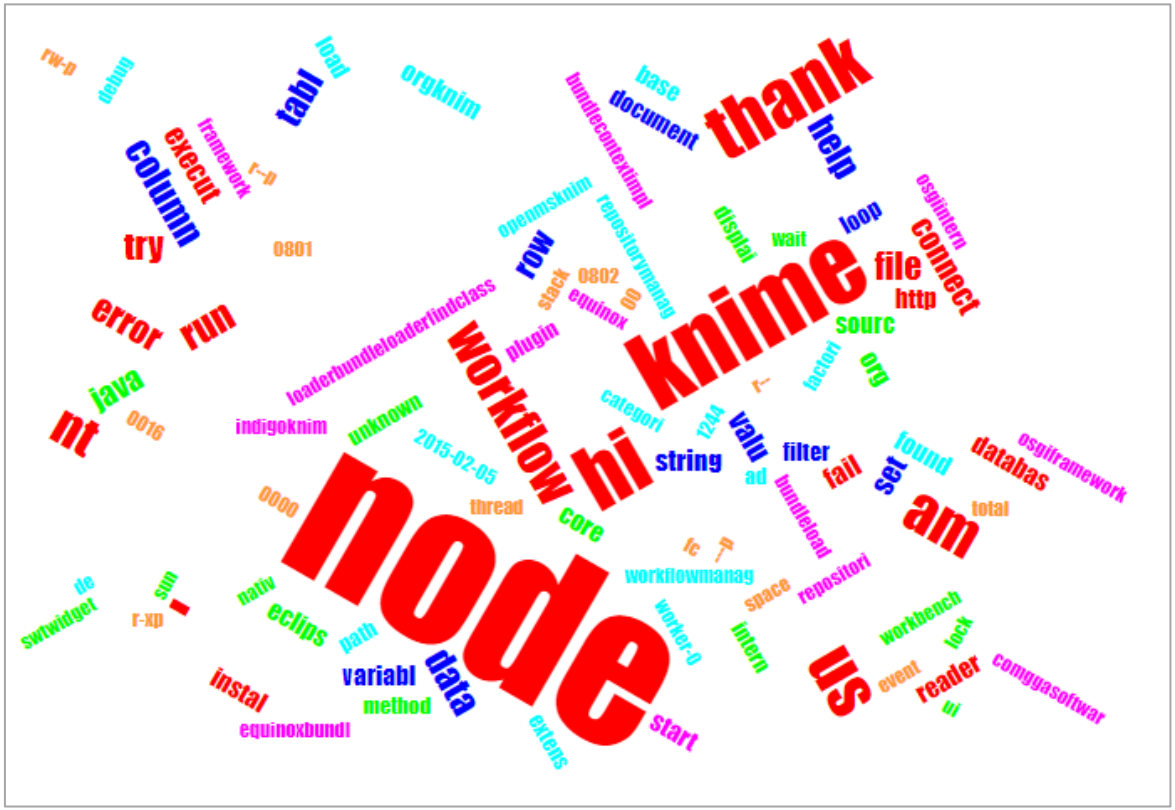


Figure 6.14. Word Cloud of topic words extracted from the KNIME Forum dataset by the LDA algorithm



6.3. Document Classification

Sometimes we are lucky and, instead of searching for topic related keywords randomly in the collection of Documents, we have a labelled data set. Here Documents have been labelled with classes and we can train a Machine Learning model to predict them. This is not the case of the KNIME Forum data set, where topic labels are not available. Thus, for this particular section, we will use another dataset, from file Thedata/Pubmed_Articles.csv.

File Pubmed_Articles.csv contains a number of scientific articles extracted from the PubMed library. Articles include: title, abstract, full text, author(s), category, and source. Source is always PubMed, as the origin of all articles in the dataset. The articles come from two queries to PubMed. The first query searched for articles related to human AIDS; the second query looked for articles on mouse cancer experiments. The articles then belong to two different sets: "Human AIDS" and "mouse cancer". Those are the Document classes and are stored in column "Category". Summarizing, we have a small dataset, with 300 scientific articles and a binary class. The articles are also equally distributed across the two classes: 150 articles on human AIDS and 150 on mouse cancer.

We start with reading the dataset in the file and converting the articles into Documents with a Strings to Document node inside the "Reading Data" metanode. The article class is stored in field "category" of the Document metadata. We then move to the Enrichment metanode/phase where we assign Part Of Speech tags to each word. In the next metanode, named Pre-processing, we keep only words with noun, verb, and adjective tags, clean up from stop words, punctuation, and numbers, and finally extract the stemming for each one of the remaining words. All these metanodes are implemented in the workflow named *03_Document_Classification* in folder Chapter6.

At this point we have 300 Documents, each one containing a PubMed article, ready for classification. We first create a training set with 80% of the articles and a test set with the remaining 20%, using the Partitioning node. A Keygraph Keyword Extractor node extracts 3 keywords to represent each Document. Now the classic procedure would be to apply a Document Vector node and transform my list of Documents into a matrix of word presence / absence or frequency. However, since our data set is very small, we should proceed with caution.

Note. Transforming a number of texts into word vectors might create a very high number of columns, which, in case of small datasets, might lead to overfitting and poor generalization performance.

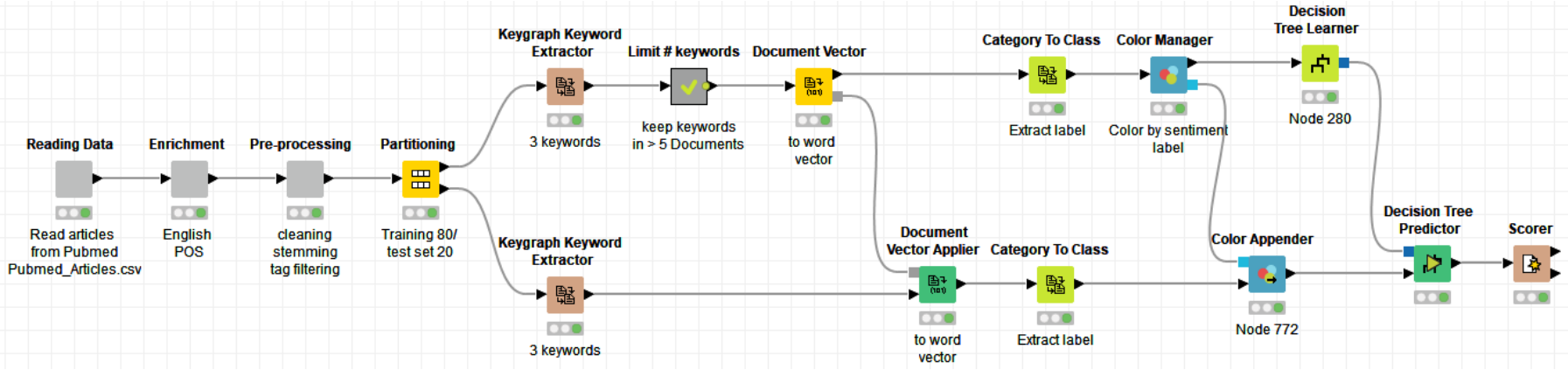
In our case, to make sure that overfitting does not happen, we artificially limit the number of produced columns by limiting the size of the keyword set. Indeed, we only accept keywords that appear in at least 5 of the Documents in the training set. After the execution of the Document Vector node, we get a data table with 201 rows (Documents in training set) and 25 columns (keywords).

During all those operations we have dragged along the Document column. Now it is the time to use it. We extract the class of each Document from the "category" field using the Category To Class node. Now we have a numerical data table, where each row has a class label. We can train any supervised Machine Learning (ML) algorithm. We choose a decision tree, since the dataset is small enough

On the test branch, we proceed with the same processing as for the training set, we apply the decision tree model, and then we measure the model performance (Accuracy ~72%) with the Scorer node.

There is nothing text processing specific in this topic classification approach. After the Document to Vector conversion, it becomes a classic classification problem, which can be solved with any supervised ML algorithm.

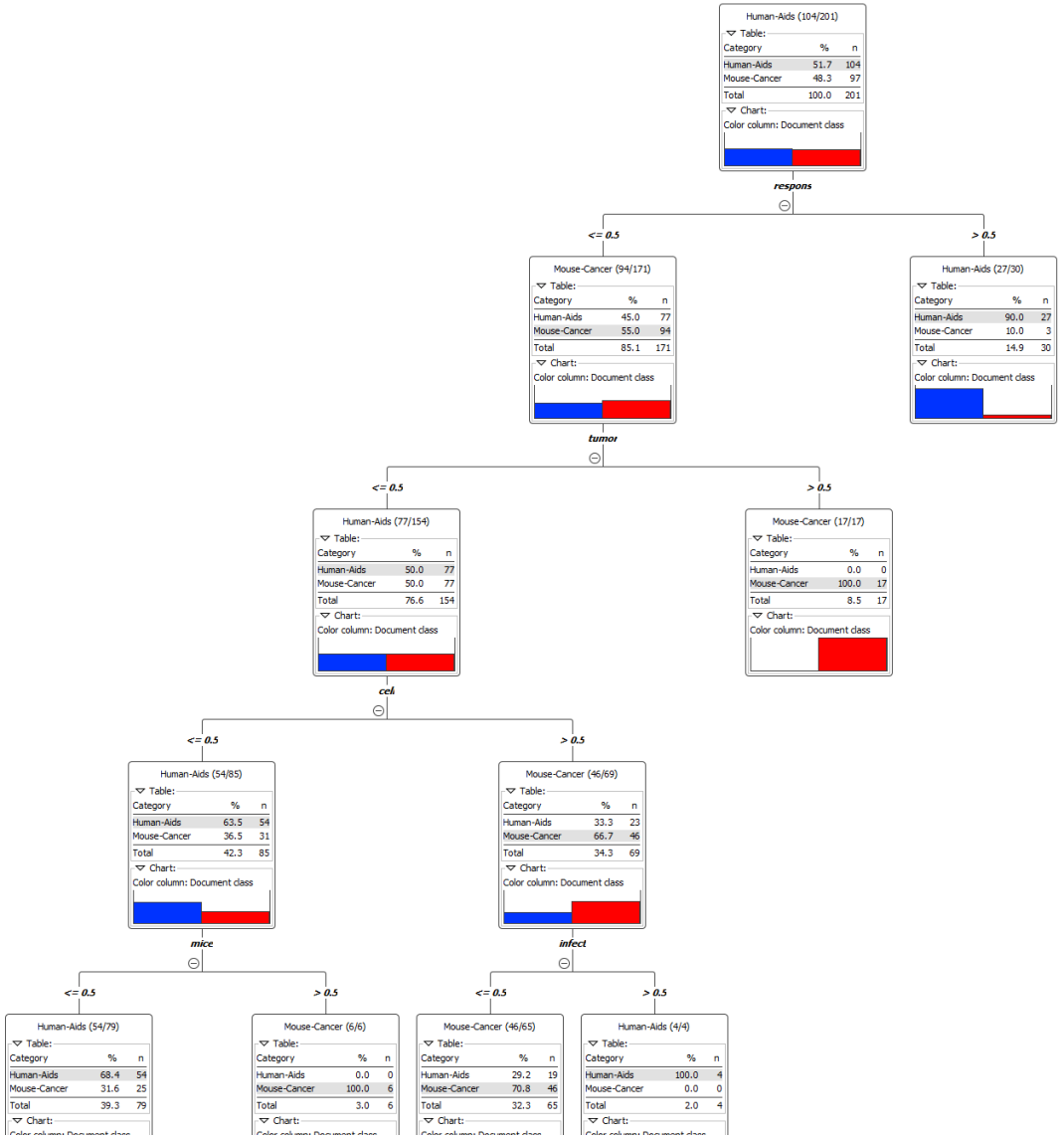
Figure 6.14. Workflow Chapter6/03_Document_Classification. This workflow trains a decision tree to recognize the topics of Documents.



The decision tree trained for topic classification, mouse cancer vs. human AIDS, is reported in figure 6.15. It is worth to notice that the tree structure is very vertical, with very few, if any, parallel branches. At each split, the decision tree identifies the most discriminative word, identifying one or the other topic. At the root node, the presence of word “response” identifies Human AIDS articles; in the next split, the presence of word “tumor” identifies mouse cancer related articles; later on “mice” identifies mouse cancer articles and “infect” human AIDS related articles; and so on.

It is also possible to use N-grams rather than uni-grams for topic classification. The process is the same, besides an N-grams extraction instead of a simple Bag of Words or keyword extraction. For a practical example, we advise you to read the blog post “Sentiment Analysis with N-grams” by K. Thiel (2016).

Figure 6.15. The decision tree trained to discover whether an article is about mouse cancer or human AIDS. Decision trees trained on word vectors usually show a very vertical organization, without many collateral branches. Indeed, the decision tree at each step tends to identify the discriminative words for a certain topic.



6.4. Neural Networks and Deep Learning for Text Classification

A new set of recently emerging techniques for text processing is based on neural networks, in its traditional instance or in its most recent deep learning variation. In this section, we will describe two approaches: one for word and Document embedding and one for topic/sentiment classification.

Word and Document Embedding

Word embedding, like Document embedding, belongs to the text pre-processing phase and precisely to that part that transforms a text into a row of numbers.

In the previous chapters we have seen that the Document Vector node transforms a sequence of words into a sequence of 0/1 – or frequency numbers – on the basis of the presence/absence of a certain word in the original text. This is also called "hot encoding". Hot encoding though has two big problems:

- it produces a very large data table with potentially a very big number of columns;
- it produces a very sparse data table with a very high number of 0s, which might be a problem to train certain Machine Learning algorithms.

The Word2Vec technique is then conceived with two goals in mind:

- reduce the size of the word encoding space (embedding);
- compress in the word representation the most informative description for each word.

Interpretability of the embedding results becomes secondary.

The Word2Vec technique is based on a feed-forward fully connected architecture [23] [24]. Let's start with a simple sentence, like "*the quick brown fox jumped over the lazy dog*" and let's consider the context word by word. For example, the word "*fox*" is surrounded by a number of other words; that is its context. If we use a forward context of size 3, then the word "*fox*" depends on context "*the quick brown*"; the word "*jumped*" on context "*quick brown fox*"; and so on. If we use a backward context of size 3, the word "*fox*" depends on context "*jumped over the*"; the word "*jumped*" on context "*over the lazy*"; and so on. If we use a central context of size 3, "*fox*" context is "*quick brown jumped*"; "*jumped*" context is "*brown fox over*"; and so on. The most commonly used context type is a forward context.

Given a context and a word related to that context, we face two possible problems:

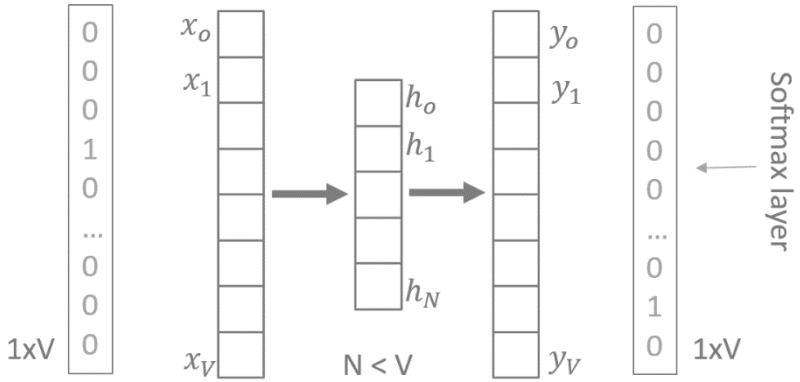
- from that context predict the related word (Continuous Bag Of Words or CBOW approach)
- from that word predict the context it came from (Skip-gram approach)

Let's limit ourselves for now to context size C=1. If we use a fully connected neural network, with one hidden layer, we end up with an architecture like the one in figure 6.16 for both approaches.

The input and the output patterns are hot encoded vectors, with dimension 1xV where V is the vocabulary size, representing words. In case of the CBOW strategy, the hot encoded context word feeds the input and the hot encoded related word is predicted at the output layer. In case of the Skip-gram strategy, the hot encoded missing word feeds the input, while the output layer tries to reproduce the hot encoded one word context. The number of hidden neurons is N, with N < V.

In order to guarantee a probability based representation of the output word, a softmax activation function is used in the output layer and the following error function E is adopted during training: $E = -log(p(w_o|w_i))$ where w_o is the output word and w_i is the input word. At the same time, to reduce computational effort, a linear activation function is used for the hidden neurons.

Figure 6.16. V-N-V neural architecture to predict a target word from a one-word context (CBOW) or a one-word context from a target word (Skip-gram). Softmax activation functions in the output layer guarantee a probability compatible representation. Linear activation functions for the hidden neurons simplify the training computations.



Notice that the input and output layers have both dimension 1xV, where V is the vocabulary size, since they both represent the hot encoding of a word. Notice also that the hidden layer has less units (N) than the input layer (V). So, if I represent the input word with the hidden neuron outputs rather than with the original hot encoding, I already reduce the size of the word vector, hopefully maintaining enough of the original information. The hidden neuron outputs provide the word embedding. This word representation, being much more compact than hot encoding, produces a much less sparse representation of the Document space.

If we move to a bigger context, for example with C=3, we need to slightly change the network structure. For the CBOW approach we need C input layers of size V to collect C hot encoded word vectors. The corresponding hidden layers then provide C word embeddings, each one of size N. In order to summarize the C embeddings, an intermediate layer is added to calculate the average value of the C embeddings (Fig. 6.17). The output layer tries to

produce the hot encoded representation of the target word, with the same activation functions and the same error function as for the V-N-V network architecture in figure 6.16.

Figure 6.17. CBOW neural architecture with context size C=3

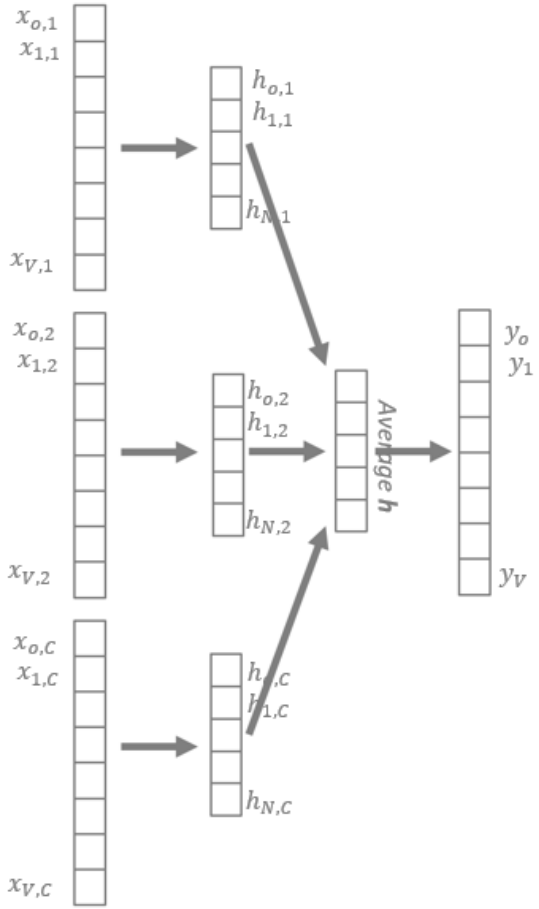
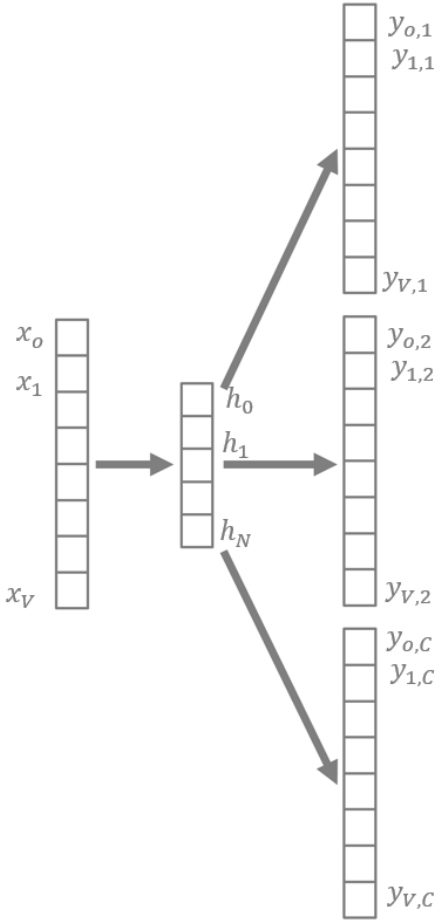


Figure 6.18. Skip-gram neural architecture with context size C=3



A similar architecture (Fig. 6.18) is created for the Skip-gram approach and context size C > 1. In this case, we use the weight matrix row enabled by the hot encoded input vector, to represent the input word. Even here the word representation dimensionality gets reduced from V to N and the Document space gets a more compact representation. This should help with upcoming Machine Learning algorithms, especially on large datasets and large vocabularies.

The vocabulary set can potentially become very large, with many words being used only a few times. Usually, a survival function examines all words in the vocabulary and decides which ones survive. The survival function assumes the following form: $P(w) = \left(\sqrt{\frac{z(w)}{s}} + 1\right)\frac{s}{z(w)}$ where w is the word, $z(w)$ its frequency in the training set, and s is a parameter called sampling rate. The smaller the sampling rate, the less likely a word will be kept. Usually, this survival function is used together with a hard coded threshold, to remove very infrequent words.

Sometimes, in addition to the classic positive examples in the training set, negative examples are provided. Negative examples are wrong inputs for which no outputs can be determined. Practically, negative examples should produce a hot encoded vector of all 0s. The literature reports the usage of negative examples to help, even though a clear explanation of why has not been yet provided.

In general, the CBOW approach is already an improvement with respect to the calculation of a co-occurrence matrix, since it requires less memory. However, training times for the CBOW approach can be very long. The Skip-gram approach takes advantage of the negative sampling and allows for the semantic diversification of a word.

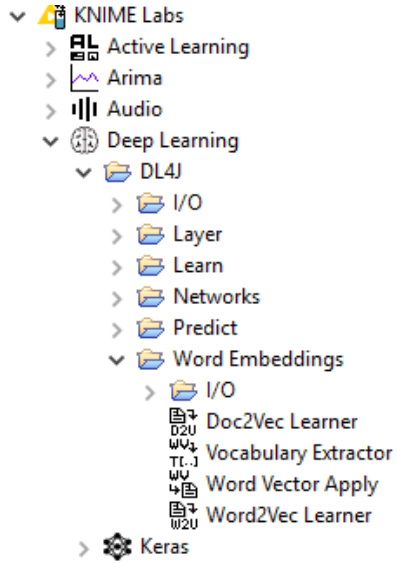
In KNIME Analytics Platform, the node Word2Vec Learner encapsulates the Word2Vec Java library from the DL4J integration. It trains a neural network with one of the architectures described above, to implement a CBOW or a Skip-gram approach. The neural network model is made available at the node output port.

The Vocabulary Extractor node runs the network on all vocabulary words learned during training and outputs their embedding vectors.

Finally, the Word Vector Apply node tokenizes all words in a Document and provides their embedding vector as generated by the Word2Vec neural network at its input port. The output is then a data table where words are represented as sequences of numbers and Documents are represented as sequences of words.

All these nodes are available in KNIME Labs/Deep Learning/DL4J/Word Embeddings in the Node Repository panel.

Figure 6.19. Nodes for Word Embedding from the DL4J integration

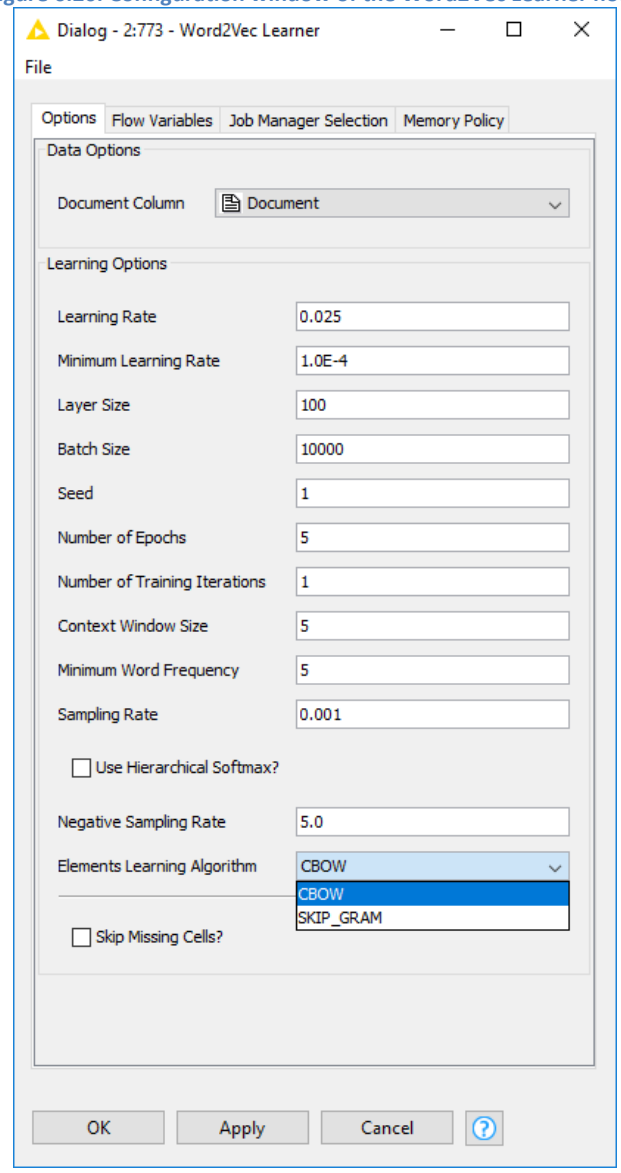


Word2Vec Learner

This node relies on the Word2Vec library of the DL4J integration. The node trains a neural network to implement word embedding following a CBOW or Skip-gram approach. Configuration settings then require:

- The Document type column with the examples to be used for training.
- The network uses an adaptive learning rate, therefore with a start value ("*Learning Rate*") and a minimum value ("*Minimum Learning Rate*")
- The number of hidden neurons to use for the word representation ("*Layer Size*"). Notice that this number will define the size of the word embedding.
- The training phase will run over separate batches of size K ("*Batch Size*"), M times on each batch ("*Number of Training Iterations*"), and N times ("*Number of Epochs*") on the whole training set
- "*Context Window Size*" defines the context size C
- "*Seed*" defines the starting point for the random generation of the initial weights of the neural network. This is necessary for repeatability.
- "*Sampling Rate*" and "*Minimum Word Frequency*" are used to trim the vocabulary. The word is removed if the word frequency is below "*Minimum Word Frequency*". "Sampling Rate" is a parameter in the word forgetting function. The smaller the sampling rate, the faster the word is forgotten from the vocabulary.
- It is possible to use negative examples in the training set. That is examples where no target word is provided (all 0s in the hot encoded output). The ratio of negative examples vs. positive examples in the training set is set through the "*Negative Sampling Rate*".
- The Strategy to learn: CBOW or Skip-gram ("*Elements Learning Algorithm*")
- Whether hierarchical softmax should be used instead of simple softmax (default). Hierarchical softmax can speed up the search for frequent words

Figure 6.20. Configuration window of the Word2Vec Learner node



Word Vector Apply

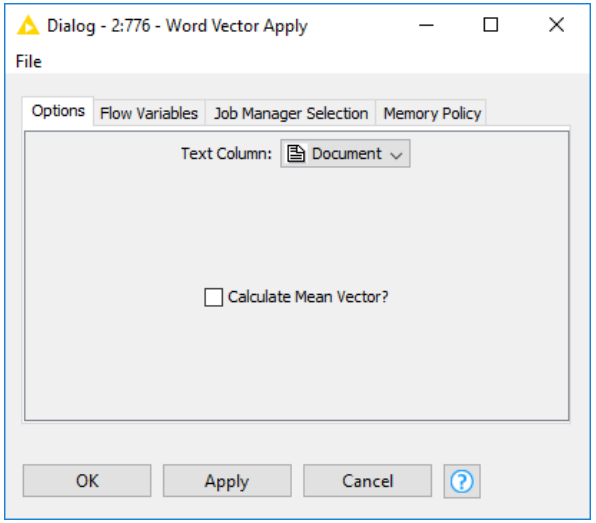
This node gets a number of Documents and a Word2Vec model at its input ports. After Document tokenization, it applies the Word2Vec model to each word and produces the word embedding for each word in the Document.

The result is a collection of words where each word is a collection of numbers, i.e. the word embedding. A Split Collection Column node or an Ungroup node might be necessary to extract the single elements of the collection.

All required settings are available in the Word2Vec model, including the training dictionary.

The only option in the configuration window is whether to calculate the mean vector of all word vectors in the Document; that is whether to calculate the center of the Document. Document centers can be used to define Document similarity via a distance measure.

Figure 6.21. Configuration window of the Word Vector Apply node



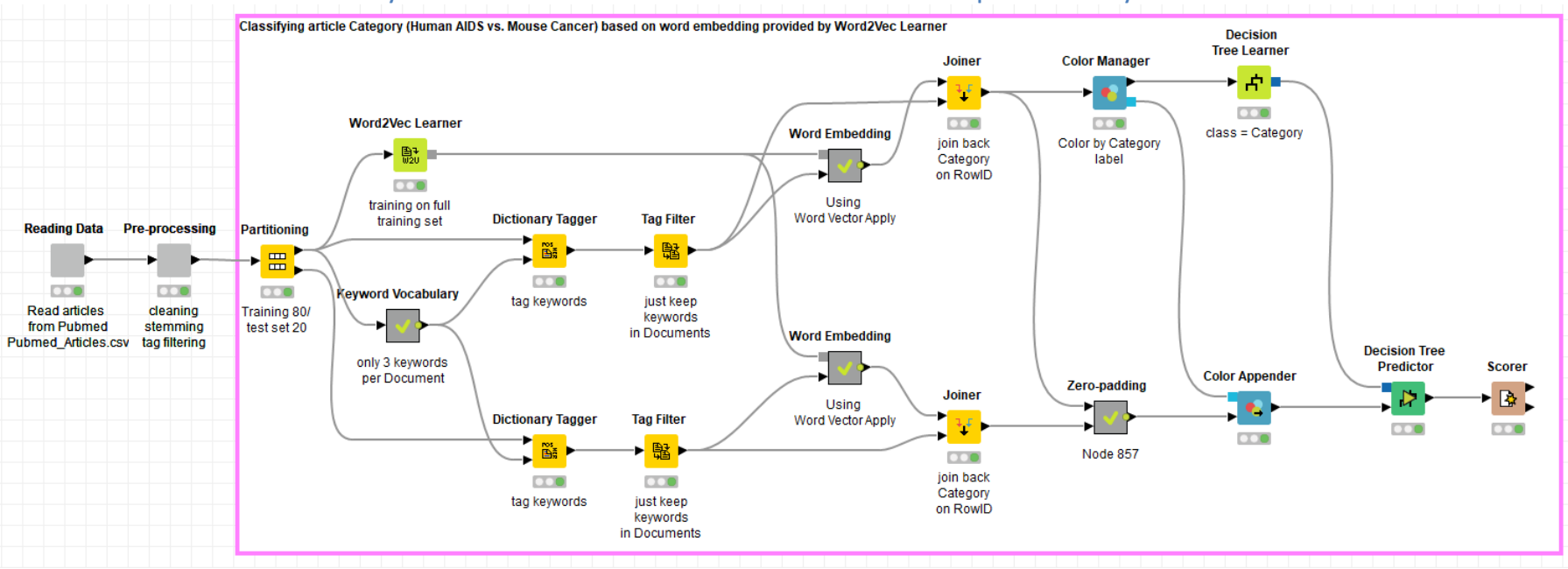
The first workflow we developed for this section is in Chapter6/04_Word_Embedding_Clax. This workflow trains a Word2Vec model. This model is then used to embed all words in the dictionary for the dataset in file Pubmed_Articles.csv. This data set includes articles from Pubmed database either on human AIDS or mouse cancer.

After some pre-processing involving data cleaning and stemming, articles are partitioned in a training set (80%) and a test set (20%). The Word2Vec model is trained on all words in the training set. Given the reduced size of the dataset, we used only 10 hidden neuron, which means size=10 for the embedding vectors. Three keywords are extracted to represent each Document, and through a Dictionary Tagger node and a Tag Filter node the Documents are reduced to just their three keywords.

The word embedding is then applied to the three keywords for each Document, in the training and in the test set, using a Word Vector Apply node in the “Word Embedding” metanode. At this point each Document is represented by a sequence of keywords and each keyword is represented by a sequence of numbers. We feed the training set sequences into a Decision Tree Learner node to train a decision tree, we apply the decision tree model to the test set, and finally we score the model performance with a Scorer node.

Remember that the word embedding is produced at the output port of the Word Vector Apply node in form of a cell collection. Inside the "Word Embedding" metanode, vectors are extracted using the Split Collection Column node.

Figure 6.22. This workflow (04_Word_Embedding_Clax) uses word embedding to train a decision tree to classify topics from scientific articles extracted from Pubmed. This is a binary class problem, since articles can either be on human AIDS or on mouse cancer. A Word2Vec model is trained on the training set word set. In order to make the classification class as similar as possible to the classification task implemented in figure 6.15, Documents are represented each through only 3 keywords. Word embedding vectors of the 3 keywords for each Document are then used to train a decision tree to predict the binary classes.



The approach is indeed similar to what developed in the previous section, as shown in the workflow in figure 6.14. Even there we used only 3 keywords to represent the Documents and a decision tree for classification. The only difference with the previous workflow consisted of the encoded representation of the Documents. Here we got accuracy ~81%, which is quite an improvement with respect to the previous accuracy ~72%. Of course, given the small size of the dataset, this comparison carries little meaning. Nevertheless, such a result is still encouraging.

With the Word Vector Apply node we have obtained a Document representation based on words and word embedding vectors. The final vectors still have a bigger size then the hot-encoded vectors. Would not it be better to use a Document embedding instead of a word embedding? To respond to this request, two additional neural architectures have been proposed. The inspiration of these new architectures comes of course from the networks designed to implement the CBOW and the Skip-gram strategy. The additional piece here is the Document ID.

Figure 6.23. Distributed Memory Paragraph Vector (DM). This network expands the Word2Vec concept by adding the paragraph (or Document) ID in the input layer. The hidden layer processing the Document ID produces the document/paragraph embedding.

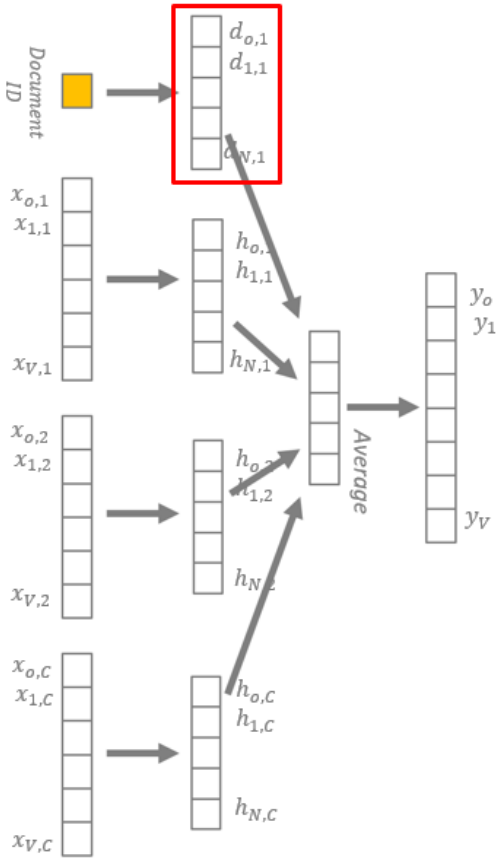
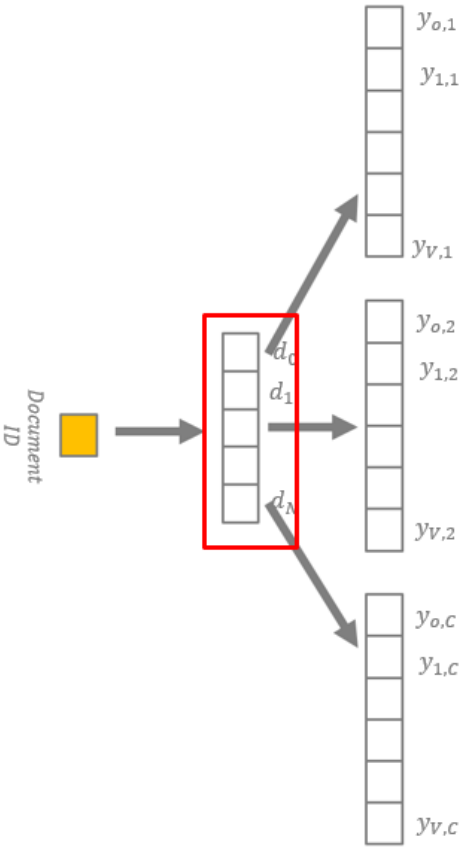


Figure 6.24. Distributed Bag Of Words Paragraph Vector (DBOW). This network predicts a sequence of words from a paragraph (or Document) ID. The hidden layer processing the Document ID produces the document/paragraph embedding.



In the CBOW architecture, we add the Document ID to the input words [25]. In the hidden layer then we have the word embedding vectors and one Document embedding vector. This new neural architecture is shown in figure 6.23, where the hidden layer responsible for the Document embedding is circled in red. This architecture is called Distributed Memory Paragraph Vector (DM).

An alternative architecture is shown in figure 6.24 [25]. Here the only input is the Document ID and the network is supposed to predict the sequence of associated context words. The hidden layer following the input layer with the Document ID is used to generate the Document embedding. This approach is called Distributed Bag of Words Paragraph Vector (DBOW). This second approach provides only Document embedding and no word embedding at all.

However, since word embedding can also be useful, the Deep Learning 4 J (DL4J) extension uses a hybrid approach, instead of using this algorithm pure, by including a few steps of the DM approach as well. The Doc2Vec Learner node trains a DM or a DBOW network.

Doc2Vec Learner

This node trains a Doc2Vec (DM or DBOW - Paragraph Vector) from a set of labelled Documents. Label can be a class or a Document ID, depending on whether we want a class or a Document embedding.

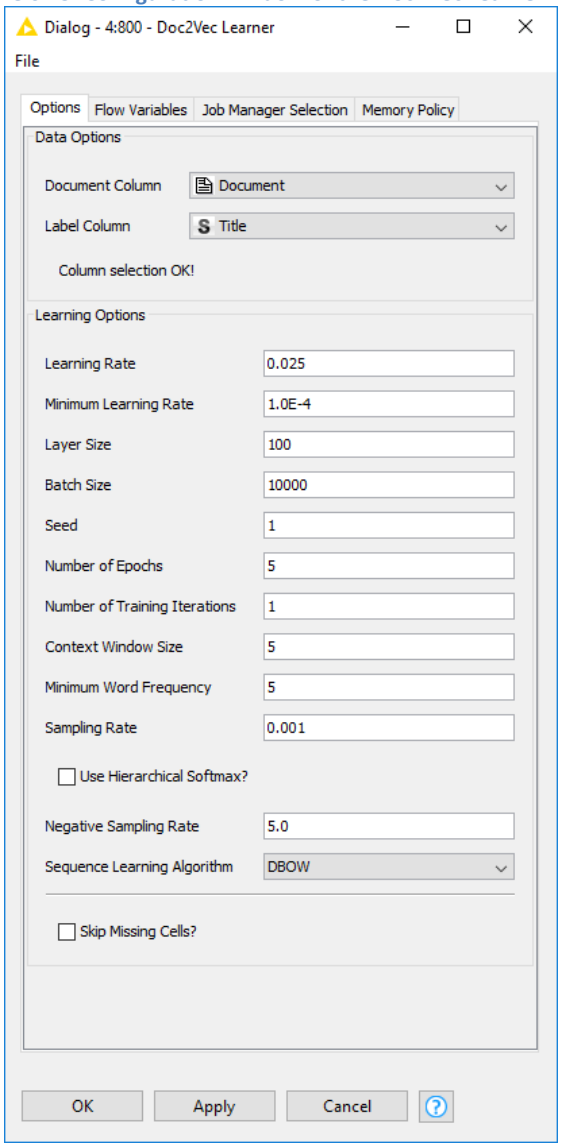
The Doc2Vec network is an extension of the Word2Vec network. Therefore the node configuration window is an extension of the node configuration window of the Word2Vec Learner node.

It contains exactly the same parameters for the network training (*Learning Rate* and *Minimum Learning Rate*, *Number of Epochs*), the network architecture (*Context Window Size*, *Layer Size*), parallel computation (*Batch Size*, *Number of Training Iterations*), word survival (*Minimum Word Frequency*, *Sampling Rate*), learning algorithm (*Sequence Learning Algorithm*), training set composition (*Negative Sampling Rate*).

The only additional parameter in the Doc2Vec learner configuration window is "*Label Column*" for the Document/class ID.

The output is a Doc2Vec model; that is a model that produces a Document embedding.

Figure 6.25. Configuration window of the Doc2Vec Learner node



The Vocabulary Extractor node, if applied to a Doc2Vec model, produces 2 outputs:

- At the top output port, the embedding vectors for all words in the model dictionary, learned during the model training.
- At the lower output port, the document embedding vectors.

Note. The Document vectors are learned with respect to a label, i.e. a Category ID. So, the embedding vectors will be the representative embedding vectors for the labelled classes.

If the label refers to a binary class problem, the embedding vectors, at the lower output port of the Vocabulary Extractor node, represent only 2 classes. If the label is actually the Document ID, i.e. each Document makes a class for itself, each Document will get its own embedding vector.

A Document Vector Apply node, as of KNIME Analytics Platform 3.5, is not yet available.

Vocabulary Extractor

This node extracts the words in the model vocabulary, passes them through the Word2Vec / Doc2Vec network at its input port, and produces the embedding for each one of them.

If the input model is a Doc2Vec network, the node also produces the embedding for the document classes / IDs specified in the training set.

The node has 2 output ports.

- The top output port contains the embedding vectors of the vocabulary words
- The lower output port produces the embedding vectors of the Document classes in the training set. This port is enabled only when a Doc2Vec model is connected to its input port. Otherwise it is disabled.

The whole required information, that is vocabulary words and neural network, is available in the model at the input port. So, no configuration settings are required for this node.

The whole intuition behind the Word2Vec approach consists of representing a word based on its context. This means that words appearing in similar contexts will be similarly embedded. This includes synonyms, opposites, and semantically equivalent concepts. In order to verify this intuition, we built a new workflow, named *05_Word_Embedding_Distance* available in folder Chapter6. Here we extract all words from the model dictionary and we expose their embedding vectors with a Vocabulary Extractor node. Then we calculate the distances among word pairs.

Figure 6.26. Chapter 6/05_Word_Embedding_Distance workflow. This workflow analyzes the embedding vectors of all words in the dictionary produced by a Word2Vec model trained on the dataset in the Pubmed_articles.csv file. These Pubmed articles deal either with human AIDS or with mouse cancer. Distances between embedding vectors of word pairs are also calculated.

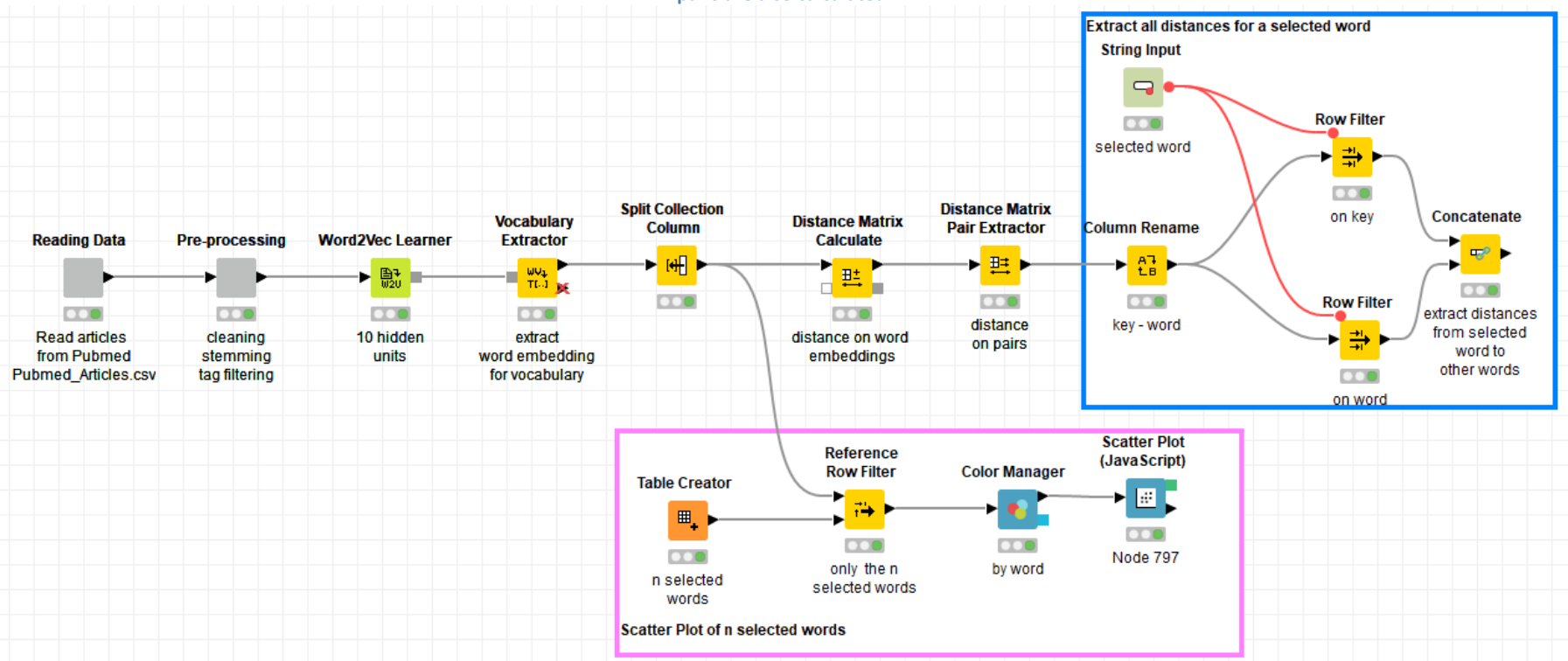


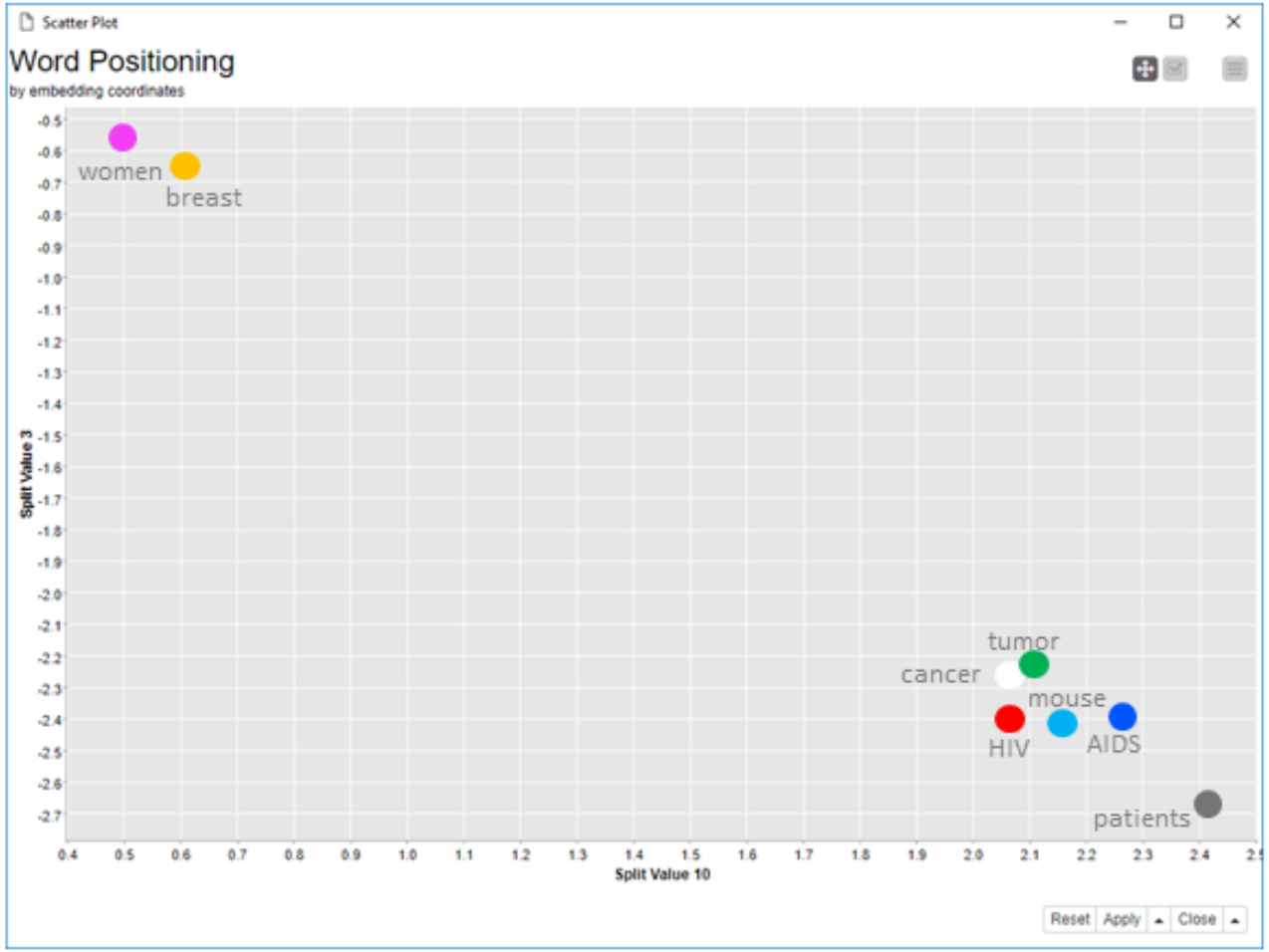
Figure 6.27 shows the positioning of some of the dictionary words in the embedding space using an interactive scatter plot. The interactive scatter plot allowed for a drill-down investigation across all pairs of embedding coordinates. For the screenshot below, we chose coordinate #3 and coordinate #10.

In the embedding coordinate plot, “cancer” and “tumor” are very close, showing that they are often used as synonyms. Similarly “AIDS” and “HIV” are also very close, as it was to be expected. Notice that “mouse” is in between “AIDS”, “cancer”, “tumor”, and “HIV”. This is probably because most of the articles in the data set describe mice related findings for cancer and AIDS. The word “patients” while still close to the diseases it is actually further away than the word “mouse”. Finally, “women” are on the opposite side of the plot, close to the word “breast”, which is also plausible. From this small plot and small dataset, the adoption of word embedding seems to make sense.

Note. All disease related words are very close to each other, like for example “HIV” and “cancer”. Even if the words refer to different diseases and different concepts, “HIV” and “cancer” are still the centers of most of the articles in the dataset. This means that from the point of view of the semantic role, they could be considered equivalent.

Inspecting the word pairs with smallest distance, we find that “condition” and “transition” as well as “approximately” and “determined” are the closest words. Similarly, unrelated words - like “sciences” and “populations”, “benefit” and “wide”, “repertoire” and “enrolled”, “rejection” and “reported”, “Cryptococcus” and “academy” - are located very closely in the embedding space.

Figure 6.27. Scatter Plot of Word Embedding Coordinates (coordinate #3 vs. coordinate #10). You can see that semantically related words are close to each other.

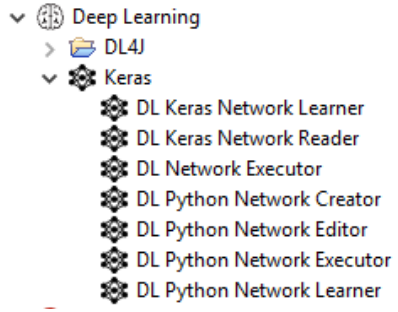


Sometimes, instead of classifying a text into classes, it might be of interest to form a sentence by predicting the next word from the past words. In this case, we are working with a time series made of words rather than of numbers. For this particular task, prediction algorithms able to capture the hidden dynamic of the sequence are to be preferred. So far, Hidden Markov Models have been used for this kind of dynamic prediction problems. However, recently, attempts have been made using Recurrent Neural Networks (RNN). Recurrent Neural Networks allow self- and backward connections within the neural architecture. This allows to show a dynamic temporal behavior and makes these networks particularly fit to process sequences, like for example word sequences. RNNs, however, are very complex to train, due to instability of the temporal behavior. One simplified version of RNNs, recently introduced, are the Long Short-Term Memory networks (LSTM).

In Long short-term memory (LSTM) networks are RNNs built using special neurons, named Long Short-Term Memory (LSTM) units. An LSTM unit consists of a *memory cell,* an *input gate,* an *output gate,* and a *forget gate*. The memory cell is responsible for "remembering" previous values over a time interval. Each of the three *gates* are classic artificial neurons, computing an activation function of the inputs weighted sum. The input gate fires up only for sufficiently large input sum; the forget gate takes care of forgetting rare inputs; and the output gate combines memory output, forget output and input output to produce the total neuron output. The memory and forget components allow the network to exploit the input sequence hidden dynamic and to produce the right prediction based on the input history.

There is no node to train an SLTM network in KNIME Analytics Platform (so far). However, we could do that via the KNIME integration with Python Keras.

Figure 6.28. Nodes for Keras integration in KNIME Analytics Platform



To run the Keras nodes, you will need to install Python 3.0 (see Instructions), Anaconda, numpy, and a few additional components as described in the "KNIME Deep Learning – Keras Integration" page.

6.5. Exercises

Exercise 1

Extract 4 topics from articles in file Thedata/Pubmed_Articles.csv, using the LDA algorithm, and describe each topic with 10 keywords. Draw a word cloud on the discovered 40 keywords, with color by topic and size by LDA weight. Are the topics uncovering mouse cancer and AIDS related articles?

Solution

Text Pre-processing includes: Strings to Document conversion; keeping only nouns, verbs, and adjectives; punctuation erasure, case conversion, stop word filtering, and stemming. Then the LDA algorithm is applied via the Topic Extractor (Parallel LDA) node. Resulting keywords are colored by topic cluster, sized by LDA weight, and displayed in the word cloud in figure 6.30.

Figure 6.29. Solution workflow of this exercise available under Chapter6/Exercises/ Exercise 1. LDA on Mouse Cancer vs. Human AIDS Documents.

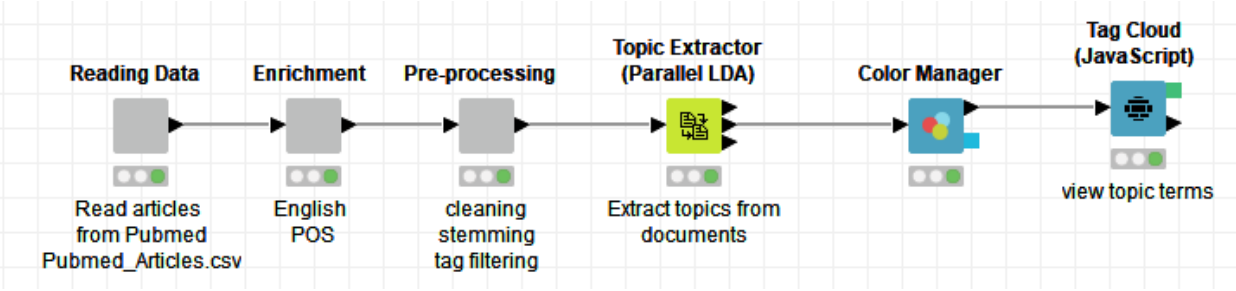
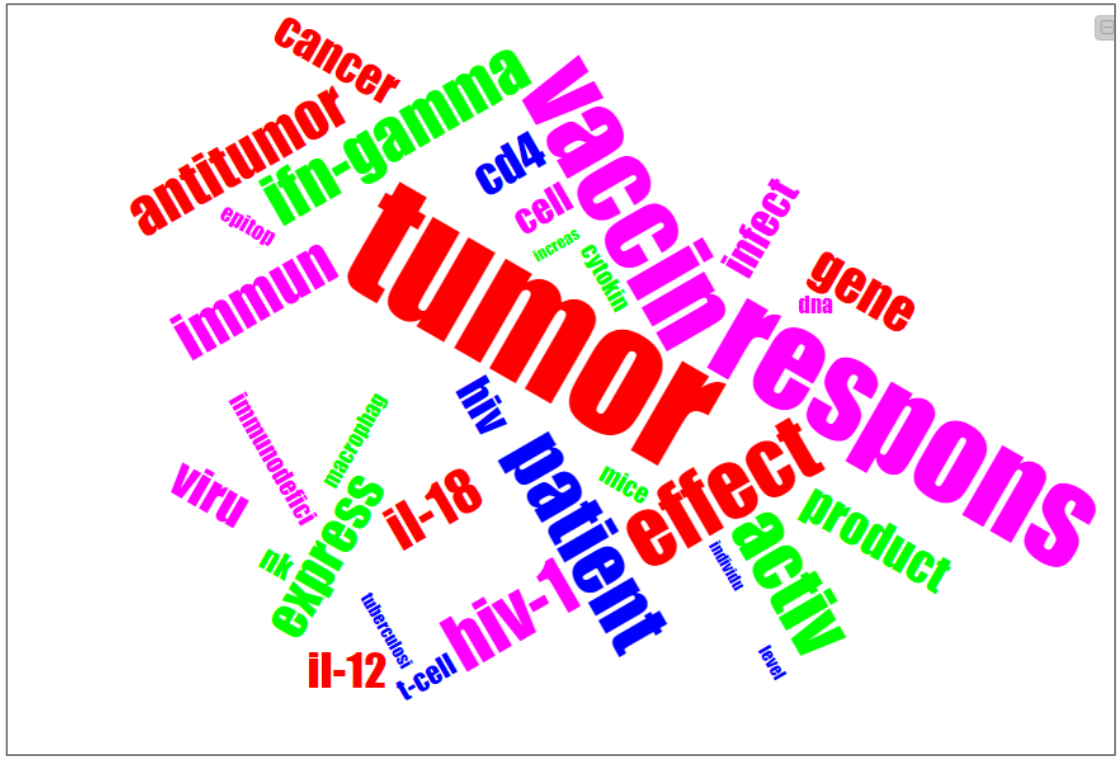


Figure 6.30. Word cloud of keywords identified by LDA algorithm running on Pubmed articles to detect 4 topics. Keywords are colored by topic cluster. In pink and blue you can see keywords describing AIDS related articles. In red and green keywords about mouse cancer.



Chapter 7. Sentiment Analysis

7.1. A Measure of Sentiment?

A relatively more recent trend in the analysis of texts goes beyond topic detection and tries to identify the emotion behind a text. This is called sentiment analysis, or also opinion mining and emotion AI.

For example, the sentence *"I love chocolate"* is very positive with regards to chocolate as food. *"I hate this new phone"* also gives a clear indication of the customer preferences about the product. In these two particular cases, the words "*love*" and "*hate*" carry a clear sentiment polarity. A more complex case could be the sentence "*I do not like the new phone*", where the positive polarity of "*like*" is reversed into a negative polarity by the negation. The same for "*I do not dislike chocolate*", where the negation of a negative word like "dislike" brings a positive sentence.

Sometimes the polarity of a word is context dependent. "*These mushrooms are edible*" is a positive sentence with regards to health. However, "*This steak is edible*" is a negative sentence with regards to a restaurant. Sometimes the polarity of a word is delimited in time, like "*I like to travel, sometimes.*" where sometimes limits the positive polarity of the word "*like*". And so on to even more subtle examples like "*I do not think this chocolate is really great*" or even worse "*Do you really think this concert was so fantastic?*".

We have talked here about positive and negative sentiment. However, POSITIVE and NEGATIVE are not the only labels you can use to define the sentiment in a sentence. Usually the whole range VERY NEGATIVE, NEGATIVE, NEUTRAL, POSITIVE, and VERY POSITIVE is used. Sometimes, however, additional less obvious labels are also used, like IRONY, UNDERSTATEMENT, UNCERTAINTY, etc …

Sentiment analysis is widely applied in voice of the customer (VOC) applications. For example, when analyzing responses in a questionnaire or free comments in a review, it is extremely useful to know the emotion behind them, in addition to the topic. A disgruntled customer will be handled in a different way from an enthusiastic advocate. From the VOC domain, the step to applications for healthcare patients or for political polls is quite short.

How can we extract sentiment from a text? Sometimes even humans are not that sure of the real emotion in between the lines. Even if we manage to extract the feature associated with sentiment, how can we measure it? There are a number of approaches to do that, involving natural language processing, computational linguistics, and finally text mining. We will concern ourselves here with the text mining approaches, which are mainly two: a Machine Learning approach and a lexicon based approach.

7.2. Machine Learning Approach

The Machine Learning approach relies on a labelled data set. That is on a data set containing texts whose sentiment has already been evaluated, usually by a human. It then relies on the training of a supervised Machine Learning model to associate such labels to new similar texts.

In this section, we use the IMDB dataset, where movie reviews have been manually labelled as positive or negative. The goal here is to train a model to predict the sentiment of a movie review based on its words.

This approach follows the classic Machine Learning process. First the data set is acquired, pre-processed, and partitioned in training set and test set. Pre-processing means mainly text cleaning and word stemming. In order to evaluate sentiment, a wide range of grammar entities, such as nouns, verbs, adjectives, adverbs etc … , is required. Indeed, an adverb can change the whole sentiment of the text. For this reason, Part Of Speech tagging and filtering have not been used here. We assumed that prepositions and conjunctions were already removed during the text cleaning part.

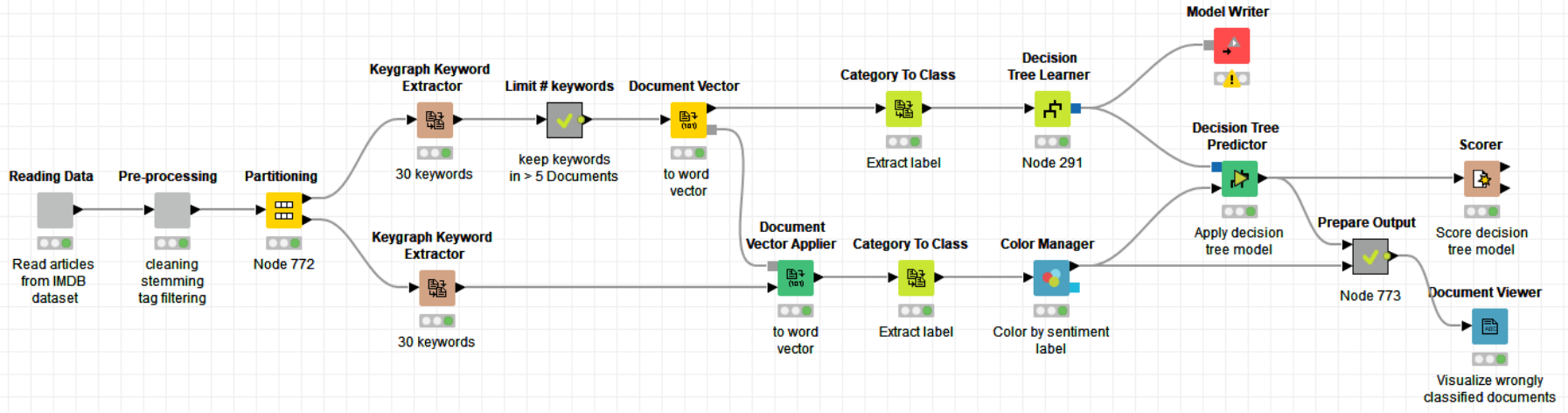
Next, a word vector is created for each Document. Here we used 30 keywords for each Document from a Keygraph Keyword Extractor node, but we could have also used all words in the Document from a Bag of Words node. The decision of using only 30 keywords was made to accelerate the workflow execution speed.

On one side, a higher number of keywords or words might lead to better performances in sentiment classification. On the other side, too many columns on too little data might lead to overfitting. In order to make sure that not too many columns are generated in the word vector, we exclude all keywords appearing in less than 5 Documents in the training set. At the end of this phase, the label for the sentiment class is extracted from the Document and appended to the corresponding word vector with a Category To Class node.

Next, a machine learning model is trained to predict the sentiment labels, positive or negative, associated with each Document. The metanode with name “Prepare Output” extracts all wrongly classified Documents. They can be visualized later with a Document Viewer node. The Scorer node measures Accuracy at ~75%.

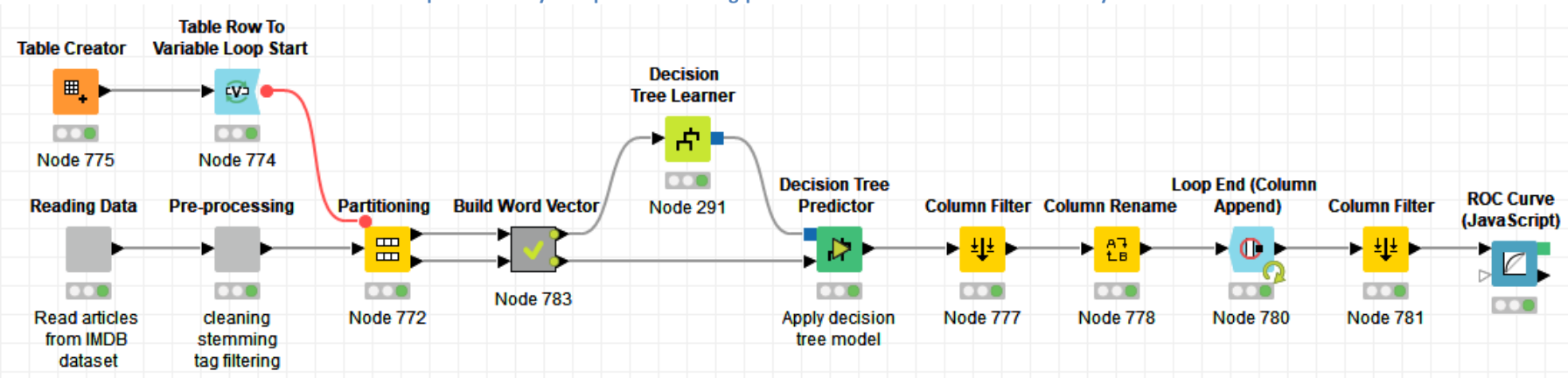
This workflow is located in *Chapter7/01_Sentiment_Analysis_ML_Approach*.

Figure 7.1. Workflow *Chapter7/01_Sentiment_Analysis_ML_Approach* to classify sentiment in reviews in the IMDB dataset.



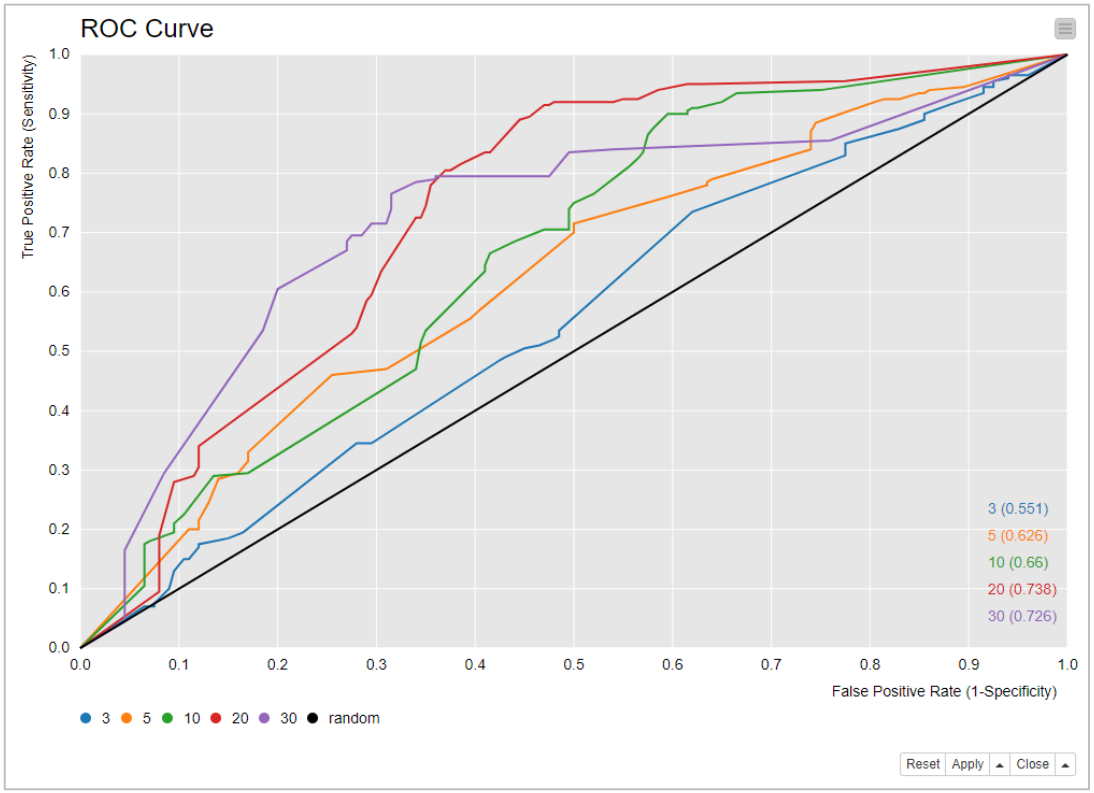
Probably the accuracy depends on the number of extracted keywords. Shall we verify this? The second workflow of this chapter, named *02_Num_Keywords_Sentiment_Analysis_ML* (Fig. 7.2), wraps the central part of the previous workflow (Fig. 7.1) – from partitioning to sentiment prediction – into a loop, where different values for the number of extracted keywords are explored. The final performances for sentiment predictions are then compared through an ROC curve (Fig. 7.3). The Partitioning node samples the input dataset linearly, to produce the same training set and test set for all loop iterations.

Figure 7.2. Workflow *Chapter7/02_Num_Keywords_Sentiment_Analysis_ML*. This workflow trains a model on 3, 5, 10, 20, 30 keywords per Document, to predict its sentiment. The loop is necessary to repeat the training-prediction on the different number of keywords.



The ROC curves from the sentiment prediction sets created using different numbers of keywords are displayed in figure 7.3. 3 keywords are clearly insufficient to represent the text of most movie reviews. The Area under the Curve (AuC) in this case is barely above 50%, which corresponds to a random choice. Performances of course improve with the increasing of the number of keywords used to represent a Document. With 20 keywords, the AuC measures already raises beyond 70%.

Figure 7.3. ROC curves for different sentiment prediction models, based on different number of text keywords: 3, 5, 10, 20, 30.



Another approach uses N-grams rather than uni-grams for sentiment classification. This approach is described in the KNIME blog post "Sentiment Analysis with N-grams" by K. Thiel (2016).

7.3. Lexicon-based Approach

Another approach to sentiment analysis relies on the usage of specific dictionaries. The idea is to measure the positivity or negativity – that is the polarity - of a Document according to the number of positive and negative words in it. For this approach we need a lexicon with the word polarity. Such

lexicon are of course language dependent and are usually available at linguistic department at national universities. Here is a list of possible sources for some languages.

- English http://mpqa.cs.pitt.edu/
- Chinese http://compling.hss.ntu.edu.sg/omw/
- Farsi http://compling.hss.ntu.edu.sg/omw/
- Arabic http://compling.hss.ntu.edu.sg/omw/
- Thai http://compling.hss.ntu.edu.sg/omw/
- Italian https://github.com/clips/pattern/tree/master/pattern/text/it
- Russian: http://wordnet.ru/

For the movie review IMDB dataset, we used English dictionaries from the MPQA corpus, where each word is associated to a polarity: positive, negative, or neutral.

After reading the data and proceeding with the usual text cleanup - case conversion, punctuation erasure, number removal, stop word filtering, and word stemming – two lists of words are imported from the MPQA Corpus: the list of negative words and the list of positive words. Each word in each text is matched against the two lists and assigned a sentiment tag consequently, through a Dictionary Tagger node. After that, terms with attached polarity are extracted from Documents using a Bag of Word Creator node. Positive words are counted. Negative words are counted. The Document sentiment score is finally calculated, in the metanode "Calculate Score", as the difference between the number of positive words and the number of negative words, relatively to the total number of words in the Document.

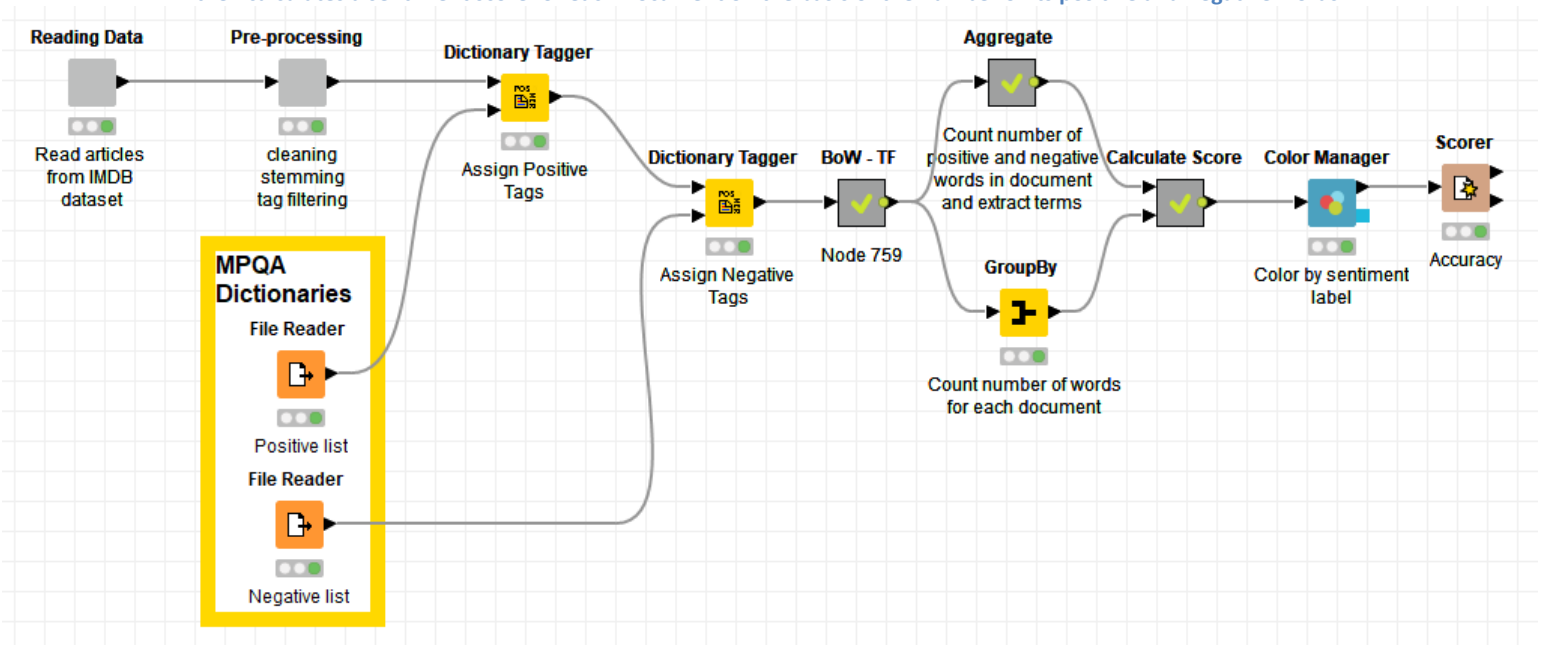
$$sentiment\ score = \frac{\#\ positive\ words - \#\ negative\ words}{\#total\ words}$$

A Rule Engine node decides whether a Document has positive or negative sentiment, based on the sign of the sentiment score.

The workflow used to implement this second approach is in *Chapter7/03_Sentiment_Analysis_Lexicon_Based_Approach* and reported in figure 7.4. The final accuracy on the movie review IMDB dataset is ~65%. Indeed it is lower than what achieved in the previous section. However, the approach is easier to implement, does not need sentiment labels, and it is easily customizable.

Indeed, relying on different dictionaries, for example listing not only positive/negative words but also serious/ironic words, we can detect different kinds of sentiments for the same Document.

Figure 7.4. Workflow *Chapter7/03_Sentiment_Analysis_Lexicon_Based_Approach*. This workflow tags words in a Document as negative or positive based on dictionary entries. It then calculates a sentiment score for each Document on the basis of the number of its positive and negative words.



7.4. Exercises

Exercise 1

On the KNIME Forum dataset (file Thedata/ForumData_2013-2017.table) calculate the sentiment score for each Document, using the Lexicon based approach and the MPQA Corpus files available in folder Thedata.

Then show the Documents as points in a scatter plot with % of negative words (x-axis) vs. % of positive words (y-axis) and colored by final predicted sentiment (green = Positive, red = Negative).

Solution

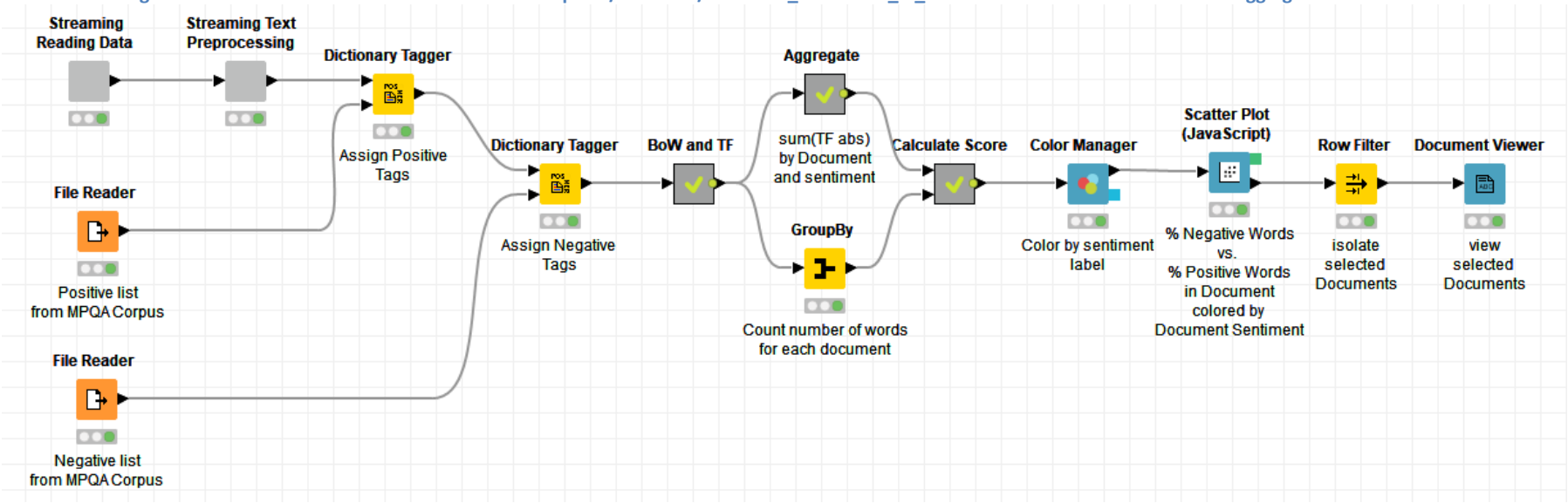
The solution workflow is shown in figure 7.5. After reading the dataset, converting the text rows into Documents, we perform some basic text pre-processing, which does not include any POS tagging or POS filtering.

After that, we read the two lists of positive and negative words from the MPQA corpus and we tag the words in the Documents accordingly.

We then generate the Bag of Words and we count the number of positive words and the number of negative words in each Document. The sentiment score is calculated as (#positive words - # negative words) / #all words in the Document. If sentiment score is greater than 0, then the Document is labelled as positive; otherwise as negative.

An interactive scatter plot finally displays each Document in terms of % of negative words vs. % of positive words. The dots representing the Documents are also colored by their final sentiment label (Fig. 7.6).

Figure 7.5. Solution workflow available under Chapter7/Exercises/Exercise1_Sentiment_in_Forum. Notice that we use no POS tagging in this workflow.



Since the scatter plot is interactive, points can be selected singularly or in group via a rectangle. We have chosen 4 points: two positive (green) and two negative (red). The two negative Documents are on top of each other and this is why we see only one in the plot. The selected points are a bit bigger than the other points in the plot.

The Row Filter node after the Scatter Plot (Javascript) node extracts these selected points - via “Selected” = true - and the Document Viewer node visualizes the underlying Documents.

Figure 7.6. Scatter Plot of Forum dataset Documents. Each Document is represented in terms of % of positive words vs. % of negative words. Green points are positive Documents; red points are negative Documents.

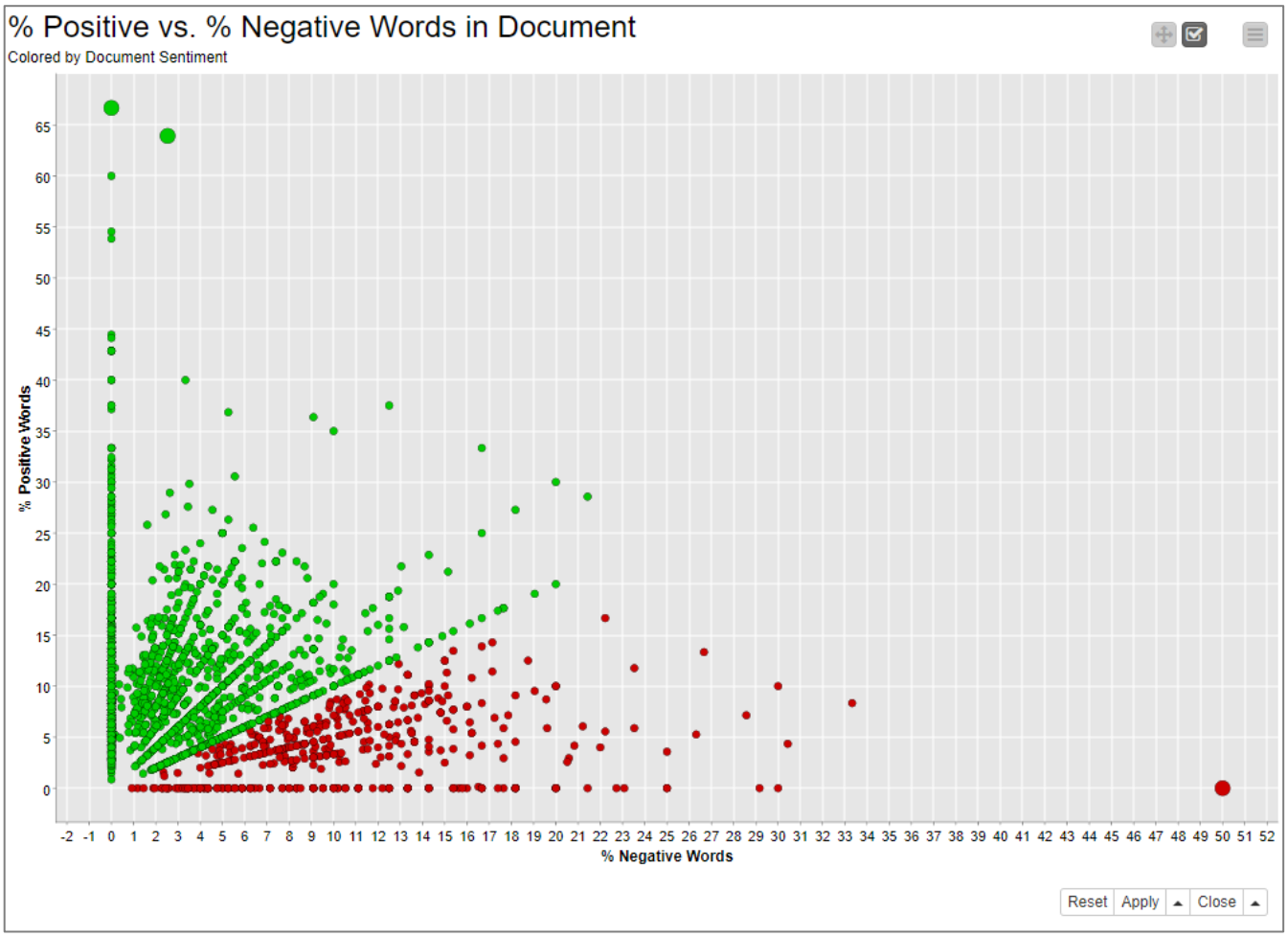


Figure 7.7. This post has only 2 words: “deep learning”. “deep” is considered negative and hence 50% of negative words.

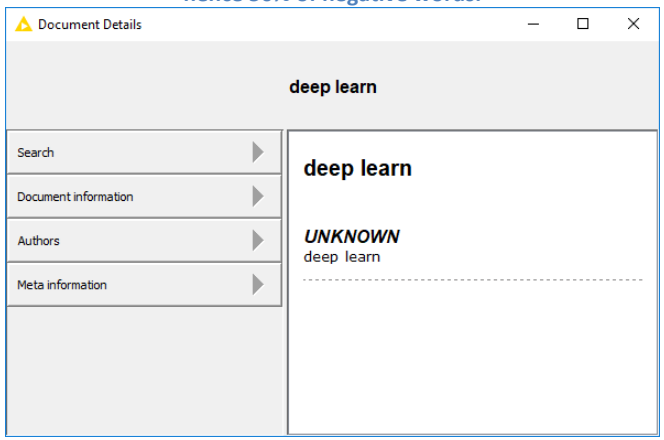


Figure 7.8. This post has only two words: “follow error”. “error” is a negative word and hence 50% of negative words.

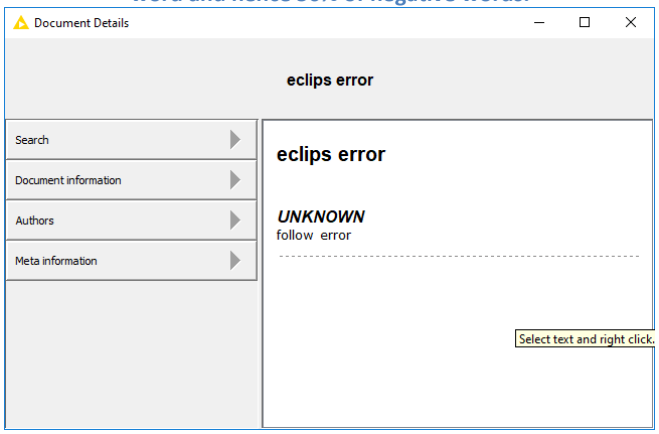


Figure 7.9. This post refers to an error, but has a long list of words “preparation”, which is considered a positive word according to the dictionary. This brings a high percentage of positive words within this post.

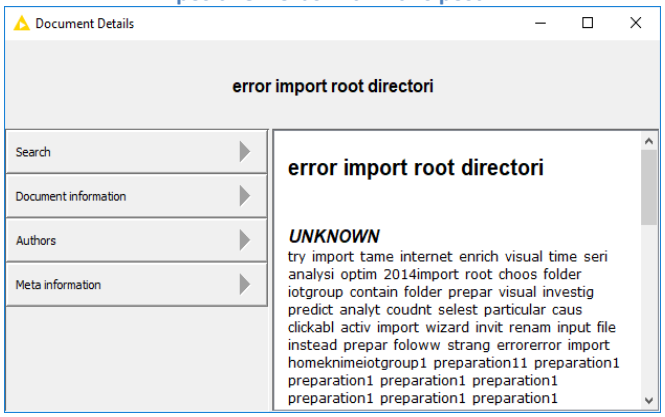
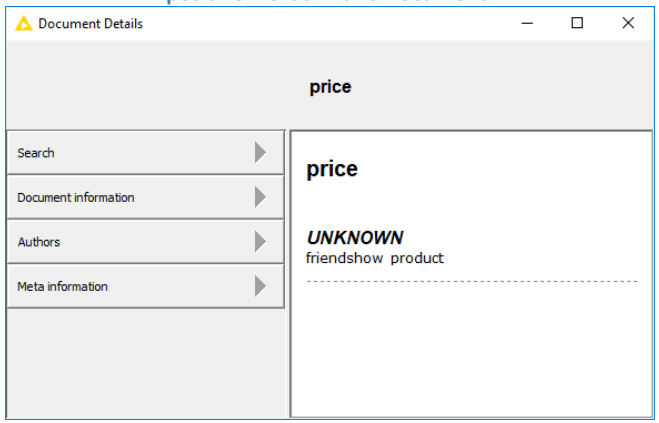


Figure 7.10. This post has only two words: “friendshow product”. Any word starting with “friend” is considered positive in the dictionary and hence 50% of positive words in this Document.



References

[1] Berthold M. R., "Open Innovation in the Age of Big Data", Datafloq Blog, 2015

[2] Amatriain X., "Is Data more important than Algorithms in AI?", Forbes blog, 2017

[3] Rotella P., "Is Data the new Oil?", Forbes Blog, 2012

[4] Silipo R., "KNIME Beginner's Luck", KNIME Press, 2010

[5] Various Authors, "Will they Blend? The Blog Post Collection", Silipo R. Ed., KNIME Press 2017

[6] Brants T. (2000) TnT - A Statistical Part-of-Speech Tagger. In *6'th Applied Natural Language Processing*.

[7] Adwait Ratnaparkhi (1996). A maximum entropy model for part-of-speech tagging. *EMNLP 1*, 133-142.

[8] Shen L., Satta G., and Joshi A. (2007). Guided learning for bidirectional sequence classification. In *Proc. of ACL*.

[9] Porter, M. (1980). An Algorithm for Suffix Stripping. Program, pp. 130-137.

[10] Rainer Kuhlen, "Experimentelle Morphologie in der Informationswissenschaft" Verlag: Dokumentation, München, 1977

[11] Manning, Christopher D., Surdeanu, Mihai, Bauer, John, Finkel, Jenny, Bethard, Steven J., McClosky, David (2014). The Stanford CoreNLP Natural Language Processing Toolkit In *Proc. of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 55-60.

[12] Sparck Jones, K. (1972). A statistical interpretation of term specificity and its application. *Journal of Documentation*, Vol.28, pp. 11-21.

[13] Robertson, S. (2004). Understanding inverse document frequency: on theoretical arguments for IDF. *Journal of Documentation*, Vol. 60 no. 5, pp 503–520.

[14] Manning C. D., (2009). *An Introduction to Information Retrieval.* Cambridge, England: Cambridge University Press.

[15] Stuart J. Russell, P. N. (2003). *Artificial Intelligence: A Modern Approach (2nd ed.).* Upper Saddle River, New Jersey: Prentice Hall.

[16] David Newman, A. A. (2009). Distributed Algorithms for Topic Models. *Journal of Machine Learning Research*, 1801-1828.

[17] Beliga S., A. M.-I. (2015). An Overview of Graph-Based Keyword Extraction Methods and Approaches. *Journal of information and organizational sciences*, 1-20.

[18] Matsuo, Y., Ishizuka, M. (2004). Keyword extraction from a single document using word co-occurrence statistical information. *International Journal on Artificial Intelligence Tools* 13, 157–169.

[19] Oshawa, Benson, and Yachida (1998) KeyGraph: Automatic Indexing by Co-occurrence Graph based on Building Construction Metaphor

[20] Blei, Ng, and Jordan. (2003). Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 993-1022.

[21] Newman, Asuncion, Smyth and Welling (2009) Distributed Algorithms for Topic Models. *JMLR*

[22] Yao, Mimno and McCallum (2009) Efficient Methods for Topic Model Inference on Streaming Document Collections, *KDD*

[23] Le Q., Mikolov T. (2014) Distributed Representations of Sentences and Documents, Proceedings of the 31st International Conference on Machine Learning, Beijing, China, 2014. JMLR: W&CP volume 32.

[24] Analytics Vidhya (2017), An Intuitive Understanding of Word Embeddings: From Count Vectors to Word2Vec

[25] McCormick, C. (2016, April 19). *Word2Vec Tutorial - The Skip-Gram Model*. Retrieved from http://www.mccormickml.com http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

Node and Topic Index

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Open for Innovation ®
KNIME

From Words to Wisdom

This book extends the catalogue of KNIME Press books with a description of techniques to access, process, and analyze text documents using the KNIME Text Processing extension. The book covers text data access, text pre-processing, stemming and lemmatization, enrichment via tagging, keyword extraction, word vectors to represent text documents, and finally topic detection and sentiment analysis. Some basic knowledge of KNIME Analytics Platform is required. The book has been updated for KNIME Analytics Platform 3.5.

About the Authors

Vincenzo Tursi *has been working as Data Scientist at KNIME since May 2016. During this time he worked on text processing, network graph analysis and 360-degree customer data analysis. Before joining KNIME, Vincenzo worked as Business Consultant for Capgemini S.p.A and Business Integration Partners S.p.A. in Italy. He then moved to Germany to work shortly as a Research Associate at Saarland University first and to KNIME later.*

Rosaria Silipo *has been mining data, big and small, since her master degree in 1992. She kept mining data throughout all her doctoral program, her postdoctoral program, and most of her following job positions. So many years of experience and passion for data analytics, data visualization, data manipulation, reporting, business intelligence, and KNIME tools, naturally led her to become a principal data scientist and an evangelist for data science at KNIME.*

ISBN: 978-3-9523926-2-1