The KNIME Text Processing Plugin

Kilian Thiel

Nycomed Chair for Bioinformatics and Information Mining, University of Konstanz, 78457 Konstanz, Deutschland, Kilian.Thiel@uni-konstanz.de

Abstract. This document describes the function, usage and underlying structure of the KNIME [1] text processing plugin. The plugin provides new data types and nodes, which allow to parse textual documents, represent these documents by KNIME data types and operate with them in multiple manners. Documents can be enriched in a way such that named entities are recognized, structural representation can be changed, like transforming a bag of words into a vector space and several preprocessing mechanisms can be applied. Altogether the KNIME text processing plugin is a powerful tool to integrate, process, and handle textual data the KNIME way.

1 Introduction

The KNIME Text Processing plugin was developed to process textual and natural language data. It is a mixture of natural language processing, text mining and information retrieval. With this plugin it is possible to read textual data, represent this data internally as documents and terms and apply several tagging, filtering, frequency computation, data structure transformation and other preprocessing and analysis tasks on it. Finally, the processed text data can be transformed into vector spaces and mined by usual data mining methods provided by KNIME. This means that the text processing plugin basically is a kind of intermediary plugin, reading, preprocessing and transforming textual data into a numerical representation.

In the next section the philosophy and usage of the plugin is described: which procedures to apply in which order and how to set them up. Section 3 depicts the available nodes and their functionality. In Section 4 the underlying data structures and new KNIME data types are explained, as well as the different data table structures, like bag of words, lists of documents, document vectors, etc.

2 Philosophy

In this section the philosophy of the text processing plugin is explained, meaning which categories of nodes have to or can be applied in which order. The order is important since the nodes of the plugin require certain structures or specifications of input data tables, i.e. the “BoW” node, creating a bag of words requires
Fig. 1. An example workflow illustrating the basic philosophy and order of the text processing nodes.

a list of documents as input data and returns a bag of words as output data. This node will only work (configure) properly with a list of documents as input data. Like the “BoW” node also other nodes require specific input data table structures and return specific output data tables. Consequently certain types of nodes can only be used in a certain order. This order or configuration is the basic policy of the KNIME text processing plugin.

The example workflow in figure 1 illustrates the basic order of text processing nodes.

In the following the order of procedures is specified and explained in detail:

1. Parsing
2. Enrichment
3. Transformation
4. Preprocessing
5. Frequencies and keyword extraction
6. Transformation

2.1 Parsing

First of all the text documents have to be parsed. The text and the structure, if available, have to be extracted and represented in a data structure, KNIME is able to access and use in an appropriate manner. A document is represented internally by an instance of the Java class Document, this class is wrapped by
the KNIME data type DocumentCell and DocumentValue. More details about these data types and the underlying data structure can be found in section 4. For now the only important thing is that the parser nodes read text stored in a certain format, analyze the structure and convert both into a DocumentCell for each document to parse.

All nodes of the KNIME text processing plugin can be found in the category “Textprocessing” of the node repository, if the plugin is installed properly. The parser nodes are in the subcategory “IO”. Two nodes, the “DML Document Parser” and the “SDML Document Parser” parse XML document formats which are used to store the data of DocumentCells internally or to serialize them respectively. Additionally a PubMed parser node is available enabling to read PubMed XML results. Furthermore the “Flat File Document Parser” node reads unstructured text from flat files and represents it in document form.

The question now is, how to get your text data into KNIME? If you have PubMed text data you can simply use the PubMed parser, for flat file, the flat file parser will do, but what for a particular XML format?

1. One possibility is to transform this particular XML into the internally used DML or SDML, which is basically the same but simpler. Once the particular XML is transformed, i.e. by XSLT, the DML or SDML parser can be used to parse the data.
2. Another way would be, to write your text into flat text files and use the flat file parser, the main drawback of this option is that the structural information of the text will be lost.
3. The third possibility is to implement an additional parser node, which is able to parse the particular format.

The last option is definitively the most complex and time consuming one but once the node is implemented no more transformation has to be done. As a start it is recommendable to use one of the existing parsers and transform the texts into the DML or SDML format.

Once the texts have been parsed, the output data table of the corresponding parser node contains the documents, in each row one, as illustrated in figure 2. The icon, shown in the header of the second column, points out that the type of the data contained by this column is the document data.

### 2.2 Enrichment and tagging

After parsing the text and converting it into document cells, the documents and the included information can be enriched, i.e. by recognizing named entities. All enrichment nodes require a data table containing exactly one column of document cells and returns an output data table with exactly one column of document cells. Each node analyses the documents in a way such that it combines multiple words to terms, if reasonable, and applies certain tags to these terms. These tagged terms can be recovered later on, and be treated in a certain way, for instance, be filtered. A document after the tagging process contains more information than a document before, it has been enriched.
Fig. 2. A data table containing a list of PubMed documents, in each row one.

All available enrichment nodes can be found in the “Enrichment” node category. The “POS tagger” node adds part of speech tags, according to the Penn Treebank tagset\(^1\), to each word, which is afterwards considered as a term. Later on a filtering can be applied based on the POS tags, i.e. is it possible to filter all nouns or verbs. The underlying model used to assign the POS tags was taken from the OpenNLP project\(^2\). OpenNLP models that recognize named entities, such as locations, organizations, or persons have been wrapped by the “OpenNLP NE tagger” node. Another enrichment node is the “Abner tagger” which recognizes biomedical named entities, such as cell lines, genes, or proteins. The model used here is the “ABNER” [2] model. Since there are many different types of named entities a “Dictionary tagger” has been implemented, able to read a certain dictionary of named entities, search the documents for them and tag the corresponding terms. Once a dictionary of interesting named entities, i.e. company names is created, this node can be used to find them.

Once words have been recognized as named entities, bundled together as terms and been tagged, these terms can be set unmodifiable, meaning that preprocessing nodes will not filter or change these terms anymore. To set the terms unmodifiable simply check the “Set named entities unmodifiable” check box of the enrichment nodes, as shown in Figure 3.

A very important behavior of these tagger nodes is that the node applied in the end of the enrichment workflow dominates in case of conflicts. Assume a certain named entity recognizer node finds and tags company names, now assume another enrichment node is applied exactly afterwards in the workflow and a conflict appears related to the tagging of a certain term. The last applied node breaks the conflicting terms into words, recombines them in a way of the enrichment strategy and finally applies tags to them. This means that the word combination of the antecessor node does not exist anymore.

---

\(^1\) http://www.cis.upenn.edu/ treebank/
\(^2\) http://opennlp.sourceforge.net/
2.3 Transformation

Before the enriched documents can be preprocessed, i.e. filtered, stemmed etc., the data table structure has to be transformed into a bag of words. Therefore the “BoW creator” node has to be used. All nodes, transforming the data table structure can be found in the “Transformation” category. The BoW node requires a list of documents as input data table, consisting of exactly one column of document cells. The input documents and its terms are converted into a bag of words structure. The output data table consists of two columns, the first is the term column and the second the document column. A row of this data table means that the term in this row occurs in the document. Figure 4 shows a bag of words data table with two rows, one containing the terms and the other the documents. Be aware that the terms tags are listed in brackets after the term.

2.4 Preprocessing

Once a bag of words is created preprocessing nodes, such as filtering, stemming etc. can be applied, which can be found in the “Preprocessing” category. These nodes require a bag of words input data table and return a preprocessed bag of words output data table. Filters simply filter rows according to the terms of these rows and the filter criteria. For instance:

- the “Stop word Filter” node filters terms which are *stop words*
- the “N Char Filter” filters terms consisting of less then a specified number of characters
Fig. 4. A bag of words data table containing two columns, one consisting of terms and one of documents.

- the “POS Filter” filters terms which are tagged as nouns or verbs, etc.

The available set of filters allows to filter irrelevant terms and reduce the size of the bag of words. Additionally preprocessing nodes are available which modify terms, such as

- the “Porter stemmer”, reducing the words to their stems by the porter [3] algorithm
- the “Case converter”
- the “Punctuation erasure”, removing all punctuation marks
- the “Replacer”, replacing specified sequences of characters with other characters.

To stem English words the Porter or the Kuhlen stemmer can be used. For other languages the Snowball stemmer node has to be applied, using the Snowball3 stemmer library. A very important issue about preprocessing a bag of words data table is that on the one hand only the terms contained in the term column can be preprocessed. On the other hand the terms contained in the document itself can be affected by the preprocessing procedure as well. This is called deep preprocessing. As one can imagine deep preprocessing is more time consuming than the shallow one, since it processes the terms inside the document too and builds a new document afterwards.

The deep preprocessing is essential if one wants to compute frequencies afterwards, based on the preprocessed terms. For example, if a term has been stemmed it is impossible to the term frequency node, to find and count the stemmed term in the original document. The terms contained in the document have to be stemmed too. Therefore the deep preprocessing option is available for all preprocessing nodes and can simply be activated by checking the “Deep preprocessing” check box in the node dialog, shown in Figure 5. The original document can be appended unchanged, so that both versions are available.

3 http://snowball.tartarus.org/
The dialog of the POS filter. The topmost check box is the “Deep preprocessing” check box. If checked deep preprocessing will be applied.

Additionally it can be specified whether the preprocessing step of the node is applied to terms, which have been set unmodifiable as well, see bottom of Figure 5. By default this option is unchecked, meaning that terms which have been set unmodifiable by an enrichment node will not be affected by the preprocessing. In some cases ignoring the unmodifiable flag can be useful, i.e. when named entity recognition has been applied and persons and locations have been identified and tagged. To separate persons from locations the “Standard Named Entity Filter” node has to be used ignoring the unmodifiable flag of persons or locations respectively. Otherwise the node would not filter either persons or locations.

2.5 Frequencies and Keyword extraction

By applying preprocessing nodes, the bag of words can be reduced and irrelevant terms can be filtered out. To specify the relevancy of a term, several term frequencies can be computed and the terms can be filtered according to these
values. To compute these frequencies two nodes are available, one to compute the well known term frequency \(tf\), relative or absolute, and one to compute the inverse document frequency \(idf\). These nodes require a bag of words as input data table, and return a bag of words as output data table, with additional columns containing the computed frequencies.

After the frequencies have been computed the terms can be filtered, by the “Frequency Filter” node, related to the frequency values. It is possible to specify minimum and maximum values or a particular number \(k\), so that only the \(k\) terms with the highest frequency are kept. Beside extracting keywords based on their \(tfidf\) value other nodes can be applied as well, like the “Chi-square keyword extractor” using a chi-square measure to score the relevance [4] or the “Keygraph keyword extractor” [5] using a graph representation of documents to find keywords.

2.6 Transformation

Now only the relevant terms should remain in the bag of words and one can finally begin with the mining task. Therefore one last transformation has to be done. The bag of words has to be transformed into vectors. Again the necessary transformation nodes can be found in the “Transformation” category. The “Document vector” node creates a document vector for each document, representing it in the term space. This means, every different term is considered as a feature and the document vector is a vector in this feature space. If a document contains a certain term, a specific value (usually \(tfidf\) or 1) occurs in the document vector at the corresponding dimension. Analogous the “Term vector” node creates a term vector for each term, representing it in the document space. Once these vectors have been created it is possible to apply data mining methods to them. Unlabeled document vectors for instance can be clustered, labeled ones can be classified, or term association rules can be mined.

3 Nodes

In the following table all the available nodes are listed together with a short description of their functionality.

<table>
<thead>
<tr>
<th>IO</th>
<th>DML Document Parser</th>
<th>Parses DML formatted files.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Document Grabber</td>
<td>Sends specified query to PubMed, downloads and parses the resulting documents.</td>
</tr>
<tr>
<td></td>
<td>Flat File Document Parser</td>
<td>Reads flat files and creates one document for each.</td>
</tr>
<tr>
<td></td>
<td>SDML Document Parser</td>
<td>Parses SDML formatted files.</td>
</tr>
<tr>
<td>Enrichment</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Abner tagger</td>
<td>Recognizes and tags biomedical named entities based on ABNER.</td>
<td></td>
</tr>
<tr>
<td>Dictionary tagger</td>
<td>Searches for named entities specified in a dictionary and tags them.</td>
<td></td>
</tr>
<tr>
<td>OpenNLP NE tagger</td>
<td>Recognizes persons, locations, organization, money, date, or time, based on different OpenNLP models.</td>
<td></td>
</tr>
<tr>
<td>POS tagger</td>
<td>Tags words with part of speech tags.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BoW creator</td>
<td>Transforms a list of documents into a bag of words.</td>
</tr>
<tr>
<td>Document Data Extractor</td>
<td>Extracts specified fields, like authors, publication date, title etc. of documents and adds them as string columns.</td>
</tr>
<tr>
<td>Document vector</td>
<td>Transforms documents in a bag of words into document vectors.</td>
</tr>
<tr>
<td>Sentence Extractor</td>
<td>Extracts all sentences of documents and adds them as string column.</td>
</tr>
<tr>
<td>String to Term</td>
<td>Converts strings into terms.</td>
</tr>
<tr>
<td>Strings to Document</td>
<td>Builds documents out of specified strings.</td>
</tr>
<tr>
<td>Tags to String</td>
<td>Converts tag values to strings.</td>
</tr>
<tr>
<td>Term to String</td>
<td>Converts a term into a string.</td>
</tr>
<tr>
<td>Term vector</td>
<td>Transforms terms in a bag of words into term vectors.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Preprocessing</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abner Filter</td>
<td>Filters terms tagged with ABNER tags.</td>
</tr>
<tr>
<td>Case converter</td>
<td>Converts terms to lower or upper case.</td>
</tr>
<tr>
<td>Dict Replacer</td>
<td>Replaces certain terms or substrings of terms with a particular replacement all specified in a dictionary.</td>
</tr>
<tr>
<td>Kuhlen stemmer</td>
<td>Stems words with the Kuhlen algorithm.</td>
</tr>
<tr>
<td>Modifiable Term Filter</td>
<td>Filters terms which have been set un(modifiable).</td>
</tr>
<tr>
<td>N Chars Filter</td>
<td>Filters terms with less than N characters.</td>
</tr>
<tr>
<td>Number Filter</td>
<td>Filters term which are numbers.</td>
</tr>
<tr>
<td>POS Filter</td>
<td>Filters terms with specific part of speech tags.</td>
</tr>
<tr>
<td>Porter stemmer</td>
<td>Stems terms with the Porter algorithm.</td>
</tr>
<tr>
<td>Punctuation Erasure</td>
<td>Removes all punctuation marks from terms.</td>
</tr>
<tr>
<td>Replacer</td>
<td>Replaces in terms a certain character sequence by another one.</td>
</tr>
<tr>
<td>Node</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Snowball Stemmer</td>
<td>Stems terms using the Snowball stemmer library.</td>
</tr>
<tr>
<td>Standard named entity filter</td>
<td>Filters terms tagged with standard named entities.</td>
</tr>
<tr>
<td>Stop word Filter</td>
<td>Filters terms which are stop words.</td>
</tr>
<tr>
<td>Term Grouper</td>
<td>Groups terms w.r.t. their strings ignoring their tags.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Frequencies</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frequency Filter</td>
<td>Filters terms with a certain frequency.</td>
</tr>
<tr>
<td>IDF</td>
<td>Computes the inverse document frequency of terms.</td>
</tr>
<tr>
<td>TF</td>
<td>Computes the term frequency of terms.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Misc</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category to class</td>
<td>Adds the category of a document as string column.</td>
</tr>
<tr>
<td>Chi-square keyword extractor</td>
<td>Extracts keywords, estimated as relevant by the Chi-square measure.</td>
</tr>
<tr>
<td>Document Viewer</td>
<td>Displays the content of a document.</td>
</tr>
<tr>
<td>Keygraph keyword extractor</td>
<td>Extracts keywords, estimated as relevant by the keygraph algorithm.</td>
</tr>
<tr>
<td>String Matcher</td>
<td>Computes the Levenshtein distance between a list of strings and a string dictionary.</td>
</tr>
<tr>
<td>Tagcloud</td>
<td>Displays a bag of words as tag cloud.</td>
</tr>
</tbody>
</table>

Table 1: A listing of all available nodes on the left and a short description on the right.

4 Structures

In this section the different data table structures required by the text processing nodes, the three new data types, DocumentCell, DocumentBlobCell and TermCell as well as the internal document and term structures and its serialization is described.

4.1 Data table structures

As already mentioned in section 2 the nodes, provided by this plugin, require a certain structure of input data tables in order to configure and execute properly. What kind of data table structure is specified by the category of the nodes. Basically there are three different kind of structures. A table with only one column containing a document cell in each row, shown in figure 2, a bag of words consisting of two columns, one containing a term cell, the other containing a document
Fig. 6. A bag of words data table with an additional column containing the original, unchanged document.

cell in each row, illustrated in figure 4 and a vector structure representing term or document vectors.

This basic bag of words structure can be expanded in a way such that additional columns can be added, like the original, unchanged document, when

Fig. 7. A bag of words data table with an additional column containing \( tf \), \( idf \) and \( tfidf \) frequencies.
deep preprocessing is activated. In figure 6 such kind of a data table is depicted. Furthermore it can be seen that the document titles of the middle column and those of the right column differ. The original document on the right still retains its original title, whereas the document in the middle column, affected by the impacts of deep preprocessing, lost several terms during the filtering process. Not only the documents body is changed by the filtering, the title can change as well.

In addition to the original document column, frequency columns are added by node of the category “Frequencies”. Figure 7 shows a data table with additional columns containing $tf$, $idf$ and $tf \times idf$ frequencies. According to these frequencies certain filters can be applied, as described in 2.5. The final data table structure change is usually the transformation into a vector space, such as the document or term vector space. It is possible to create bit vectors, or to specify a frequency column, which entries are considered as vector values.

![Fig. 8. The dialog of the document vector generator node.](image)

<table>
<thead>
<tr>
<th>Table &quot;default&quot; - Rows: 100</th>
<th>Spec - Columns: 27</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Row ID</strong></td>
<td><strong>Document</strong></td>
<td><strong>D P&lt;0.01</strong></td>
</tr>
<tr>
<td>1</td>
<td>‘Expression of hum...’</td>
<td>0.062</td>
</tr>
<tr>
<td>2</td>
<td>‘Enrollmet Fluid 2...’</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>‘Hyperinsulinemia F...’</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>‘Portal Venous Den...’</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>‘The environmenta...’</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>‘Angioprotein-1 , b...’</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>‘CDH (+)025 (+ ...’</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>‘Overexpression 0...’</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>‘Diabetes and hear...’</td>
<td>0</td>
</tr>
</tbody>
</table>

![Fig. 9. A data table consisting of document vectors.](image)
of the document vector generator is shown in figure 8. The before computed
frequencies can be selected as vector values. Figure 9 illustrates the output data
table of the document vector generator. The first column contains the document,
the following the specified frequency accordant to the document and a certain
term, displayed in the column headings. Based on these vectors conventional
data mining methods, like clustering or categorization, can be applied.

4.2 Document(Blob)Cell

It is obvious that almost all of the data tables processed by the nodes of the text
processing plugin contain documents and/or terms. Therefore three new data
types have been implemented, the DocumentCell, the DocumentBlobCell and
the TermCell. Basically the DocumentCell and the DocumentBlobCell are the
same except that the super class of the DocumentBlobCell is BlobDataCell. This
means that, in contrast to a usual DataCell, the multiple occurrence of a certain
document in a data table is referenced. For instance the bag of word data table
contains tuples of terms and documents, which leads to multiple occurrences of
the same document in that table. To save memory all occurrences of a certain
document, after its first listing, are referenced, meaning that the data table holds
only an address of the corresponding data cell.

This behavior can be switched off in the preference page of the text processing
plugin, which can be found as a sub page in the KNIME preferences. Figure 10
shows the text processing preference page and the check box to specify the usage
of blob cells. Be aware that if no blob cells are used, this will consume a lot of
memory since the same documents are stored multiple times in the data tables.

Fig. 10. Preference page of the text processing plugin.
The benefit of not using blob cells is an increase of processing speed, since the references don’t have to be resolved. However, the benefit can not compensate the drawback if one have to deal with limited memory. By default this check box is checked and blob cells are used. Uncheck this box only after carefully balancing the reasons!

In figure 11 a column containing document cells is illustrated. The heading of this column shows the icon of the document cell. Table views, like the output table view or the interactive table only depict the title of the document.

### 4.3 TermCell

For the adequate representations of terms the data type TermCell is provided by the plugin. A term is not only a string, but a list of words bundled together, constituting a multi word, or term. Furthermore several tags can be attached to a term, accordant to the terms meaning and the specific tagger. For instance the words “New York City” can be recognized as a named entity, more precisely a location, by a named entity tagger and bundled together to a term. Additionally the tag “Location” can be attached to that term. According to the used taggers and the meaning of terms it is possible that terms may have more then one tag attached.

Figure 12 illustrates a column containing term cells. In the column heading the term cell icon is displayed. The string representation of a term, shown in table views is assembled by its words first followed by the attached tags in brackets. First the name of the tag is shown, i.e. “NN” for noun and second in parenthesis the type of the tag, i.e. “POS” for part of speech.

### 4.4 Internal data structures

The two kinds of document cells are both based on the essential Document class. This class encapsulates the document itself, its data and meta data. Basically a document consist of sections, which can be chapter, abstract or title. To each section an annotation can be attached, which specifies the kind of section, i.e.
“Title”. A section again is composed several paragraphs, and a paragraph consists of sentences. A sentence contains terms, and terms contain words and tags, as mentioned before. Figure 13 illustrates the composition of a document.

This level of granularity allows to access all parts of the document easily and enables to capture the principal part of the structure of several document formats.

4.5 Serialization

As all data cells, the document cells can be serialized. The serialization is done via XML, more detail via DML the internally used XML format to save documents. Since the plugin provides two parser nodes, one to parse DML and one for SDML it is useful to convert the documents to parse into DML or SDML. If you already have implemented a parser node which is able to read your specific document format, of course a conversion is unnecessary. But implementing a parser node however is more time consuming than converting your files. In the following the DML and SDML specification is shown.

**DML specification**
Listing 1.1. DTD of the DML format.

<!ELEMENT Document
( Category *, Source *, DocumentType ?, Authors ?,
PublicationDate ?, Section +, Filename ? )>

<!ELEMENT Category (#PCDATA)>
<!ELEMENT Source (#PCDATA)>
<!ELEMENT Filename (#PCDATA)>
<!ELEMENT DocumentType (#PCDATA)>

<!ELEMENT Authors (Author*)>
<!ELEMENT Author (Firstname?, Lastname)>
<!ELEMENT Firstname (#PCDATA)>
<!ELEMENT Lastname (#PCDATA)>

<!ELEMENT PublicationDate (Day?, Month?, Year)>
<!ELEMENT Day (#PCDATA)>
<!ELEMENT Month (#PCDATA)>
<!ELEMENT Year (#PCDATA)>

<!ELEMENT Title (#PCDATA)>
<!ELEMENT Section (#PCDATA)>

<!ATTLIST Section Annotation CDATA #REQUIRED>

<!ELEMENT Paragraph (Sentence+)>
<!ELEMENT Sentence (Term+)>
<!ELEMENT Term (Word+, Tag*)>
<!ELEMENT Word (#PCDATA)>
<!ELEMENT Tag (TagValue, TagType)>
<!ELEMENT TagType (#PCDATA)>
<!ELEMENT TagValue (#PCDATA)>

SDML specification

Listing 1.2. SDML-DTD

<!ELEMENT Document
( Authors?, PublicationDate?, Title, Section+)>

<!ELEMENT Authors (Author*)>
<!ELEMENT Author (Firstname?, Lastname)>
<!ELEMENT Firstname (#PCDATA)>
<!ELEMENT Lastname (#PCDATA)>

<!ELEMENT PublicationDate (Day?, Month?, Year)>
<!ELEMENT Day (#PCDATA)>
<!ELEMENT Month (#PCDATA)>
<!ELEMENT Year (#PCDATA)>

<!ELEMENT Title (#PCDATA)>
<!ELEMENT Section (#PCDATA)>

<!ATTLIST Section annotation CDATA #REQUIRED>
5 Visualization

So far there are two nodes that visualize textual data, the “Document Viewer” and the “Tagcloud”. The “Document Viewer” node visualizes documents in a very simple way. A data table containing a list of documents can be used as input data. The document viewer shows first a list of all available documents to visualize, as shown in Figure 14.

A double click on a particular document row will open a new frame showing the details of the corresponding document. All the detail data of documents are shown. In the bottom most panel, filename, publication date, document type, categories and sources are displayed. In the panel above all authors names are shown and in the main panel the text of the document and its title are displayed. Furthermore assigned tags of a specified type can be highlighted, which can be seen in Figure 15. Here terms that have ABNER tags assigned are highlighted red.

The “Tagcloud” node provides a typical tagcloud with some additional options, such as different arrangements of terms, i.e. size sorted, alphabetic, inside out, adjustable minimal and maximal font size, or transparency of terms etc. The node requires a input data table consisting of a bag of word with a additional column containing a weight or score of each term. This weight is used to

![Fig. 14. The list of available documents.](image)
Fig. 15. The view of a particular document, showing all its data, such as authors, categories, publication date, text etc.

determine the terms font size, its transparency and boldness. Figure 16 shows a tag cloud containing named entities, such as organizations, locations and persons. It can be seen that the term “Microsoft” is the largest term due to its high frequency which is used as weight value.

Additionally the terms can be colored differently. Therefore the on the one hand the “Color Manager” node can be used which assigns colors to each row according to the values of a specified column. In Figure 16 named entity tags have been extracted as strings and colors have been assigned based on these tag values. Organizations are colored green, persons red and locations blue. The tags have been assigned by the “OpenNLP NE Tagger” node.
6 Acknowledgments

Thanks to Pierre-Francois Laquerre and Iris Adă for their work on chi-square keyword extractor and keygraph keyword extractor resp. the tag cloud display.

References