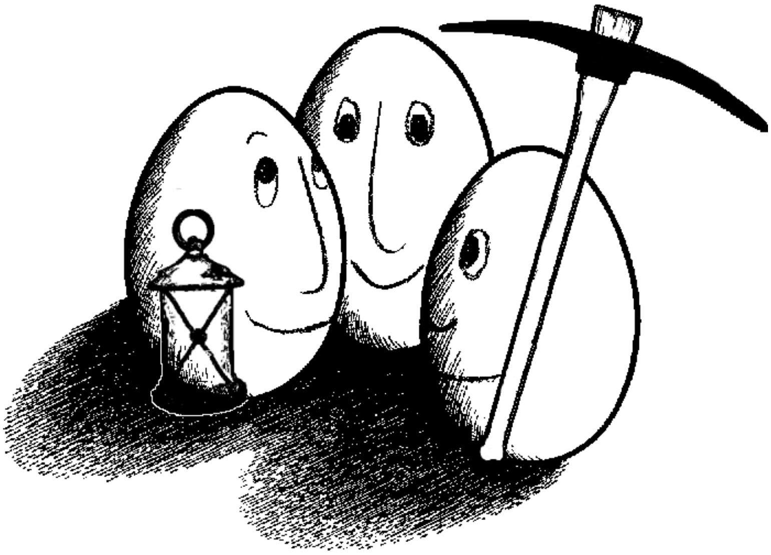
The Word Vector Tool

User Guide
Operator Reference
Developer Tutorial



Michael Wurst
michael.wurst@uni-dortmund.de

http://wvtool.sf.net/

May 25, 2007

Contents

Chapter 1

Introduction

The Word Vector Tool (*WVTool*) is a flexible Java library for statistical language modeling. In particular it is used to create word vector representations of text documents in the vector space model [1]. In the vector space model, a document is represented by a vector that denotes the relevance of a given set of terms for this document. Terms are usually natural language words, but they can also be more general entities, as words that are reduced to some linguistic base form or abstract concept as "<number>" denoting any occurrence of a number in the text.

	agent	Java	<number>	...
doc1.txt	1.0	0.3	0.0	...
doc2.txt	0.9	0.0	0.6	...
...	...	...	...	...

From the early days of automatic text processing and information retrieval, the vector space model has played a very important role. It is the point of departure for many automatic text processing tasks, as text classification, clustering, characterization and summarization as well as information retrieval [2].

The aim of the Java *WVTool* is to provide a simple to use, simple to extend pure Java library for creating word vectors. It can be easily invoked from any Java application. Furthermore, the tool is tightly integrated with the RapidMiner data mining environment [3], allowing to perform diverse experiments using textual data directly. In this way, the *WVTool* bridges a gap between highly sophisticated linguistic packages as the GATE system [11] on the one side and many partial solutions that are part of diverse text and information retrieval applications on the other side. Closest related to the Word Vector Tool is the Bow package [10], which is a C library, for the creation of word vectors and clustering/classifying text.

In the next chapter, the basic concepts of the library are explained and how to use it from Java applications. Chapter 3 discusses the RapidMiner integration. In chapter 4 some advanced topics as using a web crawler or dictionaries are introduced. Chapter 5 gives a brief overview of the performance of the *WVTool* on a test corpus.

Chapter 2

Using the WVTool as Java Library

The *WVTool* can be used as a standalone Java library or as plugin for the RapidMiner data mining environment. In this section, we will first discuss the use of the *WVTool* as library. In section 3 the use of the *WVTool* in RapidMiner is discussed in detail.

2.1 Installation

To use the *WVTool* as Java library, first obtain a copy of the *WVTool* from the sourceforge *WVTool* homepage[1], uncompress the archive and put the wvtool.jar file and all jar files in the lib subdirectory into your classpath.

There are two basic operations the *WVTool* is able to perform: 1. Create a word list (the dimensions of the vector space) from a set of text documents and 2. Create word vectors from a set of texts (based on a word list). A word list contains all terms used for vectorization together with some statistics (e.g. in how many documents a term appears). The word list is needed for vectorization to define which terms are considered as dimensions of the vector space and for weighting purposes.

Both functions have two basic input parameters. First, an input list that tells the system which text documents to process and second, a configuration object, that tells the system which methods to use in the individual steps.

[1] http://wvtool.sourceforge.net

2.2 Defining the Input

The input list tells the *WVTool* which texts should be processed. Every item in the list contains the following information:

- A URI to the text resource. Currently this can be a local file/directory or a URL
  In the case of a directory, all files in this directory are processed (not recursing to subdirectories). As the *WVTool* is extendable, other types of file references could be used as well, as long as the user provides a method that handles them (see 2.3)
- The language the document is written in (optional)
- The type of the document (optional)
- The character encoding of the document, e.g. UTF-8 (optional)
- A class label
  Texts can be assigned to classes, such as topics. This information is usually used for automatic text classification, but could be relevant for word vectorization as well. A class label index is ranging from 0 to $m-1$, where m is the number of classes (optional)

In the following example, an input list with three entries is created, two pointing to documents on the local file system and one pointing to a webpage.

```
//Initialize the input list with three classes

WVTFileInputList list = new WVTFileInputList(3);

//Add entries

list.addEntry(
   new WVTDocumentInfo("data/alt.atheism",
                       "txt","","english",0));

list.addEntry(
   new WVTDocumentInfo("data/soc.religion.christian",
                       "txt","","english",1));

list.addEntry(
   new WVTDocumentInfo("http://www-ai.cs.uni-dortmund.de",
"html","","english",2));
```

Every entry is assigned to one class.

2.3 Configuration

基本的思路是，向量化和单词列表的生成是由一个固定顺序的步骤生成的

The *WVTool* is written in a modular way, as to allow a maximum of flexibility and extendibility. The general idea is, that vectorization and word list creation consists of a fixed sequence of steps. For every step in the vectorization process, the user states the Java class that should be used for this step. This class can be one already included in the tool or a new one, written by the user. The only constraint is, that it has to implement the corresponding interface of a given step. In the following, these steps will be described in more detail together with the available Java implementations:

constraint
约束

- *TextLoader*
  The TextLoader is responsible for opening a stream to the processed document. Currently, the system provides one loader capable of reading from local files and URLs. The corresponding class is called *UniversalLoader* and should be sufficient for most applications.

  `UniversalLoader` - Loads texts from local files and URLs (default)

- *Decoder*
  If the text is encoded/wrapped (e.g. in HTML code), it has to be decoded to plain text before vectorization. Currently, only plain text (no decoding necessary) and XML/HTML based markup languages (tags are ignored) are supported.

  `SimpleTagIgnoringReader` - Removes tags from a file without parsing it.

  `XMLInputFilter` - Parses the file and removes tags from it.

  `TextInputFilter` - Reads the file as text file.

  `PDFInputFilter` - Extracts the text parts of a PDF file.

  `SelectingInputFilter` -Selects the input filter automatically, based on the file suffix (default).

  An important thing to note here is encoding. All readers, beside the *PDFInputFilter*, evaluate the encoding information given for each entry in the input list. If no (legal) encoding is given, the system default is used. Note, that currently the encoding cannot be determined automatically for XML and HTML files.

- *CodeMapper*
  In some cases the encoding of a text has to be mapped to another encoding. One might like to remove all the accents from a French text for instance in this step. At the moment only a dummy class is available.

  `DummyCharConverter` - does nothing (default).

- *Tokenizer*
  The tokenizer splits the whole text into individual units. Tokenization is a non-trivial task in general. Though for vectorization often a simple heuristic is sufficient. Currently, only one tokenizer is available, which uses the Unicode specification to decide whether a character is a letter. All non-letter characters are assumed to be separators, thus the resulting tokens contain only letters. Additionally, there is a tokenizer that creates character n-grams from given tokens.

  `SimpleTokenizer` - tokenization based on letters and non-letters (default).

  `NGramTokenizer` - creates character n-grams.

分词

- *WordFilter*
  In this step, tokens that should not be considered for vectorization are filtered. These are usually tokens appearing very often (referred to as "stopwords". Standard English and German stopword lists are included. You may also specify the stopwords using a file.

  `StopWordFilterFile` - reads stopwords from a file.

  `StopWordsWrapper` - a standard English stopword list (default).

  `StopWordsWrapperGerman` - a standard German stopword list.

  `DummyWordFilter` - does not filter anything.

  `CombinedWordFilter` - combines two or more word filters in a disjunctive way.

- *Stemmer/Reducer*
  Often it is useful to map different grammatical forms of a word to a common term. At the moment the system incorporates several different stemming algorithms: a Porter Stemmer, a Lovins Stemmer, a German Stemmer and the Snowball Stemmer package (providing stemmers for different languages, see [4]). Also, there is the possibility to define additionally an own dictionary or to use the Wordnet thesaurus (see 4.2).

  `LovinsStemmerWrapper` - a Lovings stemmer (default)

  `PorterStemmerWrapper` - a Porter Stemmer

  `SnowballStemmerWrapper` - the Snowball stemmer package.You need to define the language of each text that is parsed, as the corresponding stemmer is chosen according to this information

`ToLowerCaseConverter` - converts all characters in the word to lower case

`DictionaryStemmer` - uses a manually specified dictionary to reduce words to a base form (see 4.2.1 for more information)

`DummyStemmer` - does nothing

`WordNetHypernymStemmer` - uses Wordnet to replace a word by its hypernym (see 4.2.2 for more information)

`WordNetSynonymStemmer` - uses Wordnet to replace a word by a representative element of its synset (see 4.2.2 for more information)

- *VectorCreation*
  After the tokens have been counted, the actual vectors have to be created. There are different schemes for doing this. They are based on the following counts:

  f_{ij} the number of occurrences of term i in document j
  fd_j the total number of terms occurring in document j
  ft_i the total number of documents in which term i appears at least once

  Based on these counts, currently four classes are available that measure the "importance" of term i for document j, as denoted by v_{ij}:

  `TFIDF` - the tf/idf measure with $v_{ij} = \frac{f_{ij}}{fd_j} log(\frac{|D|}{ft_i})$, where $|D|$ is the total number of documents. The resulting vector for each document is normalized to the Euclidean unit length (default).

  `TermFrequency` - the relative frequency of a term in a document, $v_{ij} = \frac{f_{ij}}{fd_j}$. The resulting vector for each document is normalized to the Euclidean unit length.

  使用词频统计

  `TermOccurrences` - the absolute number of occurrences of a term $v_{ij} = f_{ij}$ The resulting vector is not normalized.

  (1203)考虑到小数难以转换，还是选择使用单词出现次数

  `BinaryOccurrences` - occurrences as a binary value $v_{ij} = \begin{cases} 1, & f_{ij} > 0 \\ 0, & else \end{cases}$

  The resulting vector is not normalized.

- *Output*
  The output steps determines where the resulting vectors are written to. Currently, only writing them to a file is supported. This step must be configured, as there is no default where to write the vectors to.

  configuration
  配置

The *WVTool* Operator allows you to specify which java class to use for a given step. This can be done in a static way (for each document the same java class is used) or dynamically (the java class is chosen depending on properties of the document, such as the language or the encoding). The following two examples describe this process. The first example sets the java class for the output step in a static way.

动态和静态两种参数选择的方式

```
FileWriter outFile = new FileWriter("wv.txt");

WordVectorWriter wvw = new WordVectorWriter(outFile, true);

config.setConfigurationRule(WVTConfiguration.STEP_OUTPUT,
   new WVTConfigurationFact(wvw));
```

The second example selects the the stemming algorithm dynamically, depending on the language the text document is written in:

```
final WVTStemmer dummyStemmer =
   new DummyStemmer();

final WVTStemmer porterStemmer =
   new PorterStemmerWrapper();

config.setConfigurationRule(WVTConfiguration.STEP_STEMMER,
   new WVTConfigurationRule() {

   public Object getMatchingComponent(WVTDocumentInfo d)
     throws Exception {
     if(d.getContentLanguage().equals("english"))
        return porterStemmer;
     else
        return dummyStemmer;
   }
});
```

By writing your own classes (implementing the corresponding interface) you can use your own methods instead of the ones provide with the tool.

2.4 Using Predefined Word Lists

In some cases it is necessary to exactly define the dimensions of the vector space, yet leaving the counting of terms and documents to the *WVTool*. This can be

事先定义单词列表，暂时用不到

achieved by calling the word list creation function with a list of String values as in the following example (creating a word list with only two entries):

```
List dimensions = new Vector();

dimensions.add("apple");
dimensions.add("pc");

wordList =
   wvt.createWordList(list, config, dimensions, false);
```

The last parameter determines whether additional terms occurring in the texts should be added to the word list.

Chapter 3

The Word Vector Tool and RapidMiner

Instead of using the *WVTool* as a library, you can use it directly with the RapidMiner system (see [3]). RapidMiner provides a nice GUI to specify the input and the configuration for vector creation. In the following, it is assumed that you are familiar with the basic concepts of the RapidMiner environment.

3.1 Installation

The *WVTool* Plugin is installed by downloading the word vector plugin jar file from the RapidMiner homepage[1] and putting it into lib/plugins directory of your RapidMiner installation (see the RapidMiner manual for details). After the plugin is installed, you see an additional category for operators "Text" in the list of RapidMiner operators.

As a starting point, take a look at the examples of the *WVTool* Plugin, which you will find at the RapidMiner homepage.

3.2 The WVTool Operator

The *WVTool* operator creates an *ExampleSet* from a collection of texts. The output *ExampleSet* contains one row for each text document and one column for each term.

The text collection must be specified in one of two ways:

[1] http://www.rapid-i.com

1. If the parameter list *texts* is specified, each key-value pair must contain the class label and the directory which holds the texts. In this case, the entries in *default_ encoding*, *default_ language* and *default_ type* are used for all input documents.

2. Otherwise the operator expects an *ExampleSet* as input. Up to four regular attributes of this example set having special names and the label are evaluated (see 2.2):

   (a) *document_ source* - A file, directory, or URL specifying a (set of) text(s)

   (b) *type* - The document type

   (c) *encoding* - The content encoding

   (d) *language* - The content language

   (e) *the label attribute* - The class label of the text(s)

The parameters *loader*, *inputfilter*, *charmapper*, *tokenizer*, *wordfilter*, *stemmer* and *vectorcreation* specify implementations that perform the respective steps (see 2.3). Within RapidMiner, only static configuration is possible.

3.3 Text Classification, Clustering and Visualization

As word vectors are stored in RapidMiner *ExampleSet*, you can use them in almost any kind of RapidMiner experiment. For text classification, the class labels (e.g. positive, negative) are defined in the *WVTool* operator, as described above. Using clustering or dimensionality reduction, there is a possibility to directly visualize text documents from the RapidMiner Visualization panel. Just double click on an item and a window pops up containing the corresponding text. This is very useful, e.g. for outlier detection.

3.4 Parameter Optimization

As part of a RapidMiner experiment, you can optimize the parameters of vector creation, such as the stemming algorithm or the pruning criteria. To do this, simply surround the *WVTool* operator by a parameter optimization chain and perform some evaluation within this chain, e.g. text classification.

3.5 Creating and Maintaining Word Lists

For many applications it is useful to create and maintain word lists (and thus the dimensions of the vector space) manually. The RapidMiner operator *InteractiveAttributeWeighting* in combination with the *WVTool* and *CorpusBasedWeighting* provides this functionality.

3.5.1 Creating an Initial Word List

An initial word list can be created by using the following chain of operators: *WVTool*, *CorpusBasedWeighting* and *InteractiveAttributeWeighting*. The *WVTool* creates a initial word list. The *CorpusBasedWeighting* operator weights every term in this list with respect to its relevance to the class label given as parameter. The weight for a given term is calculated by summing up the (tf/idf) weights for this term over all documents in the class. The objective of this method is to give terms a high weight, that are important for a specific class. Using tf/idf the other classes can be used as background knowledge about how important a term is in the whole corpus (though the operator can be used with one class only). As the *InteractiveAttributeWeighting* operator is reached a window pops up that shows the word list. You can click on the bar above the table to sort the terms either by their weight or alphabetically. Use the buttons beside every term to select the keywords (by setting their weight to one or zero). After you finished store the word list with the save button. The resulting file contains lines of the following format:

<term>: <weight>

Hint: If you sort the terms according to their weight you can finish your selection if you think that no relevant terms will appear below in the list.

3.5.2 Applying a Word List

You can apply a word list in two ways: To use the actual weights, first create word vectors using the *WVTool* Operator and then use the *AttributeWeightsLoader* and *AttributesWeightsApplier* on the resulting *ExampleSet*. To use the word list only as a selection of relevant terms and leave it to the *WVTool* to actually weight them, use the *AttributeWeightsLoader* before the *WVTool*. The *WVTool* will create vectors that contain as dimensions only terms in the word list, that have a weight larger than zero.

3.5.3 Updating a Word List

If you add new documents to your corpus, usually additional terms will be relevant and should be added to the word list. Use the experiment to create a word list described in 3.5.1. After the *InteractiveAttributeWeighting* operator pops up, use the load function to load your original word list. Make sure that the "overwrite" parameter is set. In this way, values from the file will overwrite the ones that are generated by the *WVTool*. All terms for which you already decided that they should or should not be in the word list are preserved. All new terms will be between these values in the list (sorted according to their weight).

You can also use the combo box to choose which weights should be displayed. After you finished simply save the word list as described above.

Chapter 4

Advanced Topics

4.1 Web Crawling

The *WVTool* contains an interface to the WebSPHINX web crawler package [7]. This enables you to obtain word vectors from webcontent easily. The WebSPHINX package is very flexible and allows to configure the behavior of the crawler in various ways. To use it with the *WVTool*, you must first create a subclass of the abstract class *WVToolCrawler*. The additional methods you must implement determine whether a link should be visited and whether a page should be processed by the *WVTool*. The following is an example.

```
WVToolCrawler test = new WVToolCrawler() {

  protected boolean vectorizePage(Page page) {

    String url = page.getURL().toExternalForm();
    return url.contains("PERSONAL")&&
           url.contains("html")&&
           (!url.contains("index"));
  }

  public boolean shouldVisit(Link link) {
    return link.getPageURL().
           toExternalForm().contains("PERSONAL");

  }
};

URL start = new URL("http://www-ai.cs.uni-dortmund.de/PERSONAL");
```

```
  test.addRoot(new Link(start));
  test.setMaxDepth(2);
```

The crawler visits only links, that point to an URL containing the term "PERSONAL". A page is processed if its URL contains "PERSONAL" and "html" but does not contain "index". The crawler starts at a page provided by the addRoot method. Also, the maximal depth of the crawler is set to 2. There are many other possible checks in the WebSPHINX package, e.g. based on regular expressions. Refer to the javadoc of WebSPHINX for more information.

Given the personalized web crawler, you need to create an input list based on this crawler using the following code:

```
WVTInputList list = new CrawledInputList(test);
```

You can now use this input list just as the file input list.

The crawler can also be invoked from RapidMiner.

To do so, add the *Crawler* operator to your experiment. Using the parameter *url*, you may define a at which url the crawler starts.

The crawler policy allows you to state rules, on whether the crawler should follow a link and on whether it should vectorize a page. The following conditions are possible:

visit_url A page is only visited if its url contains all terms stated in this parameter.

visit_content A page is only visited if its content contains all terms stated in this parameter.

follow_url A link is only followed, if the target url contains all terms stated in this parameter.

link_text A link is only followed, if the link text contains all terms stated in this parameter.

If several expressions are given for the same condition, they are treated a disjunction. This allows to express DNF expressions for each individual condition. Conditions of different types are combined by conjunction, i.e. all of the have to be fulfilled.

4.2 Using a Thesaurus

4.2.1 Using a Simple Dictionary

Instead of using a generic stemmer, you can provide the *WVTool* with a file that explicitly states which words should be reduced to which base forms. You may, for instance, specify that "2000" and "2K" should be both reduced to the same term. Another example is that you would like to replace all numbers in the text by the term "<number>".

The *DictionaryStemmer* allows you to apply such rules easily. It expects as a file as input in which each line has the following format:

```
<base_form>, <expression1> <expression2> ... <expressionn>
```

An expression is either a String or a regular expression. For regular expressions, the Java RegExpression semantic is used. The system first matches a word against the fixed terms specified in the file. If there are different matches, the first one is used. If no match was found, the system checks the word against all regular expressions in the order in which they appear in the file. Again, the first match is used.

4.2.2 Using Wordnet

The *WVTool* contains an interface to the popular Wordnet thesaurus [9] using the Java Wordnet Library (JWNL)[8]. Using a thesaurus has several benefits for text processing. It is, for instance, possible to map words with same meaning to a single term. It might also make sense to replace words a hypernym, e.g. "monday" by "weekday".

To use Wordnet with the *WVTool*, you need a working installation of Wordnet 2.1[1]. Also, you need a configuration file for JWNL. An example configuration file can be found in the sample directory. Usually it should be sufficient to set the correct path to your Wordnet dictionary directory (setting the parameter *dictionary_path*). For more information on configuring the JWNL please refer to their homepage.

Currently, Wordnet is supported for the use in the stemmer step, thus to reduce a word to some base form. The corresponding classes are called *WordNetHypernymStemmer* and *WordNetSynonymStemmer*. Both first resolve the synset of the given word. As the part of speech is usually not known, the Word Vector

[1] Can be obtained from [9]

Tool tries to resolve it first as noun, then as verb, adjective and adverb. For the stemmer based on synonyms, the word is reduced to the first representative of the synset, for hypernym based stemming it is reduced to the first hypernym of the synset.

4.2.3 Information Extraction

The *WVTool* is not intended to be a sophisticated information extraction system. However it allows to state simple, but powerful queries to obtain structured information from (semi-) structured data. Note, that this functionality is only available in the RapidMiner version of the *WVTool*.

The tool supports two basic ways to extract information:

1. by regular expressions
2. by XPath queries

The latter one can only be applied to XML and HTML documents.

Extracting Information with Regular Expressions

A regular expression matches against a parts of an input text. In the *WVTool*, you specify the regular expressions using the parameter list *attributes*. Each line contains an attribute name and a regular expression. The attribute name can be freely chosen. If you put an # in front of the attribute name, the attribute will be interpreted as numerical. In this case, several heuristics are used to extract a number from the string that is matched. The second column contains the regular expression. All regular expressions must follow the pattern "<regex> <replacementPattern>". The <regex> is just a standards regular expression. It is matched against the input text and only the first match is returned. The replacement pattern specifies, how the final term is derived from the matched expression. It should contain at least one expression of the form $<groupNr>, that is replaced by the corresponding matching group. In the simplest case, the replacement string is just $0, stating that the whole expression should be used. Example: If the documents contains the text "Amount: 5", the expression "Amount: ([0-9]+) $1" would extract the value 5.

By default, structured information *and* word vectors are extracted. If you want to use only extracted attributes, specify a *min_occurrences* that is higher than the number of input documents to avoid that word vectors are created.

An additional hint, you can use the preview function to interactively deploy your queries.

Extracting Information with XPath

While regular expressions are quite powerful on plain text, for information that is highly structured, there are often more appropriate solutions. XPath is a query language for xml documents. You can use XPath queries instead of regular expressions at all points in the *WVTool* (in which case you do not need a replacement pattern). They are recognized as XPath, as they start with a "/".

A common source of problems with XPath are namespaces. If your source xml makes use of namespaces, you have to use them in your XPath expressions as well (even if only a single namespace is used all over the document). You can specify namespaces in the *namespaces* attribute by pairs of identifiers (that you then use in the XPath expression) and the namespace as defined in the xml document.

Just as for regular expressions you can specify attributes as numerical by using # as prefix. Expressions as "3,4 Euro" are parsed automatically (and yield 3.4 in this case).

Selecting the Text to Vectorize

You can use regular expressions and XPath also to specify which parts of the document should be used for word vectorization (e.g. to only select the textual description on a webpage). You do this specifying a regular expression or XPath expression in the parameter *text_query*. The syntax is the same as for extracting attributes. The only difference is, that all matches are used and concatenated, instead of using only the first one.

Accessing Webservices

Many information sources on the web are available through a WebService API. The *MashUp* Operator allows you to enrich an existing example set with additional attributes obtained from such a WebService. The most important parameter of this operator is *url*. In this parameter you specify the url under which the service can be accessed. Most importantly, this url may contain expressions of the form "<"<attribute>">". These expressions are replace by the value for the attribute for each example in the example set. For each example in the example set, one query is send to the WebService in this way. The result for each query is parsed and the attributes specified in the parameter *attributes* are extracted and added to the example. The syntax for the extraction of attributes is the same as in the *WVTool*. Again, be careful about namespaces!

A special function of the *MashUp* Operator is, that it allow to use the same query twice. In this case, the result of the query is tokenized using the delimiters defined in the parameter *delimiters* and the tokens are assigned to the attributes using this query. This allows to parse expressions like <position>12,4;34,3</position> into two attributes.

Chapter 5

Performance

The *WVTool* has been designed and optimized for flexibility and extendibility rather than for efficiency. Nevertheless, it is well suited for large text corpora in the sense that it keeps only the word list and the currently processed text document in main memory. To give you an idea of the actual processing speed of the Word Vector Tool the following table shows the processing times for vectorizing the well known 20 newsgroups [6] data set, containing 20.000 news articles.

	WVTool	WVTool (RapidMiner)
word list creation	138 s	-
word vector creation	341 s	-
both	479 s	642 s

For these experiments an Intel P4 with 2,6 GHz was used. For vector creation the word list was pruned to contain only words appearing between 4 and 300 times.

Chapter 6

Aknowledgements

I would like to thank Ingo Mierswa and Simon Fischer for the first version of the *WVTool* operator and the corresponding documentation, Stefan Haustein for the TagIgnoringReader and the creators of the Snowball stemmer package[4], KEA[5], Wordnet, PDFBox, FontBox, the Java Wordnet Library and WebSPHINX for making their source code publically available.

Bibliography

[1] G. Salton, A. Wong, C. S. Yang: A vector space model for automatic indexing, Commun. ACM, 18, p. 613-620, 1975.

[2] R. Baeza-Yates, B. Ribeiro-Neto: Modern Information Retrieval; Taschenbuch - 464 Seiten - Addison Wesley, 1999.

[3] I. Mierswa and M. Wurst, R. Klinkenberg, M. Scholz and T. Euler. YALE: Rapid Prototyping for Complex Data Mining Tasks. In Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-06).

[4] http://snowball.tartarus.org/

[5] http://www.nzdl.org/Kea/

[6] http://kdd.ics.uci.edu/databases/20newsgroups/20newsgroups (originally donated by T. Mitchell)

[7] http://www.cs.cmu.edu/ rcm/websphinx/

[8] http://jwordnet.sourceforge.net

[9] http://wordnet.princeton.edu

[10] A.K. McCallum: Bow: A toolkit for statistical language modeling, text retrieval, classification and clustering, http://www.cs.cmu.edu/~mccallum/bow, 1996.

[11] H. Cunningham, K. Humphreys, Y. Wilks, R. Gaizauskas: Software Infrastructure for Natural Language Processing, Proceedings of the Fifth Conference on Applied Natural Language Processing (ANLP-97), 1997.

Chapter 7

Appendix A - Java Example

The following is a complete example of how to invoke the *WVTool* from Java.

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileWriter;
import java.util.List;
import java.util.Vector;
import edu.udo.cs.wvtool.config.WVTConfiguration;
import edu.udo.cs.wvtool.config.WVTConfigurationFact;
import edu.udo.cs.wvtool.generic.output.WordVectorWriter;
import edu.udo.cs.wvtool.generic.stemmer.DummyStemmer;
import edu.udo.cs.wvtool.generic.vectorcreation.TFIDF;
import edu.udo.cs.wvtool.generic.vectorcreation.TermOccurrences;
import edu.udo.cs.wvtool.main.WVTDocumentInfo;
import edu.udo.cs.wvtool.main.WVTInputList;
import edu.udo.cs.wvtool.main.WVTWordVector;
import edu.udo.cs.wvtool.main.WVTool;
import edu.udo.cs.wvtool.wordlist.WVTWordList;

/**
* An example program on how to use the Word Vector Tool.
*
* @author Michael Wurst
*
*/

public class WVToolExample {
```

```
public static void main(String[] args) throws Exception {

 // EXAMPLE HOW TO CALL THE PROGRAM FROM JAVA

 // Initialize the WVTool

 WVTool wvt = new WVTool(true);

 // Initialize the configuration

 WVTConfiguration config = new WVTConfiguration();
 config.setConfigurationRule(WVTConfiguration.STEP_STEMMER,
    new WVTConfigurationFact(new DummyStemmer()));

 //Initialize the input list with two classes

 WVTFileInputList list = new WVTFileInputList(2);

 //Add entries

 list.addEntry(
    new WVTDocumentInfo("data/alt.atheism",
                        "txt","","english",0));
 list.addEntry(
    new WVTDocumentInfo("data/soc.religion.christian",
                        "txt","","english",1));

 // Generate the word list

 WVTWordList wordList = wvt.createWordList(list, config);

 // Prune the word list

 wordList.pruneByFrequency(2, 5);

 // Store the word list in a file

 wordList.storePlain(new FileWriter("wordlist.txt"));

 // Alternatively: read an already created word list from a file
 // WVTWordList wordList2 =
 // new WVTWordList(
 //    new FileReader("/home/wurst/tmp/wordlisttest.txt"));
 // Create the word vectors
```

```
  // Set up an output filter (write sparse vectors to a file)

  FileWriter outFile = new FileWriter("wv.txt");
  WordVectorWriter wvw = new WordVectorWriter(outFile, true);

  config.setConfigurationRule(
     WVTConfiguration.STEP_OUTPUT,
     new WVTConfigurationFact(wvw));


  config.setConfigurationRule(WVTConfiguration.STEP_VECTOR_CREATION,
      new WVTConfigurationFact(new TFIDF()));

  // Create the vectors

   wvt.createVectors(list, config, wordList);

  // Alternatively: create word list and vectors together
  //wvt.createVectors(list, config);

  // Close the output file

  wvw.close();
  outFile.close();

  // Just for demonstration: Create a vector from a String

  WVTWordVector q =
    wvt.createVector("cmu harvard net", wordList);

 }

}
```

Chapter 8

Appendix B - RapidMiner Operator Reference

This chapter describes the Word Vector operators of the *WVTool* RapidMiner plugin.

8.1 Text

This section describes the text related operators of the *WVTool* plugin.

8.1.1 AttributeSumClusterCharacterizer

Group: Text.Misc

Required input:

- ExampleSet
- ClusterModel

Parameters:

○ **k:** the maximal number of keywords (integer; 1-+∞; default: 3)

Values:

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Characterizes clusters by selecting the attributes with the highest sum of attribute values in each cluster.

Description: Operator that calculates characteristic terms for each cluster of a cluster model by adding up feature values in each cluster and then selecting the features with the highest sum value.

8.1.2 Segmenter

Group: Text.Misc

Parameters:

- **texts:** A directory containing the documents to segment (filename)
- **output:** The directory to which to write the segments (filename)

- **expression:** Specifies a regular expression or XPath expression that matches against substrings of the content which should be treated as individual segments. The syntax is the same as for attribute extraction (see WVTool operator), but instead of extracting only the first match, all matches are extracted and written to individual files (string)
- ○ **namespaces:** Specifies pairs of identifier and namespace for use in XPath queries. The namespace for (x)html is bound automatically to the identifier h. (list)

Values:

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Segments documents.

Description: Operator that allows to extract segments from a set of documents based on regular expressions or xpath.

8.1.3 SingleText2WordVector

Group: Text

Generated output:

- ExampleSet

Parameters:

- **text:** The input text. (string)
- ○ **default_content_type:** The default content type if not specified by the example set. (string; default: ”)
- ○ **default_content_encoding:** The default content encoding if not specified by the example set. (string; default: ”)
- ○ **default_content_language:** The default content language if not specified by the example set. (string; default: ”)
- ○ **inputfilter:** Implementation class for step inputfilter.
- ○ **charmapper:** Implementation class for step charmapper.

- **tokenizer:** Implementation class for step tokenizer.
- **wordfilter:** Implementation class for step wordfilter.
- **stemmer:** Implementation class for step stemmer.
- **vectorcreation:** Implementation class for step vectorcreation.
- **wvt_configuration:** If the simple configuration specified by loader, inputfilter, ... does not suffice, an implementation of WVTConfiguration may be specified here. (string)
- **prune_below:** Prune words that appear inat most that many documents. -1 for no pruning. Alternatively you can provide a percentage value, denoting the lowest document frequency in p words with the highest frequency. (string; default: '-1')
- **prune_above:** Prune words that appear in at least that many documents. -1 for no pruning. Alternatively you can provide a percentage value, denoting the highest document frequency in p words with the lowest frequency. (string; default: '-1')
- **min_chars:** The minimum number of characters a word must contain to be processed (-1 for any). Note that this parameter works only with word filters derived from AbstractStopWordFilter (integer; 0-+∞; default: 4)
- **add_stopwords:** A file that contains additional stopwords (one per line). (filename)
- **ngrams:** If this value is larger than zero, the operator creates character ngrams of the specified size. (integer; 0-+∞; default: 0)
- **use_content_attributes:** If set to true, the returned example set will contain content type, encoding, and language attributes. (boolean; default: false)
- **input_word_list:** Load a word list from this file instead of creating it from the input data. (filename)
- **output_word_list:** Save the used word list into this file. (filename)
- **id_attribute_type:** Indicates if long ids (complete paths), short ids (last part of the source name), or numerical ids will be used.
- **attributes:** Specifies a list of attribute names and extraction expressions. Extraction expressions can be regular expressions or XPath expressions. A number sign in front of an attribute name marks the attribute as numeric. Refer to the tutorial or the samples for further information. (list)
- **namespaces:** Specifies pairs of identifier and namespace for use in XPath queries. The namespace for (x)html is bound automatically to the identifier h. (list)
- **text_query:** Query that extracts the parts of a document, that should be used for vectorization (string)

- ○ **create_text_visualizer:** Indicates if a text specific object visualizer should be created which can be used in plotters etc. (boolean; default: false)

Values:

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Generates word vectors from a single text.

Description: This operator takes a a single input text via a parameter. The result is a set of attributes representing word occurrence information from the text. The set of words (attributes) is determined from the given text or could be determined by an input word list. The parameters are the same as for the WVToolOperator.

This operator is especially useful if you want to process a single text. If you want to read more than one text directly from files we recommend the WVToolOperator of the Word Vector Tool plugin.

8.1.4 StringToWordVector

Group: Text

Required input:	**Generated output:**
• ExampleSet	• ExampleSet

Parameters:

- ○ **filter_nominal_attributes:** Indicates if nominal attributes should also be filtered in addition to string attributes. (boolean; default: false)
- ○ **default_content_type:** The default content type if not specified by the example set. (string; default: ”)
- ○ **default_content_encoding:** The default content encoding if not specified by the example set. (string; default: ”)
- ○ **default_content_language:** The default content language if not specified by the example set. (string; default: ”)
- ○ **inputfilter:** Implementation class for step inputfilter.

- **charmapper:** Implementation class for step charmapper.
- **tokenizer:** Implementation class for step tokenizer.
- **wordfilter:** Implementation class for step wordfilter.
- **stemmer:** Implementation class for step stemmer.
- **vectorcreation:** Implementation class for step vectorcreation.
- **wvt_configuration:** If the simple configuration specified by loader, inputfilter, ... does not suffice, an implementation of WVTConfiguration may be specified here. (string)
- **prune_below:** Prune words that appear inat most that many documents. -1 for no pruning. Alternatively you can provide a percentage value, denoting the lowest document frequency in p words with the highest frequency. (string; default: '-1')
- **prune_above:** Prune words that appear in at least that many documents. -1 for no pruning. Alternatively you can provide a percentage value, denoting the highest document frequency in p words with the lowest frequency. (string; default: '-1')
- **min_chars:** The minimum number of characters a word must contain to be processed (-1 for any). Note that this parameter works only with word filters derived from AbstractStopWordFilter (integer; 0-+∞; default: 4)
- **add_stopwords:** A file that contains additional stopwords (one per line). (filename)
- **ngrams:** If this value is larger than zero, the operator creates character ngrams of the specified size. (integer; 0-+∞; default: 0)
- **use_content_attributes:** If set to true, the returned example set will contain content type, encoding, and language attributes. (boolean; default: false)
- **input_word_list:** Load a word list from this file instead of creating it from the input data. (filename)
- **output_word_list:** Save the used word list into this file. (filename)
- **id_attribute_type:** Indicates if long ids (complete paths), short ids (last part of the source name), or numerical ids will be used.
- **attributes:** Specifies a list of attribute names and extraction expressions. Extraction expressions can be regular expressions or XPath expressions. A number sign in front of an attribute name marks the attribute as numeric. Refer to the tutorial or the samples for further information. (list)
- **namespaces:** Specifies pairs of identifier and namespace for use in XPath queries. The namespace for (x)html is bound automatically to the identifier h. (list)

- **text_query:** Query that extracts the parts of a document, that should be used for vectorization (string)
- **create_text_visualizer:** Indicates if a text specific object visualizer should be created which can be used in plotters etc. (boolean; default: false)

Values:

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Generates word vectors from string attributes.

Description: This operator takes an input example set and uses the values of the string attributes as texts. String attributes are attributes with value type “string”. The result is a set of attributes representing word occurrence information from the text contained in the strings. The set of words (attributes) is determined from the given data set. The parameters are the same as for the WVToolOperator.

This operator is especially usefull if you already have text data represented in a single example set file, e.g. an Arff file containing string attributes or other example sets defining such string attributes. If you want to read text data directly from files we recommend the WVToolOperator of the Word Vector Tool plugin.

8.1.5 WVTool

Group: Text

Generated output:

- ExampleSet

Parameters:

- **configure_operator:** Configure this operator by means of a Wizard. (null)
- **preview:** Shows a preview for the results which will be achieved by the current configuration. (null)
- **texts:** Specifies a list of class/directory pairs. (list)
- **default_content_type:** The default content type if not specified by the example set. (string; default: ”)

- ○ **default_content_encoding:** The default content encoding if not specified by the example set. (string; default: ")
- ○ **default_content_language:** The default content language if not specified by the example set. (string; default: ")
- ○ **loader:** Implementation class for step loader.
- ○ **inputfilter:** Implementation class for step inputfilter.
- ○ **charmapper:** Implementation class for step charmapper.
- ○ **tokenizer:** Implementation class for step tokenizer.
- ○ **wordfilter:** Implementation class for step wordfilter.
- ○ **stemmer:** Implementation class for step stemmer.
- ○ **vectorcreation:** Implementation class for step vectorcreation.
- ○ **wvt_configuration:** If the simple configuration specified by loader, inputfilter, ... does not suffice, an implementation of WVTConfiguration may be specified here. (string)
- ○ **prune_below:** Prune words that appear inat most that many documents. -1 for no pruning. Alternatively you can provide a percentage value, denoting the lowest document frequency in p words with the highest frequency. (string; default: '-1')
- ○ **prune_above:** Prune words that appear in at least that many documents. -1 for no pruning. Alternatively you can provide a percentage value, denoting the highest document frequency in p words with the lowest frequency. (string; default: '-1')
- ○ **min_chars:** The minimum number of characters a word must contain to be processed (-1 for any). Note that this parameter works only with word filters derived from AbstractStopWordFilter (integer; 0-+∞; default: 4)
- ○ **add_stopwords:** A file that contains additional stopwords (one per line). (filename)
- ○ **ngrams:** If this value is larger than zero, the operator creates character ngrams of the specified size. (integer; 0-+∞; default: 0)
- ○ **use_content_attributes:** If set to true, the returned example set will contain content type, encoding, and language attributes. (boolean; default: false)
- ○ **input_word_list:** Load a word list from this file instead of creating it from the input data. (filename)
- ○ **output_word_list:** Save the used word list into this file. (filename)
- ○ **id_attribute_type:** Indicates if long ids (complete paths), short ids (last part of the source name), or numerical ids will be used.

- **attributes:** Specifies a list of attribute names and extraction expressions. Extraction expressions can be regular expressions or XPath expressions. A number sign in front of an attribute name marks the attribute as numeric. Refer to the tutorial or the samples for further information. (list)
- **namespaces:** Specifies pairs of identifier and namespace for use in XPath queries. The namespace for (x)html is bound automatically to the identifier h. (list)
- **text_query:** Query that extracts the parts of a document, that should be used for vectorization (string)
- **create_text_visualizer:** Indicates if a text specific object visualizer should be created which can be used in plotters etc. (boolean; default: false)

Values:

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Generates word vectors from text collections.

Description: This operator wraps the Word Vector Tool by Michael Wurst, creating an ExampleSet from a collection of texts. The output example set will contain one row for each text and one column for each word (or for each word stem). The text collection must be specified in one of two ways.

- If the parameter list *texts* is specified, each key-value pair must contain the class (e.g. "positive", "negative", "interesting" etc.) and the value must be a directory which holds the texts of this class.
- Otherwise the operator expects an ExampleSet in its input. Up to four regular attributes of this example set having special names and the label are evaluated:

  "document_source" A file, directory, or URL specifying a (set of) text(s)

  "type" The document type, e.g. xml or pdf

  "encoding" The content encoding

  "language" The content language

  The label attribute The class of the text(s)

The parameters *loader*, *inputfilter*, *charmapper*, *tokenizer*, *wordfilter*, *stemmer*, and *vectorcreation* specify implementations that perform the respective step.

8.2 Web and Webservices

This section describes the web-related operators of the *WVTool* plugin.

8.2.1 Crawler

Group: Web

Generated output:

- ExampleSet

Parameters:

- **url:** Specifies the url at which the crawler should start (string)
- ○ **crawling_rules:** Specifies a set of rules that determine, which links to follow and which pages to process (see tutorial for details) (list)
- ○ **max_depth:** Specifies the maximal depth of the crawling process (integer; 0-+∞; default: 2)
- ○ **delay:** Specifies the delay when vistiting a page in milleseconds (integer; 0-+∞; default: 1000)
- ○ **max_threads:** Specifies the number of crawling threads working in parallel (integer; 1-+∞; default: 1)
- **output_dir:** Specifies the directory to which to write the files (filename)
- ○ **user_agent:** The identity the crawler uses while accessing a server (string; default: 'yale-crawler')
- ○ **obey_robot_exclusion:** Specifies whether the crawler obeys the rules, which pages on site might be visited by a robot. Disable only if you know what you are doing and if you a sure not to violate any existing laws by doing so (boolean; default: true)

Values:

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Crawls a set of web resources and writes them to a local directory.

Description: This operator crawls the net and stores pages to a local directory. It allows to specify diverse rules to determine which links to visit and which pages to store. See the the WVTool Tutorial for details. Notice: This operator is currently in an experimental state, use with care.

8.2.2 MashUp

Group: Web

Required input:

- ExampleSet

Parameters:

- ○ **attributes:** Specifies a list of attribute names and extraction expressions. Extraction expressions can be regular expressions or XPath expressions. A number sign in front of an attribute name marks the attribute as numeric. Refer to the tutorial or the samples for further information. (list)
- ○ **namespaces:** Specifies pairs of identifier and namespace for use in XPath queries. The namespace for (x)html is bound automatically to the identifier h. (list)
- ○ **url:** The url of the HTTP GET based service. This URL may contain terms of the form <attributeName> that are replaced by the value of the corresonding attribute before invoking the query. (string)
- ○ **separators:** Characters used to separate entries in the result field obtained by XPath or regular expression. (string)
- ○ **delay:** Amount of milliseconds to wait between requests (integer; 0-+∞; default: 0)

Values:

- **applycount:** The number of times the operator was applied.
- **looptime:** The time elapsed since the current loop started.
- **time:** The time elapsed since this operator started.

Short description: Extracts information from a HTTP GET based web resource.

Description: Operator that enriches an example set with attributes gathered from a web resource. The web resource must be accessible via HTTP GET and return an XML tree as result. The user specifies a generic URL of the resource (that may contain example dependent values) and a list of attribute names with a corresponding xpath expression. The attributes are then assigned the value to which the corresponding expression is evaluated. To support comma or space separated values in the XML tree it is allowed to use the same query more than once. The first attribute in the list of attributes sharing the same query is assigned the first value, the second attribute the second and so on. Separators can be specified explicitely.