Documentation of the Information Extraction Plugin for RapidMiner

Felix Jungermann
Technical University of Dortmund
Department of Computer Science - Artificial Intelligence Group
Baroper Strasse 301, 44227 Dortmund, Germany

September 14, 2011

Chapter 1

The Information Extraction Plugin for RapidMiner

In this chapter we will present the extension we developed for the well-known open source framework *RapidMiner* [Mierswa et al., 2006]. The installation of the plugin is presented in Section 1.1. The architecture of the plugin and a developer's guide are presented in Sections 1.2 and 1.3. *RapidMiner* – which is shortly presented in Section 1.4 – works with a certain data structure for storing datasets. This data structure has to be respected by our extension leading to particular requirements. These requirements in addition to the data structure used in *RapidMiner* are presented in Section 1.4.1.

The process of a particular data mining task can be separated in four distinct parts in *RapidMiner*. The first part is the retrieval of the data. We present the possible ways to retrieve data for information extraction purposes in *RapidMiner* in Section 1.5.1. After the retrieval the data has to be prepared for future use. This preparation often is called preprocessing. Although the task of preprocessing sometimes contains the process of data preparation and data cleansing, we just focus on the enrichment of the data by features to allow more precise analyses. The preprocessing of the datasets is presented in Section 1.5.2. The preprocessed datasets finally can be used to create models which in turn can be used to analyse formerly unknown datasets. The process of creating models is called modelling and it is presented in Section 1.5.3. After – and sometimes also during – the process of modelling the models have to be evaluated to get the optimal model for the given datasets. The task of evaluation is presented in Section 1.5.4.

The samples which are provided in the *./<Information Extraction Plugin>/samples* folder are presented in Section 1.6. Section 1.7 summarizes this chapter. The particular reference for each operator is presented in Appendix A.

1.1 Install

The *Information Extraction Plugin* is an extension to the open source framework *RapidMiner*. To install the *Information Extraction Plugin* you first of all have to install *RapidMiner*. For downloading *RapidMiner* visit
`http://sourceforge.net/projects/rapidminer/`

After having installed *RapidMiner* get the latest version of the *Information Extraction Plugin* at `http://sourceforge.net/projects/ieplugin4rm/`. Download the precompiled `rapidminer-Information-Extraction-*.*.*.jar`-file and move it to `<RapidMiner-home>/lib/plugins/.`

If you start *RapidMiner* afterwards you will be able to use the *Information Extraction Plugin*.

1.1.1 Build

The *Information Extraction Plugin* comes with an *ant*-script which allows the automatic compilation of the source files. Edit the *build.xml* and change the line `<property name="rm.dir" location="../RapidMiner_Vega" />`. The last part `location="../RapidMiner_Vega"` should point to the folder *RapidMiner* is located on your machine.

In the directory containing the *Information Extraction Plugin* you have to execute `ant createJar` which will compile the sources and create a *jar*-file containing all build sources. The *jar*-file is automatically copied into `<RapidMiner-home>/lib/plugins/.`. After starting *RapidMiner* the newly created plugin is available.

1.2 Architecture

The architecture of the *Information Extraction Plugin* is presented in Figure 1.1. The complete figure represents the complete process in *RapidMiner*, whereas the light blue area presents the operations done by the *Text Plugin* and the dark blue area contains the operations done by the *Information Extraction Plugin*. The very light blue boxes are operators which are processing exampleset or models. The *Read Document* operator converts documents into examplesets. *Tokenizers* convert these examplesets into examplesets containing smaller parts (tokens). *Preprocessing* operators work on such tokens to create additional attributes. The resulting examplesets can be used by *Modelling* operators to create models which can finally be used to be applied on similar examplesets to extract informational units out of those.

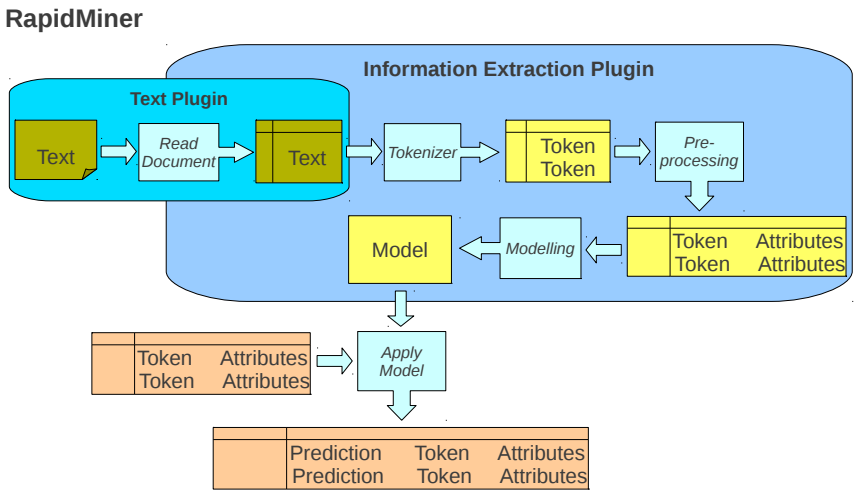

Figure 1.1: Architecture of the *Information Extraction Plugin*

1.3 Developer's Guide

Developers can easily create new operators by implementing classes which extend `com.rapidminer.operator.Operator`. In Appendix A you may find the reference of all operators which are used in the *Information Extraction Plugin*. Some of these operators extend certain abstract classes. Developers can easily implement new operators by extending those classes (for instance

`com.rapidminer.operator.preprocessing.ie.features.tools.PreprocessOperatorImpl`).

New operators have to be provided to

`./resources/com/rapidminer/resources/OperatorsInformationExtraction.xml`

and

`./resources/com/rapidminer/resources/i18n/OperatorsDocInformationExtraction.xml`.

A referencing name of the operator and the complete classpath of the corresponding class have to be put into the first file, like for the *SentenceTokenizer*:

```
<group key="">
  <group key="informationextraction">
    <group key="tokenizers">
      <operator>
        <key>sentence_tokenizer</key>
        <class>

          com.rapidminer.operator.preprocessing.ie.tokenizer.
              SentenceTokenizer

        </class>
      </operator>
      ...
    </group>
    ...
  </group>
  ...
</group>
```

The groups are defining the folder-structure which will contain the operator in the *RapidMiner* operator view.

The second file contains entries referenced with the same key containing the *name* of the operator as it is available later on in *RapidMiner*. In addition, a *synopsis* contains information concerning the certain operator.

```
<operator>
  <name>SentenceTokenizer</name>
  <synopsis>Tokenizes texts in sentences.</synopsis>
  <help />
  <key>sentence_tokenizer</key>
</operator>
```

1.4 RapidMiner

RapidMiner is an open source data mining framework. It offers many operators which can be plugged together into a process. The major function of a process is the analysis of the data which is retrieved at the beginning of the process. The framework offers a well designed graphical user interface (GUI) that allows to connect operators with each other in the process view. Most operators have input and output ports. Data that is passed to an input port of an operator is processed internally and it is presented at the output port, finally. The data which is processed is passed through connected operators. Other types of objects may be created during the process. The data can be used to create models, for instance. These models can be evaluated, and results are generated out of this evaluation process, and so on. These kinds of objects all can be passed

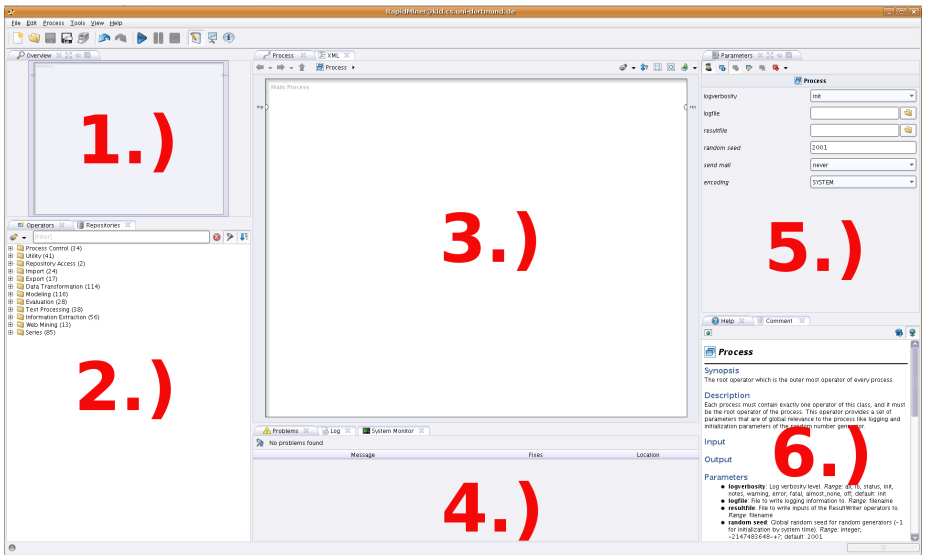

Figure 1.2: RapidMiner graphical user interface

from operators to operators by connecting the operators. The complete process – and the process view – has global output ports. Data, results or models which are passed to these ports are represented after finishing the process.

The GUI of *RapidMiner* is shown in Figure 1.2. Six main areas of the GUI – which can be rearranged by the user – are to be distinguished:

1. **Overview**
   The *Overview* tab delivers a small overview on the complete process window. If the process is too large to be displayed in the process window, the overview window will help to navigate to certain positions in the process window.

2. **Operators and Repositories**
   These tabs allow accessing operators or repositories of *RapidMiner*. Operators are the basic elements for building a process. Repositories store datasets for fast access.

3. **Process**
   The *Process* window contains the whole process. An overview of this window which could become very large is possible using the overview tab.

4. **Problems, Log and System Monitor**
   This tab contains possible log messages, problems and information about the system load.

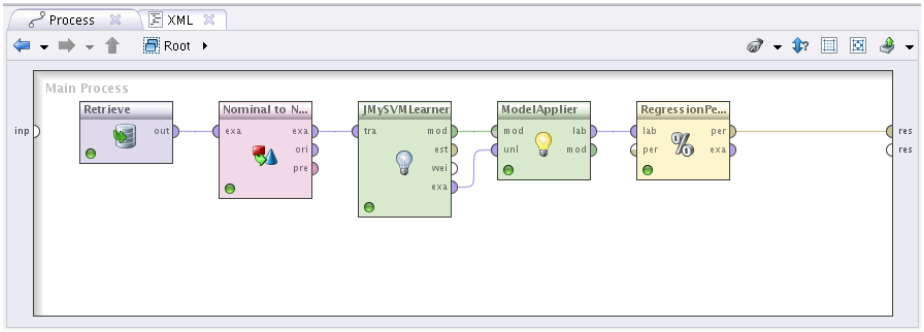

Figure 1.3: Exemplary process in *RapidMiner*

5. **Parameters**
   The *Parameters* tab is very important because it contains the parameters of operators which are currently focussed.

6. **Help**
   The *Help*-tab contains information about focussed operators.

Each *RapidMiner*-process can be split into four distinct phases. These phases are shown in Figure 1.3:

1. **Retrieve**
   The leftmost operator in Figure 1.3 is a *Retrieve*-operator. During the *Retrieve* phase the data which is used for later processing is loaded from specific datasources.

2. **Preprocessing**
   The retrieved data has to be prepared or enriched in the *Preprocessing* phase. The second operator shown in Figure 1.3 (the purple one) is a particular preprocessing operator converting nominal values to numerical ones.

3. **Modelling**
   The prepared data is used in the *Modelling* phase to extract or create models which can be used for analysing former unknown data. The third operator shown in Figure 1.3 is creating an SVM model.

4. **Evaluation**
   The two rightmost operators shown in Figure 1.3 are used to apply the model and to evaluate the perfomance achieved by the applied model. The expected or real performance of the created models is evaluated during the *Evaluation* phase.

These phases nearly stay the same for every *RapidMiner* process. Therefore, we will define the particular phases and the corresponding specialties using the *Information Extraction Plugin* in Section 1.5.

ID	Label	Att_1	Att_2	...	Att_d
1	true	2	'Felix'	...	5.4
2	false	3	'goes'	...	7.1
...	...	...	...	...	...
n	true	1	'Detroit'	...	2.3

Table 1.1: Spreadsheet datastructure internally used by *RapidMiner*

1.4.1 Data Structure

The datastructure in *RapidMiner* is comparable to a spreadsheet like it is used in many spreadsheet programs. The lines of the spreadsheet represent examples and the columns represent attributes. Table 1.1 shows an exemplary dataset containing n examples and d attributes.

It is remarkable that for many data mining task the examples are handled indepedently. It follows that the analysis of a particular example i just depends on the attributes of example i instead of depending on other examples. The structure of documents and texts should be respected for information extraction tasks. CRFs, for instance, process all the tokens of a particular sentence at the same time, and the tokens of the certain sentence condition each other. To be precise: each token is conditioning the following one, so that the sentences itself and the ordering of the tokens of the sentences have to be respected. Another example is the creation of relation candidates which takes all the pairs of entities of one sentence into account. We used the datastructure of *RapidMiner* and we developed mechanisms to respect the circumstances of information extraction tasks.

1.5 Information Extraction Plugin

Like for every data mining task the process for information extraction tasks also can be split in four phases. These phases and the according operators to be used in each phase are described in this section.

1.5.1 Retrieve

The process to retrieve datasets into *RapidMiner* is remarkable for information extraction issues. We present two approaches. The first of these approaches is to be used if the data is available as document files like **PDF**. The second approach is a loading mechanism based on well-known data formats like the one of the CoNLL 2003 shared task on NER[1]. The resulting dataset contains an additional special attribute (*batch*-attribute) which groups the single examples.

[1] `http://www.cnts.ua.ac.be/conll2003/ner/`

ID	batch	Label	Att_1	Att_2	...	Att_d
1	0	true	2	'Felix'	...	5.4
2	0	false	3	'goes'	...	7.1
...	...	...	...	...	...	...
n	m	true	1	'Detroit'	...	2.3

Table 1.2: Spreadsheet datastructure internally used by the Information Extraction Plugin

In addition to the grouping the sequential ordering of the examples is respected by many operators of the plugin. Table 1.2 shows the same dataset as presented in Table 1.1 after being converted into m groups (probably sentences).

Retrieve via Document

It is important to respect the structural circumstances of datasets consisting of textual data as presented in Section 1.4.1. The *Text Mining extension* of *RapidMiner* already offers the possibility to retrieve document structured data. Although the structure of these documents in later steps is destroyed by the *Text Mining extension*, the retrieve mechanism can be used to load documents into *RapidMiner*. The *Text Mining extension* uses a special class for handling documents: the *Document*-class. This class stores the whole document in combination with additional meta-information. In the case of text mining the document is split into unique tokens which are finally used to classify the complete document. For information extraction purposes we would like to tokenize the document and to preserve the order of such tokens, therefore, we implemented tokenizers which are able to process examplesets extracted from the *Document* classes. The application of these tokenizers result in a spreadsheet containing the tokens in the particular order as they have been found in the document. Each token contains a certain number indicating from which general unit it has been created. Each word-token of a particular sentence, for instance, contains the number of the sentence, whereas each sentence-token of a document contains the number of that document. The *Tokenizer*-class can be easily extended to create own tokenizers. Figure 1.4 shows a process containing two operators of the *Text Mining extension* (the two leftmost operators) and two operators (the two rightmost ones) of the *Information Extraction Plugin*. The process loads a document, converts it into an exampleset containing an example which holds the complete document-text and the two tokenizers are splitting the text into multiple tokens (examples). The third operator splits the text into sentences, and the fourth operator splits the sentences into words. After having finished the process the resulting dataset consists of examples holding one word each. Additionally, the words are containing sentence numbers allowing to access all the words of a particular sentence.

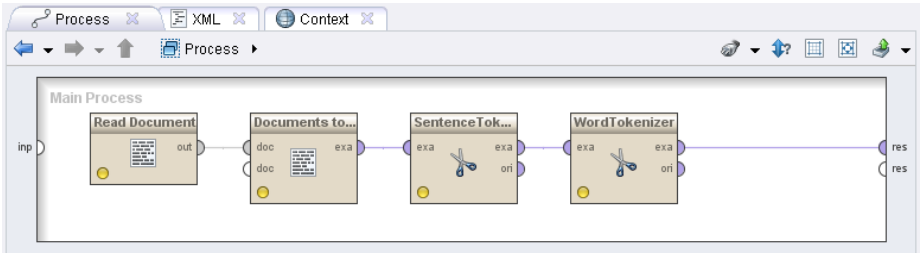

Figure 1.4: Retrieving a document for information extraction in *RapidMiner*

Retrieve via File

Although *RapidMiner* already offers many operators (like the *Read CSV*-operator) to retrieve datasets contained in spreadsheet-like files, the specific structure of document datasets necessitates a certain operator. Datasets like the one for the CoNLL 2003 shared task on NER which is well-known in the NER community are already presented as tokenized documents. Comparable to csv-files the datasets contain a token each line whereas the tokens additionally contain features which are also stored in the particular line. The main difference to ordinary csv-files is the fact that sentences are split by empty lines. It follows that after the tokens for one sentence an empty line is located. Although the *Read CSV*-operator of *RapidMiner* can be adjusted to neglect such lines, we developed an own retrieve operator which respects the empty lines to distinguish between distinct sentences. We therefore use a special attribute containing the sentence numbers. These special attributes later can be used by other operators which work on complete sentences (like CRFs, for instance).

1.5.2 Preprocess

Preprocessing is a very important point in information extraction. In contrast to traditional datamining tasks the data is not given by examples already containing different attributes extracted from a database, for example. The original data just contains tokens consisting of nothing but the token and the contextual tokens itself. The tokens have to be enriched by attributes to get a more general representation. We will distinguish two different types of preprocessing, here. The first type of processing is to enrich tokens for NER. The other type of processing is for enriching relation candidates for relation extraction. Both techniques are presented in the following subsections.

Named Entity Recognition

It is very important to enrich tokens for NER by internal and external information. We developed an abstract class for preprocessing operators that allows to focus on tokens before or after the current token and on the current token itself. The sentence presented in Figure 1.5 is converted into a spreadsheet as

ID	batch	Label	Att_1
1	0	PER	'Felix'
2	0	PER	'Jungermann'
3	0	O	'studied'
4	0	O	'computer'
5	0	O	'sciences'
6	0	O	'at'
7	0	O	'the'
8	0	O	'university'
9	0	O	'of'
10	0	LOC	'Dortmund'
11	0	O	'from'
12	0	O	'1999'
13	0	O	'until'
14	0	O	'2006'
15	0	O	'.'

Table 1.3: Spreadsheet representation of the sentence shown in Figure 1.5

shown in Table 1.3.

Using the abstract class we developed allows accessing contextual tokens in a

Felix Jungermann studied computer sciences at the University of Dortmund from 1999 until 2006.

Figure 1.5: A sentence containing two related entities ('Felix Jungermann' and 'Dortmund'). The relation is indicated by 'to study at'.

relative way. Each token is processed and the abstract class accesses a number of tokens before and after the current token. The number of tokens to access before and after the current token are parameters that can be adjusted by the user. Let the number of tokens surrounding the current token be 2 and the current token shall be example number 10: 'Dortmund'. In that case the preprocessing method would also access the 8th, 9th, 11th and 12th token in addition to the 10th token. The most simple way of preprocessing would be to enrich the current token by the surrounding tokens. Table 1.4 shows the dataset presented in Table 1.3 enriched by two surrounding tokens before and after each token. If a token is at the beginning or at the end of a sentence some of the contextual attributes will contain *null*-values because the tokens of one particular sentence only contain informational units from that sentence.

In addition to the relative contextual tokens to be taken into account another interesting parameter to set for preprocessing operators is the *length*. Some

ID	batch	Label	Att_1	$token_{-2}$	$token_{-1}$	$token_{+1}$	$token_{+2}$
1	0	PER	'Felix'	*null*	*null*	'Jungermann'	'studied'
2	0	PER	'Jungermann'	*null*	'Felix'	'studied'	'computer'
3	0	O	'studied'	'Felix'	'Jungermann'	'computer'	'sciences'
4	0	O	'computer'	'Jungermann'	'studied'	'sciences'	'at'
5	0	O	'sciences'	'studied'	'computer'	'at'	'the'
6	0	O	'at'	'computer'	'sciences'	'the'	'university'
7	0	O	'the'	'sciences'	'at'	'university'	'of'
8	0	O	'university'	'at'	'the'	'of'	'Dortmund'
9	0	O	'of'	'the'	'university'	'Dortmund'	'from'
10	0	LOC	'Dortmund'	'university'	'of'	'from'	'1999'
11	0	O	'from'	'of'	'Dortmund'	'1999'	'until'
12	0	O	'1999'	'Dortmund'	'from'	'until'	'2006'
13	0	O	'until'	'from'	'1999'	'2006'	'.'
14	0	O	'2006'	'1999'	'until'	'.'	*null*
15	0	O	'.'	'until'	'2006'	*null*	*null*

Table 1.4: Dataset of Table 1.3 enriched by contextual tokens

created attributes like *prefixes* or *suffixes*, for instance, have a specific length which has to be adjusted by this parameter.

Annotations Another important point for NER datasets is the labeling of documents and texts. The number of labeled tokens compared to tokens which are not labeled is sparse for most NER datasets. Additionally, the labels are sometimes spread over multiple tokens. Because of these two reasons it should easily be possible to mark specific tokens somewhere in a document directly and to assign a label to the marked tokens. We implemented an annotator operator which allows to display the dataset as a textual document allowing the user on the one hand to create new labels and on the other hand to use those labels for annotating the tokens. After having used that operator the dataset contains a label attribute carrying the annotations and a formerly defined default value if no annotation is given for a particular token. Figure 1.6 shows a screenshot of the annotator screen containing the text presented already in Table 1.3 and 1.4.

Relation Extraction

For relation extraction we developed particular preprocessing operators working especially for relational learning purposes. Although the flat features developed by [Zhou et al., 2005] can be seen as contextual information, the features have to be seen as contextual information relative to a pair of tokens. Respecting these two tokens is much more complex than as it is for the single tokens for NER. In addition to the flat features relation extraction heavily relies on structural information like parse trees, for instance. We developed parsing operators to first of all create tree-structured attributes. Additionally, we implemented pruning methods for the creation of more condensed tree-structures.

Tree-structures can be represented as nominal values like it is shown in Figure 1.8. It would be a computational overhead to parse these nominal values

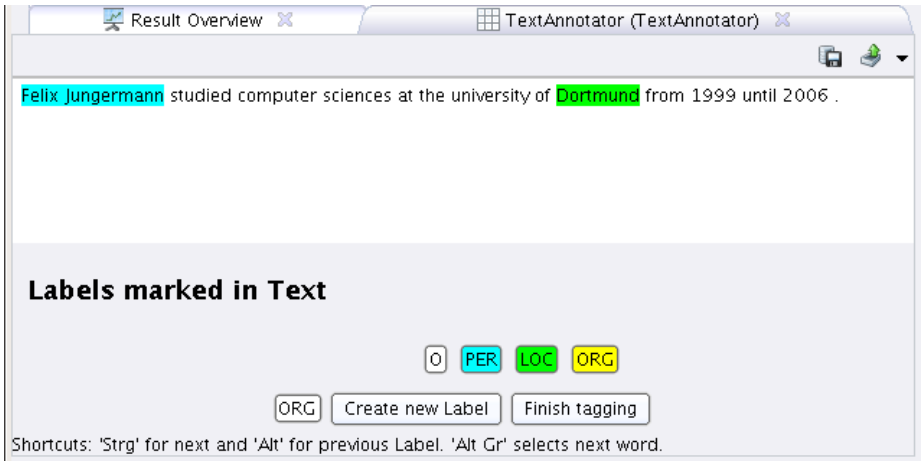

Figure 1.6: Annotating operator for RapidMiner

into tree objects for every time they are needed. We developed a generic form of attribute which allows the storage of every type of *Java*-object. This generic object-attribute can be used to work with tree-structures in *RapidMiner*. Like for nominal values, the object-attribute is storing a mapping which maps numerical values to particular objects.

1.5.3 Modelling

We implemented or embedded all of the techniques we presented in this work in *RapidMiner*. The particular learning methods are CRFs [Lafferty et al., 2001], Tree Kernel SVMs [Collins and Duffy, 2001], Tree Kernel Perceptrons [Aiolli et al., 2007] and Tree Kernel Naïve Bayes Classifier [Jungermann, 2011]. The learning methods already available in *RapidMiner* of course can be used, too.

1.5.4 Evaluation and Validation

Some information extraction tasks need specific evaluation. NER datasets, for instance, sometimes are labeled using the IOB-tagging. During the evaluation of predictions made on a testset the predicted tokens in some evaluation schemes only are considered to be correct if all the single tokens of an entity consisting of multiple tokens are predicted correctly. *George Walker Bush*, for instance, is an entity of type *PERSON* consisting of three tokens. If only one of these three tokens is not predicted to be of type *PERSON*, the complete entity is incorrectly tagged. Although the tokens are represented as single examples, the evaluation has to be done on the entity-mentions which sometimes consist of multiple tokens.

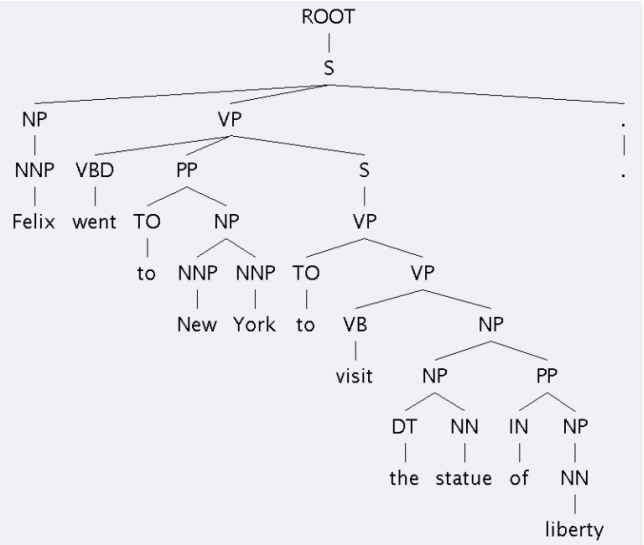

Figure 1.7: The constituent parse tree of the sentence "Felix went to New York to visit the statue of liberty."

(ROOT (S (NP (NNP Felix)) (VP (VBD went) (PP (TO to) (NP (NNP New) (NNP York))) (S (VP (TO to) (VP (VB visit) (NP (NP (DT the) (NN statue)) (PP (IN of) (NP (NN liberty))))))))) (. .)))

Figure 1.8: String representation of the constituent parse tree shown in Figure 1.7

1.6 Samples

The *Information Extraction Plugin* contains a folder storing a number of samples which present the usage of the *Information Extraction Plugin*. To use the samples just create a new *Repository* in *RapidMiner* referencing the folder *./*<Information Extraction Plugin>*/samples/*. You can access the processes stored in the samples-folder via the newly created *Repository* now.

1. *1_Read-Tokenize-Visualize*
   This process shows how to use the *Read Document* operator. Please make sure to have the *Text Plugin* for *RapidMiner* installed. The parameter *file* of the operator *Read Document* should point to *./*<Information Extraction Plugin>*/samples/toyText.txt*. Additionally, the process tokenizes the document into sentences and words. The resulting exampleset is on the one hand visualized and on the other hand it is stored in the repository.

2. *2_Preprocess-Annotate*
   This process shows the use of the *TextAnnotator* operator. The formerly

created exampleset is visualized, and the user can create annotations on the document. The annotated document finally is stored in the repository.

3. *3_VisualizeAnnotated*
   This process presents the annotated exampleset.

4. *4_Learn*
   This process at first enriches the exampleset by contextual information of each token. Afterwards, a cross-validation is performed on the dataset using one half of the exampleset to train a CRF model which is used to predict the other half of the exampleset and vice-versa. The resulting predictions are stored in the repository.

5. *5_VisualizePredicted*
   In this process the predictions and the original labels are visualized. The predictions are shown as the document, and the original labels are used to colorize the predictions. It becomes obvious that two predictions are erroneous.

1.7 Summary

We presented the *Information Extraction Plugin* in this chapter. The plugin is an extension to the well-known open source framework *RapidMiner*. After having presented the graphical user interface of *RapidMiner* we showed that most of the *RapidMiner* processes can be splitted into four particular phases. Compared to traditional datamining tasks we have shown that those phases are also apparent for information extraction tasks. We presented the possibilities to work on these phases for information extraction. Due to the spreadsheet datastructure internally used by *RapidMiner* we developed a representation of datasets for information extraction based on that datastructure. In addition, we implemented several operators helpful and needed for information extraction purposes.

Bibliography

[Aiolli et al., 2007] Aiolli, F., Da San Martino, G., Sperduti, A., and Moschitti, A. (2007). Efficient kernel-based learning for trees. In *Computational Intelligence and Data Mining, 2007. CIDM 2007. IEEE Symposium on*, pages 308 –315.

[Collins and Duffy, 2001] Collins, M. and Duffy, N. (2001). Convolution kernels for natural language. In *Proceedings of Neural Information Processing Systems, NIPS 2001*.

[Jungermann, 2011] Jungermann, F. (2011). Tree kernel usage in naive bayes classifiers. In *Proceedings of the LWA 2011*.

[Lafferty et al., 2001] Lafferty, J., McCallum, A., and Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings of the 18th International Conference on Machine Learning*, pages 282–289. Morgan Kaufmann, San Francisco, CA.

[Liu and Nocedal, 1989] Liu, D. C. and Nocedal, J. (1989). On the limited memory method for large scale optimization. In *Mathematical Programming*, volume 45, pages 503–528. Springer Berlin / Heidelberg.

[Mierswa et al., 2006] Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., and Euler, T. (2006). YALE: Rapid Prototyping for Complex Data Mining Tasks. In Eliassi-Rad, T., Ungar, L. H., Craven, M., and Gunopulos, D., editors, *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2006)*, pages 935–940, New York, USA. ACM Press.

[Moschitti, 2006a] Moschitti, A. (2006a). Efficient convolution kernels for dependency and constituent syntactic trees. In Fuernkranz, J., Scheffer, T., and Spiliopoulou, M., editors, *Procs. ECML*, pages 318 – 329. Springer.

[Moschitti, 2006b] Moschitti, A. (2006b). Making tree kernels practical for natural language learning.

[Nocedal, 1980] Nocedal, J. (1980). Updating quasi-newton matrices with limited storage. *Mathematics of Computation*, 35(151):773–782.

[Rueping, 2000] Rueping, S. (2000). *mySVM Manual*. Universitaet Dortmund, Lehrstuhl Informatik VIII. http://www-ai.cs.uni-dortmund.de/SOFTWARE/MYSVM/.

[Zhang et al., 2006] Zhang, M., Zhang, J., Su, J., and Zhou, G. (2006). A composite kernel to extract relations between entities with both flat and structured features. In *Proceedings 44th Annual Meeting of ACL*, pages 825–832.

[Zhou and Su, 2004] Zhou, G. and Su, J. (2004). Exploring deep knowledge resources in biomedical name recognition. In *Proceedings of the Joint Workshop on Natural Language Processing in Biomedicine and its Applications (JNLPBA-2004)*, Geneva, Switzerland.

[Zhou et al., 2005] Zhou, G., Su, J., Zhang, J., and Zhang, M. (2005). Exploring various knowledge in relation extraction. In *Proceedings of the 43rd Annual Meeting of the ACL*, pages 427–434, Ann Arbor. Association for Computational Linguistics.

Appendix A

Operator Reference

A.1 Tokenizers

In this section we will describe the parameters and the input and output ports of the `SentenceTokenizer` in detail. These parameters and ports are extended from the super-class `TokenizerImpl` and they are the same for `WordTokenizer` and `LineTokenizer`. A new tokenizer should be extend the class `TokenizerImpl` and the method `String[] tokenization(String text)` should be implemented.

SentenceTokenizer This tokenizer splits texts into sentences. The parameters for this operator are presented in Table A.1 and the I/O ports are presented in Table A.2.

Parameter	Description
attribute	Select the attribute here which has to be used for extracting the text to be tokenized from each example.
new token-name	Type a new attribute name in here which will be created to write the new tokens to.

Table A.1: Parameters for *SentenceTokenizer*

I/O	port-name	Description
I	**example set input**	The exampleset which will be used for tokenization.
O	**example set output**	The exampleset containing the new tokens.
O	**original example set output**	The original exampleset got at the input port.

Table A.2: I/O-ports for *SentenceTokenizer*

WordTokenizer This tokenizer splits texts into words. The parameters and the I/O ports are the same as for the `SentenceTokenizer`.

LineTokenizer This tokenizer splits texts into lines. It splits texts contained in examples by using the linebreaks in the texts, one could say. The parameters and the I/O ports are the same as for the `SentenceTokenizer`.

A.2 Visualizer

The visualizers are used for the annotation and the visualization of textual data. The parameters and the ports of the `ParseTreeVisualizer` and the `TextVisualizer` are comparable, and therefore we only present the parameters and ports of the `ParseTreeVisualizer`.

ParseTreeVisualizer This operator creates a visualization of parse trees. For each example the parse tree and the corresponding sentence will be displayed. Only one example is displayed at a time, but one can switch from one example to another during the visualizing process. Table A.3 describes the parameters of the operator and Table A.4 describes the ports.

Parameter	Description
sentence-attribute	Select the attribute here which has to be used for extracting the text to be displayed in addition to the parse tree from each example.
parsetree-atrribute	Select the attribute here which has to be used for extracting the parse tree to be displayed from each example.

Table A.3: Parameters for *ParseTreeVisualizer*

I/O	port-name	Description
I	**example set input**	The exampleset which will be used for tokenization.
O	**parsetree visualization output**	The visualization component which becomes visible in the results workspace.
O	**original example set output**	The original exampleset got at the input port.

Table A.4: I/O-ports for *ParseTreeVisualizer*

TextAnnotator The `TextAnnotator` is used for annotating a textual dataset. The textual data of the `text-attribute` are displayed. The user can annotate the tokens by marking them and by selecting formerly created labels. During

the annotation process it is possible to create new label types which can directly be used for annotating. The labels are stored in the `label-attribute` for each token. After finish the annotation process the dataset is stored in a repository which is selected using the parameter `repository-entry`.

Parameter	Description
repository-entry	Select a repository entry here which has to be used for storing the dataset after having finished the annotation process.
text-attribute	Select the attribute here which has to be used for extracting the text to be displayed in addition to labeling from each example.
label-attribute	Select the attribute here which has to be used for extracting the label to be displayed from each example.

Table A.5: Parameters for *TextAnnotator*

I/O	port-name	Description
I	**example set input**	The exampleset which will be used for tokenization.
O	**annotation output**	The annotation component which becomes visible in the results workspace.
O	**example set output**	The original exampleset got at the input port.

Table A.6: I/O-ports for *TextAnnotator*

TextVisualizer This operator creates a visualization of labeled textual data. Like for the `ParseTreeVisualizer` two attributes have to be selected. The nominal values of the first attribute will be displayed as text whereas the nominal values of the second attribute will be used as labels. The parameters and ports are comparable to the ones of the `ParseTreeVisualizer` and thererfore are not specified again.

A.3 Preprocessing

WordVectorPreprocessing This operator creates a BOW representation of the values given by the attributes defined by the parameter `valueAttribute`. The regular expression which can be set using the parameter `splitExpression` is used for splitting the suggested values. Each unique resulting splitted value is used to create a new attribute containing a 1 if the value in the attribute `valueAttribute` contains the splitted value and 0, otherwise. The operator is very similar and we disclaim on presenting the parameters and ports in detail.

A.3.1 Named Entity Recognition

The preprocessing operators for NER are extending two types of super classes. The first one is `PreprocessOperatorImpl` and the second one is `MultiPreprocessOperatorImpl`. `PreprocessOperatorImpl` enriches the dataset by one additional attribute for every value it sees, whereas `MultiPreprocessOperatorImpl` is creating multiple additional attributes for every value.

WordPreprocessing `WordPreprocessing` works like the operator `PrefixPreprocessing`. The difference is that the extracted value from attribute `wordAttributeName` is directly stored (without any manipulation) to the attribute `operatorName`. The parameter `length` is not needed and therefore it is not apparent.

PrefixPreprocessing `PrefixPreprocessing` extends `PreprocessOperatorImpl`. The most important method to implement after extending `PreprocessOperatorImpl` is
`String newValueToInsert(String w, int length,int index)`.
This method is delivering the new attribute value in a `String` representation and is called using the parameters `w` which contains the value of the attribute `wordAttributeName` of the `index`th example shifted by `position` examples. The parameter `length` can be used if the resulting value should contain a specific length. All operators extending `PreprocessOperatorImpl` until some exceptions are defined by the same parameters which exemplarily are listed in Table A.7. The input and output ports are presented in Table A.8. This operator creates a new attribute containing the prefixes of length `length` of the values extracted from attribute `wordAttributeName`. If the value extracted from attribute `wordAttributeName` is 'Felix', for instance, the prefix of length 3 is 'Fel' and will be stored in the attribute `operatorName`.

Parameter	Description
position	The position of the example to be chosen relative to the current example. (-2 will take the example two examples before, -2_2 will take all examples between two before and two after the current example, $-2, 2$ will take the examples two before and two after the current example)
length	The parameter length is determining the length of the extracted prefix.
operatorName	Select the name of the new attribute here which will contain the created values.
wordAttributeName	Select the attribute which will be used to extract the values to be used for preprocessing.

Table A.7: Parameters for *PrefixPreprocessing*

I/O	port-name	Description
I	**example set input**	The exampleset which will be used for preprocessing.
O	**example set output**	The exampleset containing the new attribute(s).
O	**original**	The original exampleset got at the input port.

Table A.8: I/O-ports for *PrefixPreprocessing*

ngram_preprocessing `ngram_preprocessing` extends `MultiPreprocessOperatorImpl`. The most important method to implement after extending `MultiPreprocessOperatorImpl` is `ArrayList<String> newValueToInsert(String w,int length,int index)`. This method is delivering a list of new attribute values and the processing is comparable to the processing of `PreprocessOperatorImpl`. The resulting values will be converted into a binary attribute, each. This operator will split the values extracted from attribute `wordAttributeName` into pieces of length `length`. Each unique piece will result in a new attribute which contains a 1 for each example containing this piece or a 0 otherwise. The parameters and input and output ports are comparable to the already presented operators.

IOBPreprocessing This operator cuts off the prefixes 'B-' and 'I-' of `wordAttributeName`.

IndexPreprocessing This operator just takes the index of the relatively chosen example and creates an additional attribute containing this information.

SuffixPreprocessing Works like `PrefixPreprocessing` but with the difference that the suffixes are extracted.

WordCountPreprocessing This operator counts the words of the current sequence and creates an attribute containing this information.

GeneralizationPreprocessing This operator generalizes the currently chosen value of `wordAttributeName` by replacing capital letters by 'A', small letters by 'a' and digits by 'x'. The resulting value is stored in the newly created attribute `operatorName`.

WikipediaPreprocessing This operator creates a new attribute and puts the wikipedia-category of the `wordAttributeName` of the relatively chosen example as a value into it.

LetterCountPreprocessing This operator counts the number of letters of the value extracted of `wordAttributeName` of the relatively chosen example and stores them as a value into a newly created attribute.

RegExPreprocessing `RegExPreprocessing` also extends `MultiPreprocessOperatorImpl`. The chosen value is checked against a number of regular expression. If an expression is matched, the corresponding attribute of the example is allocated with a 1 and otherwise it is allocated with a 0.

TagListPreprocessing This operator checks the `wordAttributeName` of the relatively chosen example against a given taglist. If the current value is contained in the list, the newly created attribute is allocated with 1 and otherwise it is allocated with 0.

StartOfSequencePreprocessing This operator creates a new attribute that contains a 1 if the current token shifted by `location` is at the beginning of the sequence/sentence and 0 otherwise.

EndOfSequencePreprocessing Works like `StartOfSequencePreprocessing` but for the end of sequences/sentences.

A.3.2 Relation Extraction

In this section we present the preprocessing operators related to relation extraction.

TreeCreatorAndPreprocessor The operator `TreeCreatorAndPreprocessor` reads the values out of a particular attribute and creates a `Tree`-representation out of it. The resulting tree-structured value is stored in a newly created attribute. The creation of the tree structure is done by parsing a machine-readable or natural language sentence. Additionally, a tree structure given in `String`-representation can be read and converted into a tree-structured attribute value. The parameters and the input and output ports of this operator are shown in Tables A.9 and A.10.

Parameter	Description
valueAttribute	The attribute which contains the tree or the sentence to be parsed.
needParsing	Does the attribute value need to be parsed?
modelfile	The file containing a parser model
parseTreeType	The trees have to be pruned for special tasks. Select pruning type here.
FTK	Selecting this will activate the list-creation needed for the FTK.

Table A.9: Parameters for *TreeCreatorAndPreprocessor*

I/O	port-name	Description
I	**example set input**	The exampleset which will be used for preprocessing.
O	**example set output**	The exampleset additionally containing the new attribute.

Table A.10: I/O-ports for *TreeCreatorAndPreprocessor*

HTMLTreePreprocessing The `HTMLTreePreprocessing` operator works like the `TreeCreatorAndPreprocessor` operator as it creates tree-structured attribute values. In contrast to the `TreeCreatorAndPreprocessor` the resulting trees are not pruned.

Zhou Features

In this section we present some of the features published in [Zhou and Su, 2004]. According to the two entities creating the relation candidate, several informational units concerning these entities are extracted by the features. The features are extending the class `RelationPreprocessOperatorImpl`. Two particular methods have to be implemented by each operator:
`String newValueToInsertTree(Tree t)` and
`String newValueToInsert(String w1, String w2, List<String> addAtts, int length, ExampleIteration exIter)`.
Both methods are creating the value for the newly created attribute. The first is based on a tree-representation, and the second one is based on the position of the two entities in the sentence.

WBNULLPreprocessing This operator creates a new binary attribute containing a 'true'-value if no word is located between the two entities creating the relation candidate. The parameters and input and output ports of this operator are presented in Tables A.11 and A.12.

Parameter	Description
operatorName	The name of the new attribute that will be created.
use tree	If this is selected the tree-structured attribute value will be used.
firstAttributeName	The name of the attribute containing the name of the first entity must be inserted here (only if tree is used).
secondAttributeName	The name of the attribute containing the name of the second entity must be inserted here (only if tree is used).
additional Attributes	Additional attributes can be selected using this parameter list. These attributes are only needed by several operators.

Table A.11: Parameters for *WBNULLPreprocessing*

I/O	port-name	Description
I	**example set input**	The exampleset which will be used for preprocessing.
O	**example set output**	The exampleset additionally containing the new attribute.
O	**original**	The original exampleset got from the input port.

Table A.12: I/O-ports for *WBNULLPreprocessing*

WBFLPreprocessing This operator creates a new attribute which will contain the word between the two entities creating the relation candidate if there is only one word between. The parameters and input and output ports are the same like for `WBNULLPreprocessing`.

WBFPreprocessing This operator creates a new attribute which will contain the first word between the two entities creating the relation candidate if there is more than one word between. The parameters and input and output ports are the same like for `WBNULLPreprocessing`.

WBLPreprocessing This operator creates a new attribute which will contain the last word between the two entities creating the relation candidate if there is more than one word between. The parameters and input and output ports are the same like for `WBNULLPreprocessing`.

WBOPreprocessing This operator creates a new attribute which will contain the words except the first and the last one between the two entities creating the relation candidate if there is more than one word between. The parameters and input and output ports are the same like for `WBNULLPreprocessing`.

BM1FPreprocessing This operator creates a new attribute which will contain the word before the first entity of the two entities creating the relation candidate. The parameters and input and output ports are the same like for `WBNULLPreprocessing`.

BM1LPreprocessing This operator creates a new attribute which will contain the word two words before the first entity of the two entities creating the relation candidate. The parameters and input and output ports are the same like for `WBNULLPreprocessing`.

AM2FPreprocessing This operator creates a new attribute which will contain the word after the last entity of the two entities creating the relation candidate. The parameters and input and output ports are the same like for `WBNULLPreprocessing`.

AM2LPreprocessing This operator creates a new attribute which will contain the word two words after the last entity of the two entities creating the relation candidate. The parameters and input and output ports are the same like for `WBNULLPreprocessing`.

M1greaterM2Preprocessing This operator creates a new binary attribute which will contain 'true' if the second entity is encapsulated in the first entity. The parameters and input and output ports are the same like for `WBNULLPreprocessing`.

NumberOfMBPreprocessing This operator creates a new attribute which will contain the number of entity mentions between the two entities creating the relation candidate. The parameters and input and output ports are the same like for `WBNULLPreprocessing`.

NumberOfWBPreprocessing This operator creates a new attribute which will contain the number of words between the two entities creating the relation candidate. The parameters and input and output ports are the same like for `WBNULLPreprocessing`.

A.4 Meta

Binary2MultiClassRelationLearner This operator works like the `Polynomial by Binomial Classification` operator already available in *RapidMiner*. If a dataset containing more than two classes should be processed by a binary learner a strategy to use the binary learner will have to be chosen. The main reason for developing a new operator for this purpose is the fact that some candidates for relation extraction are defined only for some special relation types. It follows that the amount of relation candidates varies for each type of

relation class currently focussing. For each class different parameters have to be adjusted for the internal learning mechanism. The parameters and input and output ports are presented in Tables A.13 and A.14

Parameter	Description
classification strategies	The strategy to be chosen to partition the dataset for applying the internal learner.
use local random seed	An own random seed is used to achieve repeatable results.
event1	The name of the attribute containing the type of the first entity must be inserted here.
event2	The name of the attribute containing the type of the second entity must be inserted here. Using the two types of entities allows to check whether the combination is suitable for particular relation classes.
null-Label	The default-class has to be selected here.
confidence	The confidence-level which has to be achieved to predict a certain class. If the confidence-level is not being achieved by any class the default-class is predicted.
epsilon-list	A particular epsilon-value can be adjusted for every learner/class here.

Table A.13: Parameters for *Binary2MultiClassRelationLearner*

I/O	port-name	Description
I	**training set**	The exampleset which will be used for learning.
O	**model**	The model created by the internal learning mechanism.
O	**example set**	The original exampleset got from the input port.

Table A.14: I/O-ports for *Binary2MultiClassRelationLearner*

A.5 Data

CSVbatchedDataReader The `CSVbatchedDataReader` operator extends the `Read CSV` operator which is already available in *RapidMiner*. The `CSVbatchedDataReader` operator allows to read in datasets which are prepared like described in Section 1.5.1.

A.6 Learner

TreeKernel Naive Bayes The `TreeKernel Naive Bayes` operator is the implementation of the approach presented in Section [Jungermann, 2011]. We

only present the parameters differing from the `Naive Bayes (Kernel)` operator in Table A.15. Table A.16 contains the input and output ports of the operator.

Parameter	Description
lambda	The first kernel to be used for the *Composite Kernel* (just *Entity* is possible)
sigma	The list of attributes to be used by the *Entity Kernel*
gaussian	The λ-value to be used for QTK or FTK by the *Composite Kernel*
treestructure selection	The λ-value to be used for QTK or FTK by the *Composite Kernel*
distribution	The λ-value to be used for QTK or FTK by the *Composite Kernel*

Table A.15: Parameters for *Treekernel Naive Bayes*

I/O	port-name	Description
I	**training set**	The exampleset which will be used for learning.
O	**model**	The model created by the internal learning mechanism.
O	**exampleSet**	The original exampleset got from the input port.

Table A.16: I/O-ports for *Treekernel Naive Bayes*

TreeSVM We enhanced the already available *JMySVM* [Rueping, 2000] implementation in *RapidMiner* by abilities to process tree structures. We implemented the Kernel presented by [Collins and Duffy, 2001], the Fast Tree Kernel by [Moschitti, 2006b, Moschitti, 2006a] and the Composite Kernel by [Zhang et al., 2006]. The operator `TreeSVM` has some additional parameters in contrast to the `Support Vector Machine` operator. These parameters are shown in Table A.17. The input and output ports are presented in Table A.18.

Parameter	Description
kernel type	The kernel type to be used (*Collins and Duffy (trivial)*, *Moschitti (FTK)*, *Composite Kernel*)
CollinsDuffy Kernel Lambda	The λ-value to be used (see [Collins and Duffy, 2001]) for QTK or FTK
Composite Kernel Alpha	If the *Composite Kernel* is used the α value (see [Zhang et al., 2006]) can be adjusted here.
kernel type 1	The first kernel to be used for the *Composite Kernel* (just *Entity* is possible)
attribute list	The list of attributes to be used by the *Entity Kernel*
kernel type 2	The second kernel to be used for the *Composite Kernel* (QTK and FTK possible)
Collins Duffy Kernel Lambda (composite)	The λ-value to be used for QTK or FTK by the *Composite Kernel*

Table A.17: Additional parameters for *TreeSVM*

I/O	port-name	Description
I	**training set**	The exampleset which will be used for learning.
O	**model**	The model created by the internal learning mechanism.
O	**estimated performance**	The original exampleset got from the input port.
O	**exampleSet**	The original exampleset got from the input port.

Table A.18: I/O-ports for *TreeSVM*

Kernel Perceptron The `Kernel Perceptron` operator is the implementation of the approach presented by [Aiolli et al., 2007]. The parameters of the operator are presented in Table A.19. The input and output ports are presented in Table A.20.

Parameter	Description
kernel type	The kernel type to be used (*CollinsDuffy*, *FastTree*, *Treeceptron*, *DAGperceptron*, *OneDAGperceptron*)
attribute	The attribute containing the tree-structures.
lambda	The λ-value to be used (see [Collins and Duffy, 2001]) for the tree kernel.
sigma	The σ-value to be used (see [Moschitti, 2006b]) for the tree kernel.
bootstrap	If this is selected the at each step a randomly chosen example will be selected.
stopping	After this number of iteration training will stop. Selecting -1 will make the perceptron do one run on the complete exampleset.

Table A.19: Parameters for *Kernel Perceptron*

I/O	port-name	Description
I	**training set**	The exampleset which will be used for learning.
O	**model**	The model created by the internal learning mechanism.
O	**example set**	The original exampleset got from the input port.

Table A.20: I/O-ports for *Kernel Perceptron*

ConditionalRandomField This operator implements a CRF which is presented by [Lafferty et al., 2001]. The parameters and the input and output ports of this operator are presented in Tables A.21 and A.22. This operator needs an embedded particular optimization operator. The optimizer is presented in the next section (Section A.6.1).

Parameter	Description
text-Attribute name	Select the attribute containing the words of the sentence to be processed

Table A.21: Parameters for *ConditionalRandomField*

I/O	port-name	Description
I	**example set input**	The exampleset which will be used for learning.
O	**example set output**	The original exampleset got from the input port.
O	**model output**	The model created by the CRF.

Table A.22: I/O-ports for *ConditionalRandomField*

A.6.1 Optimizer

LBFGS_optimizer The `LBFGS_optimizer` can be used as an optimization method for the `ConditionalRandomField`-operator. The optimization technique is presented by [Nocedal, 1980, Liu and Nocedal, 1989].

Parameter	Description
maximum number of iterations	After this number of iterations the optimization is stopped.
eps for convergence	The solution is assumed to be optimal if a smaller ϵ value than this one is achieved during optimization.
features sorted in array	Select this if the features are sorted in an array.

Table A.23: Parameters for *LBFGS_optimizer*

I/O	port-name	Description
I	**feature set input**	The feature set which will be used for optimization.
O	**feature set output**	The feature set containing the optimized weights.

Table A.24: I/O-ports for *LBFGS_optimizer*

A.7 Validation

If particular examples or tokens are grouped into sequences or sentences these sequences should be respected by operators which split the example set like `SplittedXBatchValidation` and `BatchSplitValidation` or `SequenceSampling`.

PerformanceEvaluator This operator is a special performance evaluator for NER. It calculates precision, recall and f-measure for each class except the default-class. Additionally, the accumulated values are calculated because those values are used very often in the information extraction community.

Parameter	Description
precision	The precision will be calculated for each class except the default class (`not-NER-tag`) if this box is selected.
recall	The recall will be calculated for each class except the default class (`not-NER-tag`) if this box is selected.
f-measure	The f-measure will be calculated for each class except the default class (`not-NER-tag`) if this box is selected.
overall-precision	The precision will be calculated for all classes together except the default class (`not-NER-tag`) if this box is selected.
overall-recall	The recall will be calculated for each class except the default class (`not-NER-tag`) if this box is selected.
overall-f-measure	The f-measure will be calculated for each class except the default class (`not-NER-tag`) if this box is selected.
stopword	A special token represents the end of a sentence in some datasets. This token can be selected here so that it is not used for the calculation of the performance.
not-NER-tag	Type in the default-class here.
iob	Select this box if the dataset is in IOB-format.

Table A.25: Parameters for *PerformanceEvaluator*

I/O	port-name	Description
I	**example set input**	The example set which will be used for evaluating the performance.
O	**example set output**	The original example set from the input port.
O	**simple-Performance**	Delivers the `SimpleResultObject` from *Rapid-Miner*.
O	**performance**	Delivers the performance calculated by this operator.

Table A.26: I/O-ports for *PerformanceEvaluator*