JRC Scientific and Technical Reports

CORLEONE
Core Linguistic Entity Online Extraction

Jakub Piskorski

EUR 23393 EN - 2008

JRC
EUROPEAN COMMISSION

ipsc
Institute for the Protection and Security of the Citizen

The mission of the IPSC is to provide research results and to support EU policy-makers in their effort towards global security and towards protection of European citizens from accidents, deliberate attacks, fraud and illegal actions against EU policies.

European Commission
Joint Research Centre
Institute for the Protection and Security of the Citizen

Contact information
Address: T.P. 267, Via Fermi 1, 21020 Ispra (VA), Italy
E-mail: Jakub.Piskorski@jrc.it
Tel.: +39 0332 78-6389
Fax: +39 0332 78-5154

http://ipsc.jrc.ec.europa.eu/
http://www.jrc.ec.europa.eu/

JRC45952

EUR 23393 EN
ISSN 1018-5593

Luxembourg: Office for Official Publications of the European Communities

Printed in Italy

CORLEONE
Core Linguistic Entity Online Extraction

Jakub Piskorski
Joint Research Centre of the European Commission
Web Mining and Intelligence Action
Support to External Security Unit
Institute for the Protection and Security of the Citizen
T.P. 267, 21027 Ispra (VA), Italy
Email: Jakub.Piskorski@jrc.it*

Abstract

This report presents CORLEONE (**Cor**e **L**inguistic **E**ntity **On**line **E**xtraction) – a pool of loosely coupled general-purpose basic lightweight linguistic processing resources, which can be independently used to identify core linguistic entities and their features in free texts. Currently, CORLEONE consists of five processing resources: (a) a basic tokenizer, (b) a tokenizer which performs fine-grained token classification, (c) a component for performing morphological analysis, and (d) a memory-efficient database-like dictionary look-up component, and (e) sentence splitter. Linguistic resources for several languages are provided. Additionally, CORLEONE includes a comprehensive library of string distance metrics relevant for the task of name variant matching. CORLEONE has been developed in the Java programming language and heavily deploys state-of-the-art finite-state techniques.

Noteworthy, CORLEONE components are used as basic linguistic processing resources in ExPRESS, a pattern matching engine based on regular expressions over feature structures [1] and in the real-time news event extraction system [2], which were developed by the Web Mining and Intelligence Group of the Support to External Security Unit of IPSC.

This report constitutes an end-user guide for COLREONE and provides scientifically interesting details of how it was implemented.

1 Overview

The European Commission monitors media reports concerning EU policies and potential threats on a daily basis. In order to facilitate and automate this endeavour the Europe Media Monitor (EMM) [3] was developed at the Joint Research Center of the European Commission in Ispra. Its major task is to scan hundreds of web sites in different languages (focusing mainly on news agencies) and to detect breaking news stories which constitute essential information for European Policy Makers. Some of the current activities on EMM enhancements focus on the development of tools and techniques for automatic extraction of structured information on events mentioned in the news stories, i.e., deriving information on events in the form of templates which record *who did what to whom where when and with what consequences* [4, 2]. Such structured information can be utilized for country conflict assessment studies, terrorism knowledge bases and general situation monitoring. A crucial prerequisite to tackle the problem of information extraction from large document collections is to have some natural language processing (NLP) components which robustly and efficiently perform basic linguistic operations. These include among others splitting a text into word-like units and classifying them in a fine-grained manner, recognition of domain-relevant named-entities, keywords and phrases based on large and exhaustive lists thereof (also associating the

*Alternatively, the author can be contacted via the following email: jpiskorski@gmail.com

recognized entities with some additional information present in the background knowledge base), performing morphological analysis, i.e., associating words with the corresponding part-of-speech, base form, and other morphological information, identifying sentence boundaries, and name variant matching.

In this report, we present CORLEONE (**Cor**e **L**inguistic **E**ntity **On**line **E**xtraction) - a pool of loosely coupled general-purpose basic lightweight linguistic processing resources, which can be independently used to identify core linguistic entities and their features (e.g., tokens, morphologically analyzed tokens, named-entities, etc.) in free texts. They provide essential information indispensable for higher-level processing, e.g. relation extraction, event extraction, etc. The main motivation behind developing CORLEONE was to have core NLP tools that: (a) efficiently and robustly process vast amount of textual data, (b) are easily adaptable to new domains and support multilinguality, (c) allow full-control over their resources, i.e., do not exhibit black-box characteristics, (d) do not rely on any third-party software, and (e) are easily portable among different operating platforms. In order to meet the aforesaid requirements CORLEONE was developed in the Java programming language and state-of-the art finite-state technology has been heavily exploited [5] since finite-state machines guarantee efficient processing and have an expressive power to model many interesting language phenomena in the context of processing online news texts. The resources used by all components may be easily manipulated and parameterized by the users in many ways. In particular the latter feature is of major importance in the context of the enhancements of the EMM system. Currently, CORLEONE consists of five processing resources: (a) a basic tokenizer, (b) a tokenizer which performs fine-grained token classification, (c) a component for performing morphological analysis, and (d) a memory-efficient database-like dictionary look-up component, and (e) sentence splitter. Further, CORLEONE includes also a comprehensive library of string distance metrics relevant for the task of name variant matching. The ideas and techniques utilized to implement CORLEONE are based on author's prior work in the years 1998-2005 on the development of the shallow processing engines SPPC [6, 7] and SProUT [8, 9, 10] and work on core finite-state technology [11, 12, 13, 14].

The rest of this report is organized as follows. First, in section 2 some instructions on how to install the tools is given. Next, in section 3, compilation and deployment of CORLEONE components is described. Subsequently, sections 4, 5, 6, 7 and 8 describe in detail the five core modules, namely the basic tokenizer, the classifying tokenizer, the morphological analyzer, the dictionary look-up component and the sentence splitter. The string distance metrics library is briefly presented in section 9. Finally, a summary and an outlook on future work is given in section 10. A comprehensive reference list is provided at the end of this report.

This report is intended for both end-users of CORLEONE and those, who are interested in details about the implementation of the tool. In particular, end-users may wish to skip the sections on implementation.

2 Installation and System Requirements

In order to be able to use any of CORLEONE components extract the archive `CORLEONE_DUE.zip` into a directory of your choice. Please note that the directory structure of the archive must be preserved. The top directory includes some scripts for running and compiling CORLEONE components (resources), which are described in detail in section 3. Further, there are five subdirectories:

- `lib` contains Java libraries required for running the programs
- `java_doc` contains Java API documentation for all higher-level classes
- `data` contains raw and compiled linguistic resources accompanied with compilation and application configuration scripts (more details in subsequent sections). For each component and language there is a specific subdirectory with all resources
- `test` contains some sample English texts to play with
- `documentation` contains CORLEONE documentation

Finally, make sure that a Java Virtual Machine (JVM) version 1.5.0. or higher is installed on your machine. This is a prerequisite for running the software. It can be downloaded from the `http://java.sun.com` web page. The CORLEONE itself is just a single jar file. There are two versions provided, one compiled with Java 1.5 and one compiled with Java 1.6.

component	alias
basic tokenizer	'basicTokenizer'
classifying tokenizer	'classifyingTokenizer'
morphological analyser	'multextMorphology'
domain-specific lexicon look-up tool	'basicGazetteer'
sentence splitter	'sentenceSplitter'

Figure 1: Available processing components and their aliases

The amount of memory required while running any of the CORLEONE components strongly depends on the amount of data involved. Nevertheless, processing a single MB-sized documents is done within seconds.

CORLEONE uses *log4j* - a logging service for printing log output to different local and remote destinations. Logging behavior can be controlled by editing a configuration file (`log4j.properties`)[1], without touching the application binary. By default all messages are sent to standard output. Please refer to the *log4j* manual available at `http://logging.apache.org/log4j/docs/` for more details on the log4j properties (configuration).

Finally, it is important to note that CORLEONE 2.0 API is not compatible with version 1.0. In particular, some of the features of CORLEONE 1.0 are no longer provided. Additionally, version 2.0 is now thread safe, except the compilation of linguistic resources, which is usually done off line.

3 Compilation and Deployment of Corleone Components

The table in Figure 1 gives a list of available processing components and their aliases. Please use these aliases when compiling/deploying CORLEONE components using the scripts described in this section. Raw and compiled linguistic resources for English, for each of the processing components listed in the table can be found in various subdirectories under `\data`. Additionally, tokenizer and morphological resources for German and Italian are provided too.[2] For instructions on using string distance metrics library please see section 9 directly.

All CORLEONE components can either be used as stand-alone modules or easily integrated within other frameworks via utilization of Java API. Please refer to Java API documentation for further details.

3.1 Compilation

For the compilation of the resources for any of the CORLEONE components please use the following script, where `<compilation configuration file>` refers to the component's specific compilation configuration file. The syntax of the latter ones is described in detail in the subsequent sections.

```
compileComp   <component alias>  <compilation configuration file>
```

Compilation configuration files can be found in appropriate subdirectories under `\data` and they have an extension `cfg`. For instance, in order to compile domain-specific lexicon with the compilation-configuration file `comp.cfg`, run:

```
compileComp  basicGazetteer  comp.cfg
```

In case of successful compilation, a message 'Compilation Successful' should be displayed, otherwise error messages will be sent to the appenders specified in the logger properties file (by default: standard output).

3.2 Deployment

If one is not a programmer or does not need or wish to use CORLEONE Java API, he/she can use the following script for toying with the component, where `<application configuration file>` refers to the component's specific application configuration file. The syntax of the application configuration file for each single component is presented in detail in the subsequent sections resp.

[1] Note that this file has to be located in the root directory for the binaries

[2] The resources for various new languages are being added continuously

```
applyComp   <component alias>  <application configuration file>
```

Sample application configuration files can be found in appropriate subdirectories of `\data` and they have an extension `cfg` and include the string `application` in their name. For instance, in order to apply the classifying tokenizer with the application-configuration file `appl.cfg`, run:

```
applyComp  classifyingTokenizer  appl.cfg
```

In case of encountered problems or errors, messages will be sent to the appenders specified in the logger properties file (by default: standard output).

4 Basic Tokenizer

4.1 Introduction to Tokenization

Tokenization is commonly seen as an independent process of linguistic analysis, in which the input stream of characters is segmented into an ordered sequence of word-like units, usually called tokens, which function as input items for subsequent steps of linguistic processing. Tokens may correspond to words, numbers, punctuation marks, or even proper names. The recognized tokens are usually classified according to their syntax, but the way in which such classification is done may vary. Since the notion of tokenization seems to have different meanings to different people, some tokenization tools fulfill additional tasks, like for instance isolation of sentences, handling of end-line hyphenations or conjoined clitics and contractions [15]. Some software systems for performing tokenization are presented in [15, 16], and [17]. The famous general-purpose character stream scanners *lex* and *flex* are described in [18].

Performing tokenization is a prerequisite in order to perform any IE task. In particular, fine-grained token classification (e.g., differentiating between numbers, lowercase words, words containing apostrophe or hyphens, all-capital words etc.) might come in handy on later processing stages [6]. Splitting a text into tokens and performing their classification can be done very efficiently via application of finite-state devices. CORLEONE provides two tokenizers, which extensively utilizes this formalism. A simple one just segments the text into word-like units based on a list of whitespaces and token separators. The more sophisticated one performs additionally fine-grained classification of tokens. In the remaining part of this section we describe the basic tokenizer, whereas the classifying tokenizer is described in detail in section 5.

4.2 Tokenizer

The CORLEONE basic tokenizer splits a stream of input characters into word-like units based on a list of whitespaces and token separators. An output of the basic tokenizer is a sequence of feature structures representing the tokens. These structures include three attributes, i.e., start/end attributes encode the positional information, whereas the type attribute can be either set to `whitespace` or `non-whitespace`. Further, the tokenizer is Unicode-aware, i.e., any Unicode input stream can be used as an input to the tokenizer. Further, there is an option to return or not to return tokens representing whitespaces or sequences thereof.

4.3 Preparation of Resources for the Basic Tokenizer

The resources needed for compiling a basic tokenizer consist of solely a compilation configuration file, which defines the following properties:

- `Name`: specifies the name of the tokenizer (arbitrary string)
- `TokenSeparators`: a list of token separators, where each entry in the list is a Unicode character in the format `\uxxxx` (`xxxx` are four hexadecimal digits)
- `WhiteSpaces`: a list of whitespaces, where each entry in the list is a Unicode character in the format `\uxxxx` (`xxxx` are four hexadecimal digits)
- `OutputFile`: a path to the output file, i.e., compiled version

The properties specifying token separators and whitespaces are not obligatory. If they are missing, the default whitespace list and token separator list is used. The default whitespaces are: `\t\n\r\f` and the standard space symbol. The default token separators include among other the following symbols (the complete list can be found in Java API):

```
. ,/ \ -: ; % ! ? ~ # ^ & * + = | @ ) ] } > ( [ { < $ " ’ ~
```

The following example gives an idea of how a compilation configuration file looks like.

```
Name=myTokenizer
WhiteSpaces=\u0009,\u0020,\u000A,\u000D
TokenSeparators=\u02bd,\u02ca,\u02cb,\u02ba,\u02dc,\u02dd
OutputFile=compiled_tokenizer.tok
```

4.4 Tokenizer Deployment

The application configuration file for the basic tokenizer specifies the following properties:

- `ResourceFile`: a path to the compiled basic tokenizer resources
- `Input`: a path to the input file or input directory
- `Output`: a path to the output directory
- `Whitespaces`: is set to ‘true’ if whitespace tokens should be returned, or to ‘false’ otherwise (default setting)
- `CharacterSet`: the name of the character set used for decoding input file(s)

A sample application configuration file looks as follows.

```
ResourceFile=compiled_tokenizer.tok
CharacterSet=ISO-8859-1
Input=./test/input/sample_file.txt
Output=./test/output
Whitespaces=false
```

`CharacterSet` attribute can be set for instance to `US-ASCII`, `ISO-8859-1`, `UTF-8`, or `UTF-16BE`. For the details please read the documentation of the class `java.nio.charset.Charset` in the Java API documentation available at `http://www.java.sun.com`

In order to directly integrate the tokenizer in other application the following piece of code could be used. It corresponds more or less to the above application configuration file (in a simplified form). Please see the Java API documentation for further details.

```
import java.util.ArrayList;
import it.jrc.lt.core.component.tokenizer.*;
import it.jrc.lt.core.component.*;
import piskorski.util.functions.*; // needed for the Files class
...

// Create an instance of a basic tokenizer
AbstractTokenizer tokenizer = AbstractTokenizer.createInstance("basicTokenizer");

// Read tokenizer resource from file
tokenizer.readFromFile("compiled_tokenizer.tok");

// Set options (no whitespace tokens)
tokenizer.ignoreWhitespaceTokens()

// Read text from file
String input = Files.FileToString("sample_file.txt","ISO-8859-1");
```

```
// tokenizer
ArrayList<AbstractTokenItem> tokens = tokenizer.tokenize(input);

// iterate over the tokens
for(AbstractTokenItem t : tokens)
 System.out.println(t.getStart() + " " + t.getEnd() + " "
                                + t.getText(input) + ":"
                                + t.getTypeAsString());
```

5 Classifying Tokenizer

5.1 Introduction

The CORLEONE basic tokenizer splits a stream of input characters into word-like units. The classifying tokenizer does the same job using somewhat more sophisticated algorithm and performs additionally fine-grained token classification. In contrast to other approaches, the context information is disregarded during token classification, since our goal is to define tokenization as a clear-cut step of linguistic analysis. We believe that strong decomposition of linguistic processing into independent components allows for preserving a high degree of flexibility. As a consequence, and unlike common tokenizers, neither multi word tokens nor simple named entities (such as date and time expressions) are recognized or classified at this stage of processing. Sentences are not isolated, since lexical knowledge might be required to do this effectively. Analogously, hyphenated words are not rejoined since the decision for such additional work is not always straightforward and might require more information than just simple lexicon lookup. Consider as an example German compound coordination, where the common part of the compound may be missing, e.g., *Audio- und Videoprodukten* (audio and video products).

An output of the classifying tokenizer is a sequence of feature structures representing the tokens. For instance, for the text fragment *well-oiled business* the tokenizer could potentially return the following structures.

```
| start:0                            |   | start:11            |
| end:9                              |   | end:18              |
| type:word_with_hyphen_first_lower  |   | type:lowercase_word |
```

The `start` and `end` attributes encode the positional information. The definition of token classes can be specified by the user in order to meet their particular needs. The tokenizer comes with a predefined set of default token classes which are listed with corresponding examples in the table in Figure 2.

Theses token classes are in general language-independent and cover well all languages spoken within the European zone. However, one would define some of the classes in a bit different way for each particular language. For instance, in English, the classes `word_with_apostrophe_first_capital` and `word_with_apostrophe_first_lower` should exclude recognition of clitics (constructions like *it's*) which are more likely to be recognized as two tokens (*it* and *'s*) or eventually three tokens (*it*, *'* and *s*).

The splitting of the text into tokens and their classification is done simultaneously by iterating over the characters from left to right according to the following algorithm.

1. **Segmentation:** Identify the next token candidate via scanning characters from the current position and by using a list of predefined whitespaces (e.g., blanks, tabs, etc.), where the latter can be parameterized by the user

2. **Classification:** classify the token candidate based on the user-specified token class definitions. If a class can be assigned (a maximum one token class may be assigned to a given token) we are done - go to 1, otherwise go to 3.

3. **Word-boundary Trimming:** Check if the current token candidate has an initial or trailing separator (e.g., a comma or period; a user-defined list of such token separators can be used for this purpose) and detach them (isolate) temporarily from the token. Try to classify the remaining part. Now, if a token class can be assigned to a trimmed token we are done (i.e., two tokens are generated out of the candidate token) - go to 1, otherwise go to 4.

class	example
any_natural_number	*123*
dot	.
comma	,
slash	/
back_slash	\
hyphen	-
colon	:
semicolon	;
apostrophe	'
quotation	"
exclamation_sign	!
percentage	%
question_mark	?
currency_sign	£
opening_bracket	([{
closing_bracket	)] }
other_symbol	# + @
all_capital_word	*EU*
lower_case_word	*commission*
first_capital_word	*Ispra*
mixed_word_first_lower	*aBB*
mixed_word_first_capital	*GmbH*
word_with_hyphen_first_capital	*Siemens-Sun-Microsoft*
word_with_hyphen_first_lower	*well-oiled*
word_with_apostrophe_first_capital	*O'Neil*
word_with_apostrophe_first_lower	*don't*
email_address	*jakub.piskorsk@jrc.it*
url	*http://www.onet.pl*
number_word_first_capital	*2MB*
number_word_first_lower	*200kg*
word_number_first_capital	*Windows2000*
word_number_first_lower	*mk2*

Figure 2: Default token classes

4. **Postsegmentation:** try to split the token candidate T into a concatenation of token separators and character sequences which do not include token separators, i.e., $T = S_0T_1S_1 \ldots T_nS_n$, where S_i is a token separator and T_i is a sequence of non token-separator symbols (for $i \in \{1, \ldots, n\}$). If all T_i can be assigned some token class, we simply post segment token T adequately. Otherwise we assign token T an *undefined* label and do not post segment it. Note that S_0 and S_n in T can be empty. Go to step 1.

For the sake of clarity, we explain the word-boundary trimming step (3) and the postsegmentation (4) in more detail. First of all, the word-boundary trimming is optional (by default word-boundary trimming is on) and can be deactivated only on demand. Nevertheless, this step seems to come in handy when handling tokens near sentence boundaries. Consider for instance, an URL address being the last part of the sentence, e.g., *http://www.jrc.it* which is directly followed by a dot. Obviously, the text fragment *http://www.jrc.it.* (with the final dot) is not a valid URL, but via switching the word-boundary trimming on, the trailing dot would be isolated, so that an URL and the following dot are recognized as two separate tokens, which is intuitively exactly what we want to achieve.

To better understand postsegmentation, consider the character sequence (token candidate) *(1,2,0)*. Since there is no default token class covering such tokens (see 2) we eventually reach step three and trim the opening and closing bracket (if they are defined as token separators). However, the remaining part *1,2,0* does not match any token class neither. As a consequence step 4, i.e., postsegmentation is triggered. Assuming that brackets and commas are defined as token separators, the postsegmentation of the sequence *(1,2,0)* yields seven tokens: *(1 , 2 , 0)* since *1*, *2*, and *0* are of type `any_natural_number`.

Although, experiments proved that the set of default whitespaces, token separators and token classes might handle most English news texts correctly with respect to splitting them into word-like units, some slight changes might be necessary to process texts of a specific type and in a different language. However, the default definitions provide a good starting point for domain-specific fine-tuning of the tokenizer.

Please note also, that the classifying tokenizer is Unicode-aware, i.e., any character set can be used for encoding the raw resources (token classes and separators) and any Unicode input stream can be used as an input to the tokenizer. Further, whitespaces can be returned as tokens as well.

5.2 Implementation

At first glance an implementation of a tokenizer might seem a straightforward task. However, the token classification should be designed with care in order to allow for having a large number of token classes and remain efficient. The token classification, in the presented tokenize, is implemented as an application of a single finite-state automaton, which eventually returns the 'token class'. This automaton is created as follows. Firstly, for each token class i (for $i \in \{1, \ldots, n\}$) we define a regular expression R_i. Subsequently, for each R_i we construct an equivalent automaton which accepts the language $R_i \circ \#i$, where the symbol $\#i$ represents the ID of the i-th token class and does not appear in $R_1 \cup \ldots \cup R_n$. In other words, the transitions labeled with token class IDs can be seen as a kind of final emissions, i.e., they produce an output (token class ID) without reading anything. Next, all the elementary automata are merged into a single automaton via a subsequent application of standard finite-state operations of union, ϵ-removal, determinization and minimization [19]. Please note at this stage that if more than one token class match a given token, the one with the lowest ID will be assigned (has the highest priority). Therefore, if for a given state in the final automaton there is more than one transition labeled with a token class ID, we trim all such transition except the one with the lowest ID. In this way, the automaton will return at most one token class for a given token. The same effect could also be obtained by using weighted finite-state devices, so that the ambiguity problem is solved during determinization [7] and no additional trimming is necessary. The compressed transition matrix memory model [20, 21, 19] was used for implementing the the tokenizer. It seemed to outperform others in terms of time efficiency.

In order to allow for using any general-purpose finite-state tools, e.g., AT&T FSM Toolkit [22], for creating the automaton which performs the classification, the definition of this automaton has to be provided in a textual format. The current version of the default token classes were created and converted into a single finite-state automaton via using the general-purpose regular expression compiler provided with SProUT [12] which comes with a user-friendly graphical interface, supports Unicode and provides a wide range of operations for manipulating and optimizing finite-state devices [11].

5.3 Preparation of the Resources for the Classifying Tokenizer

The resources needed for compiling a tokenizer consist of three files, a token classifier file (classifying automaton definition), token names file and a tokenizer compilation configuration file. They will be described one by one in the next subsections.

5.3.1 Classifier File (Automaton File)

This file contains a definition in a textual format of a deterministic automaton which will be used for classifying tokens. The details on what language it accepts are explained in section 5.2. The format of this file should be as follows.

- The first line contains the number of states in the automaton (states must be numbered from 0 to $n-1$, where n is the number of states and 0 always denotes the initial state
- Subsequently, for each state in the automaton there is a line containing a state number and a 0 or 1 (space separated) depending on whether the state is non-final (0) or final (1).
- Next, for each transition there is a line containing a triple consisting of source state, target state and a label. The labels are unicode characters in the format `\uxxxx`, where `xxxx` are four hexadecimal digits. If the label encodes a token-class ID, use the following syntax: `#ID`.

A very simple example of an automaton encoded in such a format which accepts a single character A (a single token class) is given below

```
3
0 0
1 0
2 1
0 1 \u0041
1 2 #1
```

The automaton has 3 states; state 2 is the final state and there are only two transitions: one from state 0 to state 1 labelled with `\u0041` (A) and second one from state 1 to state 2 labelled with `#1`.

Defining such automata manually might be an error-prone task. Therefore it is recommended to use any general-purpose software (currently the regular compiler described in [12]) which allows for converting regular expressions into their corresponding finite-state representation. Most likely, there will not be any need of changing the current version of the resources for classification. In such case, please use the already compiled resources for the classifying tokenizer.

5.3.2 Token-class Name File

This file is a simple list of token-class IDs (in the range of 1-255) associated with their corresponding full names. Each line represents a single token-class ID and its full name. The default token-class name file for English delivered with CORLEONE is given below.

```
1 any_natural_number
2 dot
3 comma
4 slash
5 back_slash
6 hyphen
7 colon
8 semicolon
9 apostrophe
10 quotation
11 exclamation
12 percentage
13 question_mark
14 currency_sign
15 opening_bracket
16 closing_bracket
17 other_symbol
18 all_capital_word
19 lower_case_word
20 first_capital_word
21 mixed_word_first_lower
22 mixed_word_first_capital
24 word_with_hyphen_first_capital
25 word_with_hyphen_first_lower
26 word_with_apostrophe_first_capital
27 word_with_apostrophe_first_lower
28 email_address
29 url_address
30 number_word_first_capital
31 number_word_first_lower
32 word_number_first_capital
33 word_number_first_lower
```

The user can change the IDs arbitrarily, but they have to correspond to some final emissions in the classifying automaton described in 5.3.1. The ID `0` is reserved and is used to denote unknown tokens, i.e., tokens that do not match any of the predefined token classes.

5.3.3 Compilation Configuration File

Once a classifying automaton and token-name file are prepared, we can create a tokenizer compilation configuration file, which defines the following properties:

- `Name`: specifies the name of the tokenizer (arbitrary string)

- `TokenSeparators`: a list of token separators, where each entry in the list is a Unicode character in the format `\uxxxx` (`xxxx` are four hexadecimal digits)
- `WhiteSpaces`: a list of whitespaces, where each entry in the list is a Unicode character in the format `\uxxxx` (`xxxx` are four hexadecimal digits)
- `AutomatonFile`: a path to the file with token classifier (automaton definition in textual format)
- `TokenNames`: a path to the file with token-class names
- `CharacterSet`: the name of the character set used for decoding automaton file and token names file
- `OutputFile`: a path to the output file, i.e., compiled version of the tokenizer resources

The properties specifying token separators and whitespaces are not obligatory. If they are missing, the default whitespace list and token separator list is used. The default whitespaces are the same as for basic tokenizer (see 4.3). The following example gives an idea of how a compilation configuration file might look like.

```
Name=myTokenizer
WhiteSpaces=\u0009,\u0020,\u000A,\u000D
TokenSeparators=\u02bd,\u02ca,\u02cb,\u02ba,\u02dc,\u02dd
TokenNames=sample_token_names.txt
AutomatonFile=sample_token_classifier.txt
CharacterSet=ISO-8859-1
OutputFile=sample_classifying_tokenizer.tok
```

5.4 Tokenizer Deployment

The application configuration file for classifying tokenizer specifies the following properties:

- `ResourceFile`: a path to the compiled classifying tokenizer resources
- `Input`: a path to the input file or input directory
- `Output`: a path to the output directory
- `Whitespaces`: is set to 'true' if whitespace tokens should be returned, or to 'false' otherwise (default setting)
- `CharacterSet`: the name of the character set used for decoding input file(s)
- `UseBorderSeparatorTrimming`: is set to 'true' in case word-boundary trimming option is on or to 'false' otherwise (default setting is 'true')

A sample application configuration file looks as follows.

```
ResourceFile=compiled_tokenizer.tok
CharacterSet=ISO-8859-1
UseBorderSeparatorTrimming=true
Input=./test/input/sample_file.txt
Output=./test/output
Whitespaces=false
```

In order to directly integrate the tokenizer in other applications the following piece of code could be used. It corresponds more or less to the above application configuration file (in a simplified form). Please see the Java API documentation for further details.

```
import java.util.ArrayList;
import it.jrc.lt.core.component.tokenizer.*;
import it.jrc.lt.core.component.*;
import piskorski.util.functions.*; // needed for the Files class
...

// Create an instance of a classifying tokenizer
AbstractTokenizer tokenizer = AbstractTokenizer.createInstance("classifyingTokenizer");

// Read tokenizer resource from file
tokenizer.readFromFile("compiled_tokenizer.tok");

// Set options (no whitespace tokens, border trimming on)
//
// Note that instead of using for each option a specific function call,
// a configuration object (a set of properties) representing the settings
// can be constructed and passed to a function, which configures the tokenizer.
// This is illustrated below.
//
Configuration conf = new Configuration();
conf.add("WhitespaceTokens","false");
conf.add("UseBorderSeparatorTrimming","true");
tokenizer.applySettings(conf);

// Read text from file
String input = Files.FileToString("sample_file.txt","ISO-8859-1");

// Tokenize
ArrayList<AbstractTokenItem> tokens = tokenizer.tokenize(input);

// Iterate over the tokens
//
// Note that types can be also returned as byte objects (t.getType())
for(AbstractTokenItem t : tokens)
  { System.out.println(t.getStart() + " " + t.getEnd() + " " + t.getTypeAsString());
    System.out.println("TEXT: " + t.getText(input)
  }
```

5.5 Remarks

A worthwhile extension of the tokenizer would be to allow for assigning a single token multiple token classes, e.g., natural numbers could be additionally subclassified according to the number of digits they consist of (two-digit numbers, four-digit numbers). The early identification of such detailed and eventually domain specific token classes might potentially further simplify the processing on subsequent levels like morphological analysis or named-entity recognition.

Currently, the token classifier is represented in a low-level finite-state format. In the future, it would be more convenient to provide the users with some tool for defining token classes on a somewhat more abstract level.

6 Morphological Analysis

6.1 Introduction

Morphological analysis usually refers to the process of the identification of a base form (*lemma*), syntactic category and other morphosyntactic features (e.g., case, gender, number, etc.) for a given word form. It can also be seen as a process of mapping words into linguistically interesting properties. For instance, an output of morphological analysis for the word *running* would be: `base form:run cat:verb` and `base:running cat:noun`, i.e., *running* can be either a verb or a noun. The additional morphological features strongly depend on the syntactic category, e.g., nouns have *case*, *gender* and *number*, whereas verbs have *tense* and *form*. The process of disambiguating syntactic category of a given word is called

part-of-speech tagging and might be a part of the morphological analysis too. However, in this report the term morphological analysis refers to the process of identifying all potential morphological readings of a given word in a text (disregarding the context it appears in).

Morphological information, e.g., lemma or part-of-speech, is extensively used in information extraction rules of all kinds, which makes morphological analyzer a core module in IE systems. For example, in many applications it is desirable to abstract away from inflectional variation, so that *attacked*, *attacks* and *attack* are all treated as the same word type. Conversely, it will sometimes come in handy to make use of richer information than that available in the raw text, so that *attacking* can be identified as the present participle of *attack*. The easiest and widely known method to implement a morphological analyzer is to construct a finite-state device (automaton or transducer) which accepts possibly all word forms of a given language and converts them into their corresponding base forms, part-of-speech tags, etc. [23, 24]. In such a case, morphological analysis consists of scanning a given text and performing an automaton look-up, retrieving morphological information for each word found in the text. This section introduces the morphological analyzer in CORLEONE, whose design follows this approach. First, a short overview of the morphological analyzer and its implementation is given. Subsequent sections describe how to prepare and compile morphological resources and how to perform morphological analysis on a given input text.

6.2 Morphological Analyzer

The CORLEONE morphological analyzer identifies for each word in a text: (a) its part-of-speech (syntactic category), (b) its base form, and (c) a set of feature-value pairs appropriate for the given syntactic category. If a word form is ambiguous, all interpretations are returned. For instance, the morphological analyzer would return at least the following feature structures for the word *attack*.

```
| start:0          |   | start:0          |
| end:5            |   | end:5            |
| pos:Noun         |   | pos:Verb         |
| type:common      |   | type:main        |
| gender:neuter    |   | form:infinitive  |
| number:singular  |
```

The first structure represents the noun reading of this word (common neuter singular noun), whereas the second represents a verb interpretation (infinitive form of a main verb).

In order to facilitate the problem of multilinguality, for encoding the morphosyntactic data we have chosen the format specified in the MULTEXT project [25]. This project has developed a set of generally usable software tools to manipulate and analyze text corpora, together with lexicons and multilingual corpora in several European languages. In particular, harmonized specifications for encoding computational lexicons have been established, i.e., same tagset and features are used for all languages, which makes it particularly interesting in the context of EMM dealing with texts in different languages. In addition, the outcome of MULTEXT are freely available full-form morphological lexica in several languages which we benefit of. More details on encoding the lexica will be presented in section 6.4.

The morphological analyzer provides an option to switch between case-sensitive and case-insensitive processing mode, where 'case-insensitive mode' means that the input data is temporarily downcased during processing. However, the entries in the lexicon themselves are not influenced, i.e., they are not downcased at any time. Further, it is possible to specify whether output structures for unrecognized words will be returned or not (for unknown words the additional tag - `U` has been introduced).

Before performing analysis on a given input text, the latter one has to be tokenized. The core version of the morphological analyzer is not tailored to any particular tokenizer. In order to use a specific tokenizer a 'piece of code' for converting the results of such a tokenizer into a simple array of generic tokens has to be provided – the method for applying the morphological analysis requires passing an array of generic tokens as argument. Generic tokens consist solely of the start/end positions of the tokens in the text and token type (which may be unspecified). For more details see the class `it.jrc.lt.core.component.tokenizer.AbstractTokenItem`.

Finally, for some languages the morphological analyzer handles language-specific phenomena since some words (e.g. German compounds) are not necessarily included in the full-form lexicon and therefore somewhat more advanced technique is indispensable in order to identify their morphosyntactic features. The language-specific component is fired when simple lexicon look-up fails and provided that: (a) such

component exists for a given language and (b) the classifying tokenizer is being used. Currently, we have developed a language-specific component for English and Italian.

The component for English: (a) tries to morphologically analyze hyphenated words (they can not be simply lexicalized since the process of constructing such words is very productive), and (b) joins some tokens in order to treat clitics properly (e.g., *He's* is recognized as two tokens: *He* and *'s* since there is an entry for *'s* in the MULTEXT lexicon). The first task is done via application of semi-automatically created patterns (ca. 500) which map combinations of part-of-speech information of single elements of hyphenated words into a part-of-speech information of the whole hyphenated word, e.g., `U-V -> A` means that hyphenated word consisting of an unknown word and a verb is an adjective. *Shiite-dominated* is an example of a word which would be matched by this pattern. The second task is done via utilization of a list of most frequent clitics in English.

As for Italian, the major problem to be solved are clitics and words, which include apostrophes. The former ones are not handled at the moment, whereas the words with apostrophes are processed as follows. First, the token classes for words with apostrophes are defined in such a way that all such words are matched without any constraints. Secondly, tokens representing words with apostrophes are split into two subtokens. Both subtokens are morphologically analyzed, i.e., a dictionary look-up is performed. Finally, in case one of the subtokens is 'still' unknown a bunch of part-of-speech based heuristics is applied on these morphologically annotated subtokens in order to guess the missing information. For instance, the rule `A(#gen,#num) U --> A(#gen,#num) N(#gen,#num)` represents the following heuristic: if the unknown subtoken (`U`) is preceded by an apostrophized subtoken, which is an adjective (`A`), then it is tagged as a noun `N`. Further, the gender and number information (`#gen`, `#num`) assigned to the noun is copied from the subtoken representing the adjective.

Other languages are not supported yet with respect to component performing specific treatment like the one described above. Noteworthy, language specific components are hardcoded, i.e., they can not be modified or manipulated by CORLEONE users.

Please note also, that the morphological analyzer is Unicode-aware, i.e., any character set can be used for encoding the raw resources and any Unicode input stream can be used as an input to the morphological analyzer.

6.3 Implementation

The CORLEONE morphology is implemented as a single finite-state automaton which encodes all lexicon entries. One of the common techniques for squeezing finite-state devices in the context of implementing dictionaries is an appropriate coding of the input data and turning it into a minimal acyclic deterministic finite-state automaton [24]. If the dictionary contains solely contemporary word forms, a very good compression rate can be expected, since many words share prefixes and suffixes, which leads to a high degree of transition sharing in the corresponding automaton. If the keywords are associated with additional annotations (tags) representing certain categories, attribute-value pairs, then an adequate encoding of such information is necessary in order to keep the corresponding automaton small. A simple solution is to reorder the tags from the most specific to the most general ones or to precompute all possible tag sequences for all entries and to replace them with a numerical index [26, 24]. Both aforementioned techniques work well in case of compressing morphological data. Consider now, an entry for the word *striking* in the morphological lexicon consisting of an inflected word form, a lemma and a morphosyntactic tag, i.e., `striking-strike-Vmpp`, where `Vmpp` stands for main verb and present participle. Obviously, the sequence `striking-strike` is unique. Through the exploitation of the word-specific information the inflected form and its base form share, one can introduce patterns describing how the lexeme can be reconstructed from the inflected form, e.g., `3+e` - delete three terminal characters and append an `e` (`striking -> strik + e`), which would result in better suffix sharing, i.e., the suffix `3+e Vmpp` is more frequently shared than `strike Vmpp`. We apply a bag of such patterns in order to achieve high degree of suffix sharing. Once the raw lexicon entries are converted into a new format via application of these patterns, we subsequently compile them into a single minimal deterministic automaton by applying the novel incremental algorithm for converting a list of strings into a minimal deterministic acyclic automaton, which runs in nearly linear time [27]. The aforementioned algorithm outperforms the traditional method of splitting the whole process into two steps, namely constructing an automaton and performing minimization thereof. The major bottleneck of the classical approach is that in the first phase an automaton is constructed which is not optimized in terms of space complexity and might not necessarily fit into a main memory before performing the minimization step, whereas in the novel incremental algorithm

the intermediate automaton is always minimal with respect to the entries already being included in the automaton. A detailed description of this novel and particularly useful algorithm is given in [19].

6.4 Preparation of Morphological Resources

The resources needed for compiling a morphological analyzer consist of two files, an entry file and a morphology compilation configuration file. They will be described in the next subsections.

6.4.1 Entry File

The entry file contains a list of full word forms encoded in MULTEXT format [28]. For each entry there is a separate line of the following form:

```
word-form <TAB> lemma <TAB> morphosyntactic-description
```

If a word form is ambiguous, for each reading a separate line has to be added. For the sake of clarity, we briefly describe the format of `morphosyntactic-description`. It is just a sequence of characters, where each single character encodes a value of certain morphosyntactic feature for a given word form. The first character always denotes the part of speech information. In MULTEXT there are 14 part-of-speech tags for all languages. They are listed below.

Noun	N	Adposition	S
Verb	V	Conjunction	C
Adjective	A	Numeral	N
Pronoun	P	Interjection	I
Determiner	D	Residual	X
Article	T	Abbreviation	Y
Adverb	R	Particle	Q

Each character at position $1, 2, \ldots, n$ encodes a value of some attribute appropriate for the part-of-speech tag at position 0. For instance, appropriate attributes for nouns are among others: gender, number and case. If an attribute does not apply (e.g., is not applicable to a particular language or is not applicable to a particular combination of feature-values), the corresponding position in the morphosyntactic description contains a hyphen, i.e., `-`. It is important to note that trailing hyphens are usually omitted in the morphosyntactic description (it is not harmful to include them, but not necessary). The following example of encoding the word *attack* gives an idea of the MULTEXT format.

```
attack  attack  Ncns
attack  attack  Vmn
attack  attack  Vmip-p
attack  attack  Vmip1s
attack  attack  Vmip2s
```

The first line corresponds to a noun reading (`N`), where `cns` stands for: common, neuter and singular respectively. The remaining lines refer to a verb reading (`V`), where `mn` stands for main verb in infinitive, `mip-p` stands for an indicative form of a main verb in present tense in plural (person feature is set to `-` since any person is valid), and the two last morphosyntactic tags `mip1s` and `mip2s` denote the singular version for the first and second person of the same kind of verb as the latter one.

Below we give a list of attributes appropriate for each syntactical category. Please note that for nouns and verbs we have added two additional attributes, namely `SEM_1` and `SEM_2` for encoding additional domain-specific information. These attributes are not part of the original MULTEXT tagset.

```
Noun            N
----------------
0 Type
1 Gender
2 Number
3 Case
4 Definiteness
5 Clitic
6 Animate
7 Owner_number
8 Owner_Person
9 Owned_Number
10 Sem_1
11 Sem_2

Verb            V
----------------
0 Type
1 VForm
2 Tense
3 Person
4 Number
5 Gender
6 Voice
7 Negative
8 Definiteness
9 Clitic
10 Case
11 Animate
12 Clitic_s
13 Aspect
14 Courtesy
15 Sem_1
16 Sem_2

Adjective       A
----------------
0 Type
1 Degree
2 Gender
3 Number
4 Case
5 Definiteness
7 Clitic
8 Animate
9 Formation
10 Owner_Number
11 Owner_Person
12 Owned_Number
```

```
Pronoun         P
----------------
0 Type
1 Person
2 Gender
3 Number
4 Case
5 Owner_Number
6 Owner_Gender
7 Clitic
8 Referent_Type
9 Syntactic_Type
10 Definiteness
11 Animate
12 Clitic_s
13 Pronoun_Form
14 Owner_Person
15 Owned_Number
16 Wh_Type

Determiner      D
----------------
0 Type
1 Person
2 Gender
3 Number
4 Case
5 Owner_Number
6 Owner_Gender
7 Clitic
8 Modific_Type
9 Wh_Type

Article
----------------
0 Type
1 Gender
2 Number
3 Case
4 Clitic
5 Animate
```

```
Adverb          R
----------------
0 Type
1 Degree
2 Clitic
3 Number
4 Person
5 Wh_Type

Adposition      S
----------------
0 Type
1 Formation
2 Case
3 Clitic
4 Gender
5 Number

Conjunction     C
----------------
0 Type
1 Formation
2 Coord_Type
3 Sub_Type
4 Clitic
5 Number
6 Person

Numeral         M
----------------
0 Type
1 Gender
2 Number
3 Case
4 Form
5 Definiteness
6 Clitic
7 Class
8 Animate
9 Owner_Number
10 Owner_Person
11 Owned_Number
```

```
Interjection    I
----------------
0 Type
1 Formation

Abbreviation    Y
----------------
0 Syntactic_Type
1 Gender
2 Number
3 Case
4 Definiteness

Particle        Q
----------------
0 Type
1 Formation
2 Clitic

Residual        X
----------------
= no attributes =
```

For a more comprehensive description of appropriate attributes for each single part-of-speech category please refer to the MULTEXT tagset specifications [28]. Note also that there are some subtle differences with the respect to the order of attributes in original MULTEXT resources. We have transformed the original resources in such a way that attribute order is identical for all languages.

Please note once more that in case of unknown words the morphological analyzer returns a feature structure with a part-of-speech value set to `U`.

6.4.2 Compilation Configuration File

Once the entry file is provided, we can create a morphology compilation configuration file, which defines the following properties:

- `Name`: the name of the morphology
- `EntryFile`: a path to the entry file
- `CharacterSet`: the name of the character set used for decoding the entry file
- `LanguageCode`: a language code
- `OutputFile`: a path to the output file, which constitutes the only resource needed for applying the morphology

The example given below illustrates how a configuration file should look.

```
Name=myMorphology
EntryFile=sample_entries.txt
CharacterSet=UTF-8
LanguageCode=1
OutputFile=sample_morphology_compiled.mor
```

The language code is mainly used to fire the appropriate language-specific component in order to handle words not found in the full form lexicon that might constitute valid word forms, etc. Currently following language codes are available:

```
1 - ENGLISH        13 - ICELANDIC     25 - SLOVENE       37 - BOSNIAN
2 - FRENCH         14 - RUSSIAN       26 - TURKISH       38 - BYELORUSSIAN
3 - GERMAN         15 - ESTONIAN      27 - GREEK         39 - MOLDOVIAN
4 - POLISH         16 - LATVIAN       28 - ALBANIAN
5 - ITALIAN        17 - LITHUANIAN    29 - PORTUGUESE
6 - DUTCH          18 - CZECH         30 - ARABIC
7 - SPANISH        19 - UKRAINIAN     31 - PERSIAN
8 - BULGARIAN      20 - HUNGARIAN     32 - SLOVAK
9 - FINNISH        21 - ROMANIAN      33 - CATALAN
10 - SWEDISH       22 - MACEDONIAN    34 - MALTESE
11 - NORWEGIAN     23 - SERBIAN       35 - CHINESE
12 - DANISH        24 - CROATIAN      36 - INDONESIAN
```

6.5 Deploying Morphological Analyzer

The application configuration file for the morphological analyzer specifies the following properties:

- `ResourceFile`: a path to the compiled morphological resource
- `Input`: a path to the input file or input directory
- `Output`: a path to the output directory
- `CaseSensitive` is set to 'true' if the morphological analyzer will be applied in case-sensitive mode or to 'false' otherwise (default setting is 'false')
- `CharacterSet`: the name of the character set used for decoding input files
- `OutputForUnknownWords`: is set to 'true' if output structures for unknown words should be returned or to 'false' otherwise (default setting is 'true')
- `TokenizerResourceFile`: a path to the file containing compiled resources for the classifying tokenizer (please note that in practice any tokenizer can be used with the morphology - see the JAVA API documentation)

A sample application configuration file looks as follows.

```
ResourceFile=sample_morphology_compiled.mor
TokenizerResourceFile=sample_classifying_tokenizer.tok
Input=./test/input/sample_file.txt
Output=./test/output
CharacterSet=UTF-8
CaseSensitive=false
OutputForUnknownWords=true
```

In order to directly integrate the morphological analyzer in other application the following piece of code could be used. It corresponds more or less to the above application configuration file (in a simplified form). Please see the Java API documentation and the source code for further details.

```
import java.util.ArrayList;
import it.jrc.lt.core.component.morphology.*;
import it.jrc.lt.core.component.tokenizer.*;
import piskorski.util.functions.*; // needed for the Files class

// Create an instance of a MULTEXT morphology
AbstractMorphology morphology = AbstractMorphology.createInstance("multextMorphology");

// Read morphology resource from file
morphology.readFromFile("sample_morphology_compiled.mor");

// Create an instance of a classifying tokenizer
AbstractTokenizer tokenizer = AbstractTokenizer.createInstance("classifyingTokenizer");

// Read tokenizer resource from file
tokenizer.readFromFile("sample_classifying_tokenizer.tok");

// Set options
morphology.returnOutputForUnknownWords();
morphology.swithOffCaseSensitiveMode()

// Read the input file
String input = Files.FileToString("sample_file.txt","UTF-8");

// Apply the tokenizer
ArrayList<AbstractTokenItem> tokens = tokenizer.tokenize(input);

// Apply the morphology
ArrayList<AbstractDisjunctionOfMorphologyItems> morphologyItems
                                                = morphology.findMatch(tokens, input);

// Iterate over disjunctions of morphology items
for(AbstractDisjunctionOfMorphologyItems it : morphologyItems)
 { int numItems = it.getNumberOfItems();
   for(int k=0;k<numItems;k++)
    { AbstractMorphologyItem nextItem = it.getItem(k);
      System.out.println("Analysis for: " + nextItem.getText(input));
      System.out.println(nextItem.toString());
    }
 }
```

6.6 Remarks

Currently, CORLEONE comes only with morphological resources for English, German and Italian. However, compiling resources for other languages, provided that they are encoded according to MULTEXT conventions, is a straightforward task. Envisaged future extensions to the morphology component encompass additional language-specific components for languages other than English and Italian and some mechanism for performing some partial morphosyntactic disambiguation.

7 Gazetteer Look-up Tool

7.1 Introduction

The term gazetteer usually refers to a dictionary that includes geographically related information on given places, e.g., data concerning the makeup of a country, region or location. In the NLP community, it has a broader meaning. It refers to a list of not only geographical references, but also names of various types of entities and named expressions, e.g., people, organizations, months of the year, currency units, company designators and other similar keywords. Gazetteer look-up[3] is usually seen as an autonomous process of linguistic analysis and plays a crucial role in the process of named-entity recognition and is heavily deployed for solving other information extraction tasks.

There are several well-established techniques and data structures that can be used to implement a gazetteer, e.g., hashing, tries and finite-state automata. Some studies on real-world data revealed that finite-state automata seem to be a good choice, since they require less memory than alternative techniques and at the same time guarantee efficient access to the data [29, 26, 23, 24]. This section introduces the CORLEONE gazetteer look-up component based on finite-state technology. The rest of this section is organized as follows. First, a short overview of the gazetteer component and its implementation is given. Subsequently, the process of preparing and compiling raw gazetteer resources is described. Next, the application of the gazetteer look-up component on a given input data is addressed. Finally, some current issues and ideas concerning future developments are pinpointed.

7.2 Gazetteer Look-up Component

The CORLEONE gazetteer look-up (dictionary look-up) component matches an input stream of characters or tokens against a gazetteer (dictionary) list, and produces an adequate annotation for the matched text fragment. It allows for associating each entry in the gazetteer with a list of arbitrary flat attribute-value pairs. Further, ambiguous entries are allowed too. For example, the entry *New York's* could be associated with the following attribute-value pairs:

```
concept:new york
type:city
continent:north america
case:genitive
```

Matching an occurrence of the phrase *New York's* would yield a feature structure which looks as follows:

```
| start:23                |
| end:30                  |
| concept:new york        |
| type:city               |
| continent:north america |
| case:genitive           |
```

The values of the attributes `start` and `end` correspond to the start and end positions of the matched text fragment. These values may either refer to character positions in the text or token IDs (i.e., first and last token of the matched text fragment). One can switch between these two options while using the component. Furthermore, the user is free to use any attributes and naming conventions he likes. Please note also, that the gazetteer is Unicode-aware, i.e., any character set can be used for encoding the raw resources and any Unicode input stream can be used as an input to the gazetteer. In order to avoid potential clashes in the file for encoding resources, the specific symbols used as separators can be chosen arbitrarily too.

The gazetteer provides an option (deployment option) to switch between case-sensitive and case-insensitive processing mode, where 'case-insensitive mode' means that the input data is temporarily downcased during processing. However, the gazetteer entries themselves are not influenced, i.e., they are not downcased at any time. On the other hand, there is a compilation option, which allows for downcasing all keywords in the gazetteer at compile-time.

[3] Some use the term dictionary look-up instead

Before applying the gazetteer to a given input text, the input has to be tokenized. The gazetteer is not tailored to any particular tokenizer. Similarly to morphology component, in order to use a specific tokenizer a method for converting the results of such a tokenizer into a simple array list of generic tokens has to be provided.

There are two processing styles in which gazetteer can be applied: (a) longest matching – only the longest character sequence at a given position in the text that is covered by the gazetteer is matched and no overlapping matches with different starting positions are allowed[4], and (b) full matching – all character sequences that are covered by the gazetteer at any position, which is the beginning of a token, are matched.

7.3 Implementation

This subsection elaborates on the implementation aspects of the gazetteer, which has been developed via utilization of state-of-the-art finite-state technology.

As already mentioned in 6.3 a common technique for implementing dictionaries is an appropriate coding of the input data which allows for turning it into a minimal acyclic deterministic finite-state automaton [24], so that the morphological analysis boils down to a simple automaton look-up. Unfortunately, in case of gazetteers, this strategy can not be applied in a straightforward manner like described in 6.3, since the attribute values often do not share any prefixes or suffixes with the keyword, i.e., the major part of a string that encodes a single gazetteer entry and its tags might be unique and this could potentially blow up the corresponding automaton enormously.

Before, we present our encoding technique, let us first mention that the raw gazetteer resources are represented simply by a text file, where each line represents a single gazetteer entry in the following format: `keyword (attribute:value)+`. For each reading of an ambiguous keyword, a separate line is introduced. For the word *Washington* the gazetteer could potentially include the following entries:

```
Washington | type:city | variant:WASHINGTON | location:USA
           | full-name:Washington D.C. | subtype:cap_city
Washington | type:person | gender:m_f | surname:Washington
           | language:english
Washington | type:organization | subtype:commercial
           | full-name:Washington Ltd. | location:Canada
Washington | type:region | variant:WASHINGTON | location:USA
           | abbreviation: W.A. | subtype:state
```

For the sake of explaining the compression strategy, we differentiate between open-class and closed-class attributes, depending on their range of values, e.g., `full-name` is intuitively an open-class attribute, whereas `gender` is a closed-class attribute. In practice, we use a more fuzzy definition, i.e., all attributes whose corresponding value set contains more than 512 elements is always considered to be an open-class attribute, whereas other attributes may be either defined as open-class or closed-class attributes by the user. The choice between open-class and closed-class attributes has an impact on the compression method and as a consequence on the compression rate.

Our main idea behind transforming a gazetteer into a single automaton is to split each gazetteer entry into a disjunction of subentries, each representing some partial information. For each open-class attribute-value pair present in an entry a single subentry is created, whereas closed-class attribute-value pairs (or a subset of them) are merged into a single subentry and rearranged in order to fulfill the *first most specific, last most general* criterion. In our example, the entry for the word *Washington* (city) yields the following partition into subentries:

```
Washington #1 NAME(subtype) VAL(cap_city) NAME(type) VAL(city)
Washington #1 NAME(variant) WASHINGTON
Washington #1 NAME(location) USA
Washington #1 NAME(full-name) Washington D.C.
```

where `NAME` maps attribute names to single univocal characters not appearing elsewhere in the original gazetteer and `VAL` denotes a mapping which converts the values of the closed-class attributes into single characters representing them. The string `#1`, where `#` is again a unique symbol, denotes the reading

[4]With the restriction that a match can neither start nor end in the middle of a token

index of the entry (first reading for the word *Washington*). Gazetteer resources converted in this manner are subsequently compiled into a single automaton via the application of the incremental algorithm for converting a list of strings into a minimal acyclic DFSA in linear time [27], which was mentioned earlier.

In order to gain better compression rate we utilized formation patterns for a subset of attribute values appearing in the gazetteer entries, i.e., patterns which describe how attribute values can be constructed from the keywords. For instance, frequently, attribute values are just the capitalized form or the lowercase version of the corresponding keywords, as can be seen in our example. Such a pattern can be represented by a single character. Further, keywords and some attribute values often share prefixes or suffixes, e.g., *Washington* vs. *Washington D.C.* Next, there are clearly several patterns for forming acronyms or abbreviations from the full form, e.g., *ACL* can be derived from *Association of Computational Linguistics*, by simply concatenating all capitals in the full name. We benefit from such formation rules and deploy other related patterns in a similar way in order to further reduce the space requirements. Nevertheless, some part of the attribute values can not be replaced by patterns. Applying formation patterns to our sample entry would result in:

```
Washington #1 NAME(subtype) VAL(cap_city) NAME(type) VAL(city)
Washington #1 NAME(variant) PATTERN(AllCapital)
Washington #1 NAME(location) USA
Washington #1 NAME(full-name) PATTERN(Identity) D.C.
```

where `PATTERN` maps pattern names to unique characters not appearing elsewhere in the gazetteer.

The outlined method of representing a gazetteer is an elegant solution and exhibits two major assets: (a) we do not need any external table for storing/accessing attribute values, since all data is encoded in the automaton, which means faster access time, and (b) as a consequence of the encoding strategy, there is only one single final state in the automaton. The states having outgoing transitions labeled with the unique symbols in the range of `NAME` are implicit final states (there is no need for explicit distinction between accepting and non-accepting states). The right languages of these states represent attribute-value pairs attached to the gazetteer entries.

For implementing finite-state automata themselves, we deploy the highly compact transition-list model described in [30]. In this model states are not stored explicitly and each transition is represented solely as quintuple consisting of a transition label, three bits marking: (a) whether the transition is final, (b) whether it is the last transition of the current state and (c) whether the first transition of the target state is the next one in the transition list, and a (possibly empty) pointer to the first outgoing transition of the target state of the transition. This representation is easy to implement and constitutes a good trade-off between space and time complexity [30].

Further details of the encoding approach outlined in this section as well as other more advanced gazetteer compression techniques, e.g., transition jamming, utilization of numbered finite-state automata, Ziv-Lempel style-like gazetteer substructure recognition etc., are described in more detail in [13, 14].

7.4 Compilation of Gazetteer Resources

The resources needed for compiling a gazetteer consist of three files, an attribute file, an entry file and a gazetteer compilation configuration file. They will be described one by one in the next subsections.

7.4.1 Attribute File

The attribute file simply lists all attribute names, where each line stands for a single attribute name. In order to define an attribute as an open-class attribute (see 7.3 for definition) its name has to be preceded by an asterisk. The following example gives an impression of what an attribute file would look like.

```
*concept
type
continent
case
```

The attributes `type`, `continent` and `case` are closed-class attributes, whereas the attribute `full-name` is an open-class attribute. Differentiating between these two types of attributes does not impact the output of the gazetteer, but it plays an important role in the process of compilation and compression of gazetteer resources. Specifying an attribute as an open-class (even if it intuitively seems to be a

closed-classed attribute) might result in a different size of the compiled gazetteer resource. In particular, open-class attribute values undergo an inspection whether any of the formation patterns is applicable. In order to explicitly skip such inspections, add second asterisk in front of the attribute name. Our sample attribute file would then look like follows.

```
**concept
type
continent
case
```

Using this option makes sense if it is intuitively clear that formation patterns will not be applicable (e.g. numerical attributes). At this stage, please note once more that defining an attribute as closed-class attribute which has more than 512 possible values will result in a run-time compilation error.

7.4.2 Entry File

The entry file contains the proper gazetteer entries. Each line in this file includes an entry and a list of associated attribute-value pairs. Each attribute-value pair is separated by a symbol which does not occur elsewhere in this file. Similarly, attributes and values are separated by other unique symbol also not appearing elsewhere in this file. The aforementioned symbols can be defined arbitrarily by the user. The example below illustrates an entry file which corresponds to the example of an attribute file given in 7.4.1. The two separator symbols are `|` and `:` respectively.

```
Afghanistan's | type:country | case:genitive | concept:afghanistan
```

Entry files may include comments and empty lines. A comment line is preceded with a `#` symbol.

7.4.3 Compilation Configuration File

Once the entry file and attribute file are provided, we can create a gazetteer configuration file, which define the following properties:

- `Name`: a name of the gazetteer
- `AttributeFile`: a path to the attribute file
- `EntryFile`: a path to the entry file
- `CharacterSet`: the name of the character set used for decoding input files
- `InputSeparator`: a special symbol for separating attribute-value pairs
- `AttributeValueSeparator`: a special symbol for separating attribute names from the corresponding values
- `DownCaseEntries`: is set to 'true' in case the entries in the entry file should be downcased during the compilation process, or to 'false' otherwise (default setting is 'false')
- `OutputFile`: a path to the output file, which constitutes the only resource needed for applying the gazetteer

The example given below illustrates how a configuration file should look like.

```
Name=myGazetteer
AttributeFile=sample_features.txt
EntryFile=sample_entries.txt
CharacterSet=UTF-8
InputSeparator=|
AttributreValueSeparator=:
DownCaseEntries=false
OutputFile=sample_gazetteer.gaz
```

7.5 Gazetteer Deployment

The application configuration file for the gazetteer specifies the following properties:

- `ResourceFile`: a path to the compiled gazetteer resources
- `Input`: a path to the input file or input directory
- `Output`: a path to the output directory
- `CaseSensitive`: is set to 'true' (default setting) if the gazetteer will be applied in case sensitive node, or to 'false' otherwise
- `CharacterSet`: the name of the character set used for decoding input files
- `OutputTokenPositions`: is set to 'true' if token positions should be returned, or to 'false' (default setting) in case character positions should be returned
- `SearchStrategy`: search strategy option; currently two options are available: `longest-match` which is the default one, and `all-matches`, which means that all matches at a given positions are returned

Our sample application configuration file would look as follows (with the options: case-sensitive mode, longest-match searching strategy and returning character positions)

```
ResourceFile=sample_gazetteer.gaz
Input=./test/input/sample_file.txt
Output=./test/output
CaseSensitive=true
CharacterSet=UTF-8
OutputTokenPositions=false
SearchStrategy=longest-match
```

In order to directly integrate the gazetteer in other application the following piece of code could be used. It corresponds to the above application configuration file (in a simplified form).

```
import java.util.ArrayList;
import it.jrc.lt.core.component.*;
import it.jrc.lt.core.component.gazetteer.*;
import it.jrc.lt.core.component.tokenizer.*;
import piskorski.util.functions.*; // needed for the Files class

// Create an instance of a gazetteer
AbstractGazetteer gazetteer = AbstractGazetteer.createInstance("basicGazetteer");

// Read gazetteer resource from file
gazetteer.readFromFile("sample_gazetteer.gaz");

// Set gazetteer options
gazetteer.returnCharacterPositions();
gazetteer.caseSensitiveModeActive();
gazetteer.setSearchStrategy(SearchStrategy.LONGEST_MATCH);

// Create an instance of a basic tokenizer and read resources from file
AbstractTokenizer tokenizer = AbstractTokenizer.createInstance("basicTokenizer");
tokenizer.readFromFile("sample_tokenizer.tok");

// Read the input file
String input = Files.FileToString("sample_file.txt","UTF-8");

// Apply the tokenizer
ArrayList<AbstractTokenItem> tokens = tokenizer.tokenize(input);

// Apply the gazetteer
```

```
ArrayList<AbstractDisjunctionOfGazetteerItems> gazetteerItems
                                                    = gazetteer.findMatch(tokens, input);

// Iterate over disjunctions of gazetteer items
for(AbstractDisjunctionOfGazetteerItems disj : gazetteerItems)
  { int numItems = disj.getNumberOfItems();
    // Iterate over gazetteer items
    for(int k=0;k<numItems;k++)
      { AbstractGazetteerItem item = it.getItem(k);
        // Note that character positions are returned (we can use getText() method)
        System.out.println("FOUND: " + item.getText(input));
        // iterate over attribute-value pairs in the current gazetteer item
        int len = item.getSize();
        for(int i=0;i<len;i++)
         { avPair = item.getAVPair(i)
           System.out.println(avPair.getAttributeName() + "=" + avPair.getValue());
         }
      }
 }
```

7.6 Remarks

There are still some improvements concerning time complexity, which can be addressed in the future. Firstly, the current version of the gazetteer does not compress numerical data in an optimal way. It works much better for string-valued attributes since the primary goal was to compress such kind of data. In order to achieve better performance when dealing with numerical data the technique based on numbered automata [13] could be potentially implemented to alleviate the problem. Further improvements with respect to space complexity could be achieved via utilization of some additional advanced compression techniques, e.g., transition jamming, relative addressing in data structure implementing FSAs, Ziv-Lempel style-like substructure recognition in the gazetteers. They are described in more detail in [13, 14].

For some applications list-valued attributes might come in handy. Therefore, it would be worth supporting such option in future versions of the software.

Compression rate and speed might depend on: (a) how the attributes are classified (open-class vs. closed-class attributes) and (b) settings in the compilation configuration file. Experiment with both to obtain the best results.

8 Sentence Splitter

8.1 Introduction

The segmentation of running input text into sentences or utterances is one of the crucial preprocessing steps in the process of information extraction. Since punctuation marks are not used exclusively to mark sentence breaks, sentence and utterance boundaries are ambiguous. Both knowledge-based and machine learning approaches to sentence boundary detection have been widely studied and used. CORLEONE sentence splitter follows the rule-based approach, and is based on a user-definable list of potential sentence boundary markers, a list of non-final text items (e.g. non-final abbreviations), a list of non-initial text items (e.g., some capitalized abbreviations which do not start a sentence), and a list of non-initial prefixes (e.g., symbols like '(' or '%'). For the sake of clarity, we give below an example of such resources for English.

- `Potential sentence boundary markers`: . ! ? "
- `Non-final text items`: *Prof.*, *Mr.*, *e.g.*
- `Non-initial text items`: *Ltd.*, *Mln.*
- `Non-initial prefixes`: ([{ <

Note that non-initial prefixes do not necessarily have to be followed by a whitespace, whereas non-initial text items must be followed by a whitespace, i.e., they must constitute tokens, whereas non-initial prefixes might be just a part of a token. For this reason we make the distinction between these two lists.

For the sake of clarity, we briefly sketch the sentence splitting algorithm. Firsty, the potential sentence boundary markers are identified. Next, those of them, which constitute trailing symbols in non-final text items found in the current text (e.g., abbreviations requiring an argument on the right), are excluded from further processing. Finally, several heuristics are applied in order to disambiguate the role of the remaining potential sentence boundary markers. These heuristics utilize the lists of non-initial items and non-initial prefixes. Note, that each whitespace symbol sequence, which includes two consecutive end-of-line symbols is automatically recognized as sentence/utterance boundary.

The sentence splitter applied to an input text returns an array list of sentence items, each consisting of two attributes representing the start and end position of the sentence. There is an option of returning either character positions or token-based positions, i.e., token IDs.

Before performing sentence splitting of a given input text, the latter one has to be tokenized. The sentence splitter is not tailored to any particular tokenizer. Analogously to morphology and dictionary look-up component, a method for converting the results of such a tokenizer into a simple array list of generic tokens has to be provided.

The implementation of the sentence splitter exploits the dictionary look-up component described previously. Therefore, we do not describe it here in more detail.

8.2 Compilation of Sentence Splitter Resources

The resources needed for compiling CORLEONE sentence boundary splitter consist of four files containing respectively potential sentence boundary markers, non-final text items, non-initial text items, and non-initial prefixes. Note that, each line in these files contains a single entry. When these four files are provided, we also need a sentence boundary compilation configuration file, which defines the following properties:

- `Name`: the name of the sentence splitter
- `BoundaryMarkersFile`: a path to the file containing the list of potential boundary markers
- `NonFinalItemsFile`: a path to the file containing the list of non-final items
- `NonInitialItemsFile`: a path to the file containing the list of non-initial items
- `NonInitialPrefixFile`: a path to the file containing the list of non-initial prefixes
- `CharacterSet`: the name of the character set used for decoding input files
- `OutputFile`: a path to the output file containing the binary compressed representation of the resources for the sentence boundary splitter

The example given below illustrates a sample compilation configuration file for the sentence splitter.

```
Name = mySentenceSplitter
BoundaryMarkersFile=EN-BoundaryMarkers.txt
NonFinalItemsFile=EN-NonFinalItems.txt
NonInitialItemsFile=EN-NonInitialItems.txt
NonInitialPrefixFile=EN-NonInitialPrefix.txt
CharacterSet=UTF-8
OutputFile=sample_sentence_splitter.ssb
```

8.3 Sentence Splitter Deployment

The application configuration file for the sentence splitter specifies the following properties:

- `ResourceFile`: a path to the compiled sentence splitter resources
- `Input`: a path to the input file or input directory

- `Output`: a path to the output directory
- `CharacterSet`: the name of the character set used for decoding input files
- `TokenPositions`: is set to 'true' (default setting) if token positions should be returned in sentence items or to 'false' in case character positions should be returned

Our sample application configuration file would look as follows (with the option: return character positions)

```
ResourceFile=sample_sentence_splitter.ssb
Input=./test/input/sample_file.txt
Output=./test/output
CharacterSet=UTF-8
TokenPositions=false
```

Noteworthy, there is no need of specifying the resources for the tokenizer since the script-based version of the sentence splitter automatically uses the basic tokenizer described earlier in this report.

In order to directly integrate the sentence splitter in other application the following piece of code could be used. It corresponds to the above application configuration file (in a simplified form).

```
import java.util.ArrayList;
import it.jrc.lt.core.component.*;
import it.jrc.lt.core.component.tokenizer.*;
import it.jrc.lt.core.component.sentencesplitter.*;
import piskorski.util.functions.*; // needed for the Files class

// Create an instance of a sentence splitter
AbstractSentenceSplitter splitter = AbstractSentenceSplitter.createInstance("sentenceSplitter");

// Read sentence splitter resources from a file
splitter.readFromFile("sample_sentence_splitter.ssb");

// Set sentence splitter options
splitter.switchOffReturningTokenPositions();

// Create an instance of a basic tokenizer and read resources from file
AbstractTokenizer tokenizer = AbstractTokenizer.createInstance("basicTokenizer");
tokenizer.readFromFile("sample_tokenizer.tok");

// Read the input file
String input = Files.FileToString("sample_file.txt","UTF-8");

// Apply the tokenizer
ArrayList<AbstractTokenItem> tokens = tokenizer.tokenize(input);

// Apply the sentence splitter
ArrayList<AbstractSentenceItem> sentenceItems = splitter.segment(input, tokens);

// Iterate over sentence items
for(AbstractSentenceItem it : sentenceItems)
   // Note that character positions are returned (we can use getText() method)
   System.out.println("Sentence: " + it.getText(input));
```

9 String Distance Metrics Library

9.1 Introduction

A frequently appearing problem in the context of text processing technologies involves making a decision whether two distinct strings refer to the same real-world object. This problem is also referred to as name matching. For instance, the text phrases *Dr. Jan Kowalski*, *Kowalski* and *Kowalskiego Jana* might

constitute different textual mentions of the same person. Clearly, it would be useful to identify all these text fragments as references to the same person. One of the basic approaches to tackle the problem of name matching in a 'software engineering' manner is utilization of so called string distance metrics.

Distance metrics map a pair of strings s and t to a real number r, where a smaller value of r indicates greater similarity. Frequently, the term 'string similarity metrics' is used as well. There is only one slight difference to distance metrics, namely, that string similarity metrics maps pair of strings to a real number, where a smaller number indicates greater dissimilarity. The CORLEONE string distance library contains mainly metrics applied by the database community for record linkage [31, 32, 33, 34]. The next subsections describe the metrics included in the library and some tools and code samples of how to use them. It is important to note that for the sake of introducing some of the metrics in the remaining subsections we define them as string similarity metrics since they have been defined in this way in the literature. However, all metrics in CORLEONE are implemented as string distance metric.[5]

9.2 String Distance Metrics

The string distance metrics in CORLEONE are subdivided into five categories:

- edit-distance metrics
- *Jaro* and *Jaro-Winkler* metrics
- metrics based on character-level N-grams
- metrics based on common substrings
- recursive metrics, which are designed to cope with multi-token strings, and use other metrics as subroutines

In the rest of this section each of them is briefly described. Some explorations of knowledge poor techniques for lemmatization of proper names and name matching in highly inflective languages, which utilize the CORLEONE library of distance metric is presented in [35, 36].

9.2.1 Edit Distance Metrics

The first type of string distance metrics are so called edit-distance metrics. The point of departure constitutes the well-known *Levenshtein* edit distance metric given by the minimum number of character-level operations: insertion, deletion, or substitution, which are needed for transforming one string into the other [37]. There are several extensions to this basic metric. The *Needleman-Wunsch* [38] metric modifies the original one in that it allows for variable cost adjustment to the cost of a gap, i.e., insert/deletion operation and variable cost of substitutions. Another variant is the *Smith-Waterman* metric [39], which additionally uses an alphabet mapping to costs. There are several settings for this metric, e.g., the *Smith-Waterman* score can be normalized with the length of the shorter string (default) or *Dice coefficient*, i.e., the average length of strings compared. A further extension of the *Smith-Waterman* metric introduces two extra edit operations, *open gap* and *end gap*. The cost of extending the gap is usually smaller than the cost of opening a gap, and this results in small cost penalties for gap mismatches than the equivalent cost under the standard edit distance metrics. We will refer to the aforesaid metric as *Smith-Waterman with Affine Gaps*. In general, the computation of most edit-distance metrics requires $O(|s| \cdot |t|)$ time. The recently introduced *bag distance* metric [40], which is a good approximation of the previously mentioned edit distance metrics, is also provided. It is calculated in linear time as $bag_{dist}(s,t) = \max(|M(s) \backslash M(t)|, |M(t) \backslash M(s)|)$, where $M(x)$ denotes the multiset of the characters in x.

9.2.2 Jaro and Jaro-Winkler Metrics

Good results for name-matching tasks [31] have been reported using variants of the *Jaro* metric [41], which is not based on the edit-distance model. It considers the number and the order of the common characters between two strings. Given two strings $s = a_1 \dots a_K$ and $t = b_1 \dots b_L$, we say that a_i in s is *common* with t if there is a $b_j = a_i$ in t such that $i - R \leq j \leq i + R$, where $R = \lfloor \max(|s|, |t|)/2 \rfloor - 1$. Further, let $s' = a'_1 \dots a'_K$ be the characters in s which are common with t (with preserved order of

[5] It is usually straightforward to convert a string similarity metric into a string distance metric

appearance in s) and let $t' = b'_1 \dots b'_L$ be defined analogously. A *transposition* for s' and t' is defined as the position i such that $a'_i \neq b'_i$. Let us denote the number of transposition for s' and t' as $T_{s',t'}$. The *Jaro* similarity is then calculated as:

$$J(s,t) = \frac{1}{3} \cdot (\frac{|s'|}{|s|} + \frac{|t'|}{|t|} + \frac{|s'| - \lfloor T_{s',t'}/2 \rfloor}{|s'|})$$

A *Winkler* variant thereof ($JaroWinkler$) boosts the *Jaro* similarity for strings with agreeing initial characters. It is calculated as:

$$JW(s,t) = J(s,t) + \delta \cdot boost_p(s,t) \cdot (1 - J(s,t))$$

,where δ denotes the common prefix adjustment factor (default: 0.1) and $boost_p(s,t) = \min(|lcp(s,t)|, p)$. Here $lcp(s,t)$ denotes the longest common prefix between s and t, whereas p stands for the maximum length of a prefix to consider when computing longest common prefix for s and t, i.e., common prefixes, which are longer than $boost_p(s,t)$ are not boosted more than a common prefix, whose length is exactly $boost_p(s,t)$. For multi-token strings we extended $boost_p$ to $boost^*_p$. Let $s = s_1 \dots s_K$ and $t = t_1 \dots t_L$, where s_i (t_i) represent i-th token of s and t respectively, and let without loss of generality $L \leq K$. $boost^*_p$ is calculated as:

$$boost^*_p(s,t) = \frac{1}{L} \cdot \sum_{i=1}^{L-1} boost_p(s_i, t_i) + \frac{boost_p(s_L, t_L..t_K)}{L}$$

The metric which uses $boost^*_p$ is denoted in the library as $JaroWinkler2$. The time complexity of '*Jaro*' metrics is $O(|s| \cdot |t|)$.

9.2.3 Character N-gram based Metrics

The *q-gram* metric [42] is based on the intuition that two strings are similar if they share a large number of character-level q-grams. Let $G_q(s)$ denote the multiset of all q-grams of a string s obtained by sliding a window of length q over the characters of s. Since q-grams at the beginning and the end of the string can have fewer than q characters, the strings are extended by adding $q-1$ unique initial and trailing characters to a string. The q-gram metric is calculated as:

$$q{-}grams(s,t) = \frac{|G_q(s) \cap G_q(t)|}{\max(|G_q(s)|, |G_q(t)|)}$$

An extension to this metric is to add positional information, and to match only common q-grams that occur within a maximum distance from each other (*positional q-grams*) [43]. Further, [44] introduced *skip-gram* metric. It is based on the idea that in addition to forming bigrams of adjacent characters, bigrams that skip characters are considered. *Gram classes* are defined that specify what kind of skip-grams are created, e.g. $\{0,1\}$ class means that normal bigrams (0 class) are formed, and bigrams that skip one character (1-class). The q-gram type metrics can be computed in $O(\max\{|s|, |t|\})$.

9.2.4 Common Prefix/Substring-based Metrics

Considering the declension paradigm of many inflective languages a basic and time efficient metric based on the longest common prefix information is provided. It is calculated as follows.

$$CP_\delta(s,t) = \frac{(|lcp(s,t)|)^2}{|s| \cdot |t|}$$

It was mainly designed for matching single-token strings. For coping with multi-token strings a similar metric is provided, namely, *longest common substrings* distance (LCS), which recursively finds and removes the longest common substring in the two strings compared. Let $lcs(s,t)$ denote the 'first'[6] longest common substring for s and t and let s_{-p} denote a string obtained via removing from s the first occurrence of p in s. The LCS metric is calculated as:

$$LCS(s,t) = \begin{cases} 0 \text{ if } |lcs(s,t)| \leq \phi \\ |lcs(s,t)| + LCS(s_{-lcs(s,t)}, t_{-lcs(s,t)}) \end{cases}$$

[6] Reading the strings from left to right.

The value of ϕ is usually set to 2 or 3. The time complexity of LCS is $O(|s| \cdot |t|)$. We extended LCS by additional weighting of the $|lcs(s,t)|$. The main idea is to penalize longest common substrings which do not match the beginning of a token in at least one of the compared strings. Let α be the maximum number of non-whitespace characters, which precede the first occurrence of $lcs(s,t)$ in s or t. Then, $lcs(s,t)$ is assigned the weight:

$$w_{lcs(s,t)} = \frac{|lcs(s,t)| + \alpha - \max(\alpha, p)}{|lcs(s,t)| + \alpha}$$

where p has been experimentally set to 4 (default). We denote this variant of LCS as *Weighted Longest Common Substrings* ($WLCS$).

9.2.5 Recursive Metrics

Finally, for matching multi-token strings the recursive schema, known also as *Monge-Elkan* distance [45] is provided in the library. Let us assume that the strings s and t are broken into substrings (tokens), i.e., $s = s_1 \dots s_K$ and $t = t_1 \dots t_L$. The intuition behind *Monge-Elkan* measure is the assumption that s_i in s corresponds to a t_j with which it has highest similarity. The similarity between s and t equals the mean of these maximum scores. Formally, the *Monge-Elkan* metric is defined as follows, where sim denotes some secondary similarity function.

$$Monge{-}Elkan(s,t) = \frac{1}{K} \cdot \sum_{i=1}^{K} \max_{j=1\dots L} sim(s_i, t_j)$$

Inspired by the multi-token variants of the *JaroWinkler* metric presented in [33] two additional metrics are provided. They are similar in spirit to the *Monge-Elkan* metric. The first one, *Sorted-Tokens* is computed in two steps: (a) firstly the tokens constituting the full strings are sorted alphabetically, and (b) an arbitrary metric is applied to compute the similarity of the 'sorted' strings. The second metric, *Permuted-Tokens* compares all possible permutations of tokens constituting the full strings and returns the maximum calculated similarity value.

Further details and references to papers on string distance metrics can be found in [33, 34].

9.3 Tools

In order to experiment and play with the string distance metrics, one can use the following script, where `<configuration file>` refers to the specific configuration file for the string metric being used.

```
applyDistanceMetric   <configuration file>
```

The configuration file (properties file) specifies the following properties:

- `METRIC`: the name of the metric to be used (corresponds to the name of the class implementing the metric, see JAVA API documentation)
- `METRICCONF` a configuration of the metric represented as a string of the format:

  ```
  ATTRIBUTE-1:VALUE-1 | ATTRIBUTE-2:VALUE-2 ..... | ATTRIBUTE-K:VALUE-K
  ```

- `INPUTFILE1`: a path to the first input file containing a list of strings to compare (each line contains just a single string)
- `INPUTFILE2`: a path to the second input file containing a list of strings to compare with the strings contained in the first file
- `ENCODING`: file encoding (characterset)
- `OUTPUTFILE`: a file to write the results to

- `MINVAL`: the upper bound for the distance between two strings, i.e., if the distance between two strings is more than the value specified by this parameter, then they will not be included in the output file

Please note that the description of all metric-specific settings and parameters can be found in the JAVA API documentation. Therefore, they are not described here.

An example of a configuration file for using *Smith-Waterman with affine gaps* metric is given below.

```
METRIC=SmithWatermanWithAffineGaps
METRICCONF=WINDOWSIZE:100 | GAPCOSTFUNCTION:DefaultAffineGapCostSmithWaterman
                          | SUBSTCOSTFUNCTION:DefaultSubstCostSmithWatermanAffineGaps
INPUTFILE1=sample_person_names_1.txt
INPUTFILE2=sample_person_names_2.txt
ENCODING=UTF-8
OUTPUTFILE=myResult.txt
MINVAL=0.3
```

In this particular example, there are three parameters for the metric being used, namely, window size, gap cost function and substitution cost function. The gap and cost function can be provided by the user via implementing an appropriate programming interface. Please refer to the description of *Smith-Waterman with Affine gaps* in the JAVA API documentation and standard literature cited previously for more details concerning various parameters for different string distance metrics.

The following example demonstrates a piece of code, which uses a string distance metric. It corresponds to the previous example with *Smith-Waterman with Affine gaps* metric.

```
// create an instance of a string distance metric
// IMPORTANT: the parameter to the method getInstance() is the name of the class,
//            which implements the metric
AbstractDistanceMetric myMetric
                        = AbstractDistanceMetric.getInstance("SmithWatermanWithAffineGaps");

// configure settings
Properties settings = new Poperties();
settings.put("WINDOWSIZE":"100");
settings.put("GAPCOSTFUNCTION",:"DefaultAffineGapCostSmithWaterman");
settings.put("SUBSTCOSTFUNCTION","DefaultSubstCostSmithWatermanAffineGaps");
myMetric.setProperties(settings);

// one can check the configuration of the metric as follows
System.out.println(myMetric.getName() + " : " + myMetric.getConfiguration()

// apply the metric
float distance = myMetric.getDistance("Dawid", "David");
```

For the sake of completeness a list of all currently available string distance metrics given by the corresponding class names is listed below.

```
BagDistance                NeedlemanWunsch
CommonPrefixSq             PermutedTokensDistance
CommonPrefixSq2            PositionalQgrams
Jaro                       MongeElkan
JaroWinkler                Qgrams
JaroWinkler2               SkipGrams
Levenshtein                SmithWatermanWithAffineGaps
LongestCommonSubstrings    SortedTokensDistance
SmithWaterman              WeightedLongestCommonSubstrings
```

10 Summary and Future Work

This report introduced CORLEONE - a set of general-purpose Unicode-aware components for performing basic linguistic operations, which encompasses basic tokenization, fine-grained tokenization, morphological analysis, gazetteer look-up, and sentence splitting. They can robustly and efficiently process vast

amount of textual data, which is essential in the context of the EMM system. In particular MB-sized documents can be processed within seconds or in some cases in a fraction of a second. Further, the advanced compression techniques we applied allow for storing the compiled resources in a memory-efficient way. Quick component initialization time was obtained via replacing the standard JAVA serialization mechanism, known to be enormously slow, with our own procedure for fulfilling this task. Additionally, CORLEONE comes equipped with a comprehensive library of string distance metrics, which are useful for performing name matching tasks.

The presented tools are very flexible and can be easily adapted to new domains and languages. Further, they do not rely on any third-party software and provide transparent interfaces so that their integration can be done in a straightforward manner. In particular, CORLEONE components are used as basic linguistic processing resources in ExPRESS, a pattern matching engine based on regular expressions over feature structures [1]. Furthermore, CORLEONE is a subpart of the real-time news event extraction system described in [2]. Some of CORLEONE components are used in other applications developed in the JRC.

Future activities will focus on developing some more core NLP modules, e.g., partial reference matcher, partial part-of-speech filtering and eventually a chunking component. Although the current version of the tool is time and space efficient, there is still a lot of space for improvement. In particular, implementing a space-efficient memory model for storing gazetteers with large amount of numerical data could be addressed.

11 Acknowledgments

The work presented in this report was supported by the Europe Media Monitoring Project (EMM) carried out by the Web Mining and Intelligence Action in the Joint Research Centre of the European Commission.

The author is greatly indebted to Delilah Al-Khudhairy, Jonathan Brett Crawley, Camelia Ignat, Bruno Pouliquen, Ralf Steinberger, Hristo Tanev and Vanni Zavarella for some comments, inspiring discussions and some ideas on how to improve CORLEONE and this report. The author would also like to thank Martin Atkinson for the CORLEONE logo.

Some of the raw linguistic resources, e.g. MULTEXT resources etc., have been extended and fine-tuned by Ralf Steinberger and Vanni Zavarella.

References

[1] Piskorski, J.: ExPRESS – Extraction Pattern Recognition Engine and Specification Suite. In: Proceedings of the International Workshop Finite-State Methods and Natural language Processing 2007 (FSMNLP'2007), Potsdam, Germany. (2007)

[2] Tanev, H., Piskorski, J., Atkinson, M.: Real-Time News Event Extraction for Global Crisis Monitoring. In: Proceedings of the 13th International Conference on Applications of Natural Language to Information Systems (NLDB 2008), London, UK, 24–27 June, 2008. Lecture Notes in Computer Science Vol 5039, Springer Verlag Berlin Heidelberg. (2008) 207–218

[3] Best, C., Pouliquen, B., Steinberger, R., Van der Goot, E., Blackler, K., Fuart, F., Oellinger, T., Ignat, C.: Towards Automatic Event Tracking. In: Proceedings of IEEE International Conference on Intelligence and Security Informatics – ISI 2006, San Diego, USA. (2006)

[4] Douglas, A.: Introduction to Information Extraction. AI Communications **12**(3) (1999) 161–172

[5] Emmanuel Roche and Yves Schabes, ed.: Finite-state language processing. A Bradford Book, MIT Press, Cambridge, MA (1997)

[6] Piskorski, J., Neumann, G.: An Intelligent Text Extraction and Navigation System. In: Proceedings of the 6th International Conference on Computer-Assisted Information Retrieval (RIAO-2000), Paris. (2000)

[7] Neumann, G., Piskorski, J.: A Shallow Text Processing Core Engine. Journal of Computational Intelligence **18**(3) (2002)

[8] Drożdżyński, W., Krieger, H.U., Piskorski, J., Schäfer, U., Xu, F.: Shallow Processing with Unification and Typed Feature Structures — Foundations and Applications. Künstliche Intelligenz **2004(1)** (2004) 17–23

[9] Piskorski, J.: SproUT: An Integrated Set of Tools for Shallow Text Processing. In: Proceedings of Business information Systems 2004, April 2004, Poznan, Poland (2004)

[10] Krieger, H.U., Drożdżyński, W., Piskorski, J., Schäfer, U., Xu, F.: A Bag of Useful Techniques for Unification-Based Finite-State Transducers. In: Proceedings of of 7th KONVENS Conference, Vienna, Austria. (2004)

[11] Piskorski, J.: Finite-State Machine Toolkit. Technical Report RR-02-04, DFKI, Saarbruecken (2002)

[12] Piskorski, J., Drożdżyński, W., Xu, F., Scherf, O.: A Flexible XML-based Regular Compiler for Creation and Converting Linguistic Resources. In: Proceedings of the 3rd International Conference on Language Resources an Evaluation (LREC'02), Las Palmas, Canary Islands, Spain (2002)

[13] Piskorski, J.: On Compact Storage Models for Gazetteers. In: Proceedings of the 5th International Workshop on Finite-State Methods and Natural Language Processing, Helisnki, Finland, Springer, Lecture Notes in Artificial Intelligence (2005)

[14] Daciuk, J., Piskorski, J.: Gazetteer Compression Technique Based on Substructure Recognition. In: Proceedings of Intelligent Information Systems 2006 - New Trends in Intelligent Information Processing and Web Mining, Springer Verlag series "Advances in Soft Computing" (2006)

[15] Greffenstette, G., Tapanainen, P.: What is a word, what is a sentence? Problems of tokenization. In: Proceedings of the third International Conference on Computational Lexicography – Complex 94, Budapest, Hungary. (1994) 79–87

[16] Grover, C., Matheson, C., Mikheev, A., Moens, M.: LT TTT - A Flexible Tokenisation Tool. In: Proceedings of the Second Language Resources and Evaluation Conference, Athens, Greece. (2000)

[17] Gillam, R.: Text Boundary Analysis in Java.
http://www.ibm.com/java/education/boundaries/boundaries.html (1999)

[18] Levine, J., Mason, T., Brown, D.: Lex & Yacc. O'Reilly & Associates Inc. (1992)

[19] Skut, W., Piskorski, J., Daciuk, J.: Finite-State Machines - Foundations and Applications to Text Processing and Pattern Recognition. ESSLI 2005, Heriot-Watt University, Edinburgh, Scotland (2005)

[20] Tarjan, R.E., Yao, A.C.C.: Storing a Sparse Table. Commun. ACM **22**(11) (1979) 606–611

[21] Kiraz, G.: Compressed Storage of Sparse Finite-State Transducers. In: Proceedings of WIA 1999, Potsdam, Germany (1999) 109–121

[22] Mohri, M., Pereira, F., Riley, M.: A Rational Design for a Weighted Finite-State Transducer Library. In: Revised Papers from the Second International Workshop on Implementing Automata table of contents, Lecture Notes In Computer Science; Vol. 1436. (1997) 144–158

[23] Kowaltowski, T., Lucchesi, C.: Applications of Finite Automata Representing Large Vocabularies. Technical Report DCC-01/92, University of Campinas, Brazil (1992)

[24] Kowaltowski, T., Lucchesi, C., Stolfi, J.: Finite Automata and Efficient Lexicon Implementation. Technical Report IC-98-02, University of Campinas, Brazil (1998)

[25] Ide, N., Tufis, D., Erjavec, T.: Development and Assesment of Common Lexical Specifications for Six Central and Eastern European Languages. In: Proceedings of the first International Conference on Language Resources and Evaluation, LREC 1998, Granada, Spain. (1998) 233–240

[26] Daciuk, J.: Incremental Construction of Finite-State Automata and Transducers. PhD Thesis. Technical University Gdańsk. (1998)

[27] Daciuk, J., Mihov, S., Watson, B., Watson, R.: Incremental Construction of Minimal Acyclic Finite State Automata. Computational Linguistics **26**(1) (2000) 3–16

[28] Erjavec, T.: MULTEXT - East Morphosyntactic Specifications (2004)

[29] Ciura, M., Deorowicz, S.: How to Squeeze a Lexicon. Software - Practice and Experience **31(11)** (2001) 1077–1090

[30] Daciuk, J.: Experiments with Automata Compression. In: Proceedings of CIAA - Implementation and Application of Automata, London, Ontario, Canada (2000) 105–112

[31] Cohen, W., Ravikumar, P., Fienberg, S.: A Comparison of String Distance Metrics for Name-Matching Tasks. In: Proceedings of IJCAI-03 Workshop on Information Integration on the Web (IIWeb-03), 2003, Acapulco, Mexico, Acapulco, Mexico (2003) 73–78

[32] Cohen, E., Ravikumar, P., Fienberg, S.: A Comparison of String Metrics for Matching Names and Records. KDD Workshop on Data Cleaning and Object Consolidation (2003)

[33] Christen, P.: A Comparison of Personal Name Matching: Techniques and Practical Issues. Technical report, TR-CS-06-02, Computer Science Laboratory, The Australian National University, Canberra, Australia (2006)

[34] Elmagaramid, A., Ipeirotis, P., Verykios, V.: Duplicate Record Detection: A Survey. IEEE Transactions on Knowledge and Data Engineering **19**(1) (2007)

[35] Piskorski, J., Sydow, M., Kupść, A.: Lemmatization of Polish Person Names. In: Proceedings of the ACL Workshop on Balto-Slavonic Natural Language Processing 2007 - Special Theme: Information Extraction and Enabling Technologies (BSNLP'2007). Held at ACL'2007, Prague, Czech Republic, 2007, ACL Press (2007)

[36] Piskorski, J., Wieloch, K., Pikula, M., Sydow, M.: Towards Person Name Matching for Inflective Languages. In: Proceedings of the WWW 2008 workshop on NLP Challenges in the Information Explosion Era (NLPIX 2008), ACM (2008)

[37] Levenshtein, V.: Binary Codes for Correcting Deletions, Insertions, and Reversals. Doklady Akademii Nauk SSSR **163**(4) (1965) 845–848

[38] Needleman, S., Wunsch, C.: A General Method Applicable to Search for Similarities in the Amino Acid Sequence of Two Proteins. Journal of Molecular Biology **48**(3) (1970) 443–453

[39] Smith, T., Waterman, M.: Identification of Common Molecular Subsequences. Journal of Molecular Biology **147** (1981) 195–197

[40] Bartolini, I., Ciacca, P., Patella, M.: String Matching with Metric Trees Using an Approximate Distance. In: Proceedings of SPIRE, LNCS 2476, Lissbon, Portugal. (2002) 271–283

[41] Winkler, W.: The State of Record Linkage and Current Research Problems. Technical report, Statistical Research Division, U.S. Bureau of the Census, Washington, DC (1999)

[42] Ukkonen, E.: Approximate String Matching with q-grams and Maximal Matches. Theoretical Computer Science **92**(1) (1992) 191–211

[43] Gravano, L., Ipeirotis, P., Jagadish, H., Koudas, N. Muthukrishnan, S., Pietarinen, L., Srivastava, D.: Using q-grams in a DBMS for Approximate String Processing. IEEE Data Engineering Bulletin **24**(4) (2001) 28–34

[44] Keskustalo, H., Pirkola, A., Visala, K., Leppanen, E., Jarvelin, K.: Non-adjacent bigrams Improve Matching of Cross-lingual Spelling Variants. In: Proceedings of SPIRE, LNCS 22857, Manaus, Brazil. (2003) 252–265

[45] Monge, A., Elkan, C.: The Field Matching Problem: Algorithms and Applications. In: Proceedings of Knowledge Discovery and Data Mining 1996. (1996) 267–270

European Commission

EUR 23393 EN – Joint Research Centre – Institute for the Protection and Security of the Citizen
Title: CORLEONE (**Cor**e **L**inguistic **E**ntity **On**line **E**xtraction)
Author(s): Jakub Piskorski
Luxembourg: Office for Official Publications of the European Communities
2008 – 32 pages – 21 x 29,7 cm
EUR – Scientific and Technical Research series – ISSN 1018-5593

Abstract
This report presents CORLEONE (**Cor**e **L**inguistic **E**ntity **On**line **E**xtraction) - a pool of loosely coupled general-purpose basic lightweight linguistic processing resources, which can be independently used to identify core linguistic entities and their features in free texts. Currently, CORLEONE consists of five processing resources: (a) a basic tokenizer, (b) a tokenizer which performs fine-grained token classification, (c) a component for performing morphological analysis, and (d) a memory-efficient database-like dictionary look-up component, and (e) sentence splitter. Linguistic resources for several languages are provided. Additionally, CORLEONE includes a comprehensive library of string distance metrics relevant for the task of name variant matching. CORLEONE has been developed in the Java programming language and heavily deploys state-of-the-art finite-state techniques.

Noteworthy, CORLEONE components are used as basic linguistic processing resources in ExPRESS, a pattern matching engine based on regular expressions over feature structures and in the real-time news event extraction system developed in the JRC by the Web Mining and Intelligence Action of IPSC.

This report constitutes an end-user guide for COLREONE and provides some scientifically interesting details of how it was implemented.

The mission of the JRC is to provide customer-driven scientific and technical support for the conception, development, implementation and monitoring of EU policies. As a service of the European Commission, the JRC functions as a reference centre of science and technology for the Union. Close to the policy-making process, it serves the common interest of the Member States, while being independent of special interests, whether private or national.