# jSRE - java Simple Relation Extraction

## User's Guide

### Table of Contents

### Introduction

jSRE is an open source Java tool for *Relation Extraction* . It is based on a supervised machine learning approach which is applicable even when (deep) linguistic processing is not available or reliable. In particular, jSRE uses a combination of kernel functions to integrate two different information sources: (i) the whole sentence where the relation appears, and (ii) the local contexts around the interacting entities. jSRE requires only a shallow linguistic processing, such as tokenization, sentence splitting, Part-of-Speech (PoS) tagging and lemmatization. A detailed description of *Simple Relation Extraction* is given in [1].

### License

jSRE is released as free software with full source code, provided under the terms of the Apache License, Version 2.0.

### System Requirements

The jSRE software is available on all platforms supporting Java 2.

### Installation

1. Make sure you have installed Sun's Java 2 Environment. The full version is available at `http://java.sun.com/j2se`. If you are limited by disk space or bandwidth, install the smaller, run-time only version at `http://java.sun.com/j2se`.
2. Download the jar file of the installer here. Copy the file into the directory where you want to install the program and then run:

   jar -xvf jsre-1.1.jar

## Dependencies

jSRE uses elements of the Java 2 API such as collections, and therefore building requires the Java 2 Standard Edition SDK (Software Development Kit). To run jSRE, the Java 2 Standard Edition RTE (Run Time Environment) is required (or you can use the SDK, of course).

jSRE is also dependent upon a few packages for general functionality. They are included in the lib directory for convenience, but the default build target does not include them. If you use the default build target, you must add the dependencies to your classpath.

- Jakarta Commons - required.
- Apache Log4j - required.
- LIBSVM - required.

Using a C Shell run:

```
setenv CLASSPATH jsre.jar
setenv CLASSPATH ${CLASSPATH}:lib/libsvm-2.8.jar
setenv CLASSPATH ${CLASSPATH}:lib/log4j-1.2.8.jar
setenv CLASSPATH ${CLASSPATH}:lib/commons-digester.jar
setenv CLASSPATH ${CLASSPATH}:lib/commons-beanutils.jar
setenv CLASSPATH ${CLASSPATH}:lib/commons-logging.jar
setenv CLASSPATH ${CLASSPATH}:lib/commons-collections.jar
```

## Input Format

Example files are ASCII text files and represent the set of positive and negative examples for a specific binary relation. Consider the *work_for* relation between a person and the organization for which he/she works.

```
"Also being considered are Judge \emph{<PER>}
Ralph K. Winter\emph{</PER>}
 of the
\emph{<ORG>}
2nd U.S. Circuit Court of Appeals\emph{</ORG>}
 in \emph{<Loc>}
New YorkCity\emph{</Loc>}

and Judge \emph{<PER>}
Kenneth Starr\emph{</PER>}
 of the
\emph{<ORG>}
U.S. Circuit Court of Appeals\emph{</ORG>}
 for the
\emph{<LOC>}
District of Columbia\emph{</LOC>}
."
```

In the above sentence we have 2 *PER* entities and 2 *ORG* entities, 4 potential *work_for* relations.

2 are positive examples for the *work_for* relation:

```
... \emph{<PER>}
Ralph K. Winter\emph{</PER>}
 ... \emph{<ORG>}
2nd U.S. Circuit Court of Appeals\emph{</ORG>}
 ...
... \emph{<PER>}
Kenneth Starr\emph{</PER>}
 ... \emph{<ORG>}
U.S. Circuit Court of Appeals\emph{</ORG>}
 ...
```

while 2 are negative examples:

```
... \emph{<PER>}
Ralph K. Winter\emph{</PER>}
 ... \emph{<ORG>}
U.S. Circuit Court of Appeals\emph{</ORG>}
 ...
... \emph{<PER>}
Kenneth Starr\emph{</PER>}
 ... \emph{<ORG>}
2nd U.S. Circuit Court of Appeals\emph{</ORG>}
 ...
```

Each example is essentially a pair of candidate entities possibly relating according to the relation of interest. In the jSRE example file each example is basically represented as an instance of the original sentence with the two candidates properly annotated. Each example has to be placed on a single line and the example format line is:

example  label\tid\tbody\n

| | |
|---|---|
| label | example label (e.g. 0 negative 1 positive) |
| id | unique example identifier, (e.g. a sentence identifier followed by an incremental identifier for the example) |
| body | it is the instance of the original sentence |

Where body is encoded according to the following format:

body  [tokenid&&token&&lemma&&POS&&entity_type&&entity_label\s]+

The body is a sequence of whitespace separated tokens. Each token is represented with 6 attributes separated by the special character sequence "&&". A token is any sequence of adjacent characters in the sentence or an entity. An entity must be represented as a single token where all whitespaces are substituted by the "_" character (e.g. "Ralph_K._Winter")

| | |
|---|---|
| tokenid | |
| token | |
| lemma | |
| POS | |
| entity_type | possible t |
| entity_label | A—T—O this attribute is to label the candidate pair. Each example in the jSRE file has two entities labelle |

The example for the “Ralph K. Winter” “2nd U.S. Circuit Court of Appeals” pair is:

```
1  52-6    0&&Also&&Also&&RB&&O&&O 1&&being&&being&&VBG&&O&&O 2&&considered&&considered&&VBN&&O&&O 3&&a
```

jSRE will consider the examples as examples for a binary classification problem.

In order to reduce the number of negative examples in the case of relation between entities of the same type (e.g. kill between 2 people) jSRE examples should be generated not for each pair of possibly relating entities but for each combination of possibly relating entities.

For example in the following sentence there are 3 people entities and the possible relating pairs are 6:

```
"Ides of March, 44 B.C., \emph{<PER>}
Roman Emperor Julius Caesar\emph{<PER>}
 was
assassinated by a group of nobles that included \emph{<PER>}
Brutus\emph{</PER>}

and \emph{<PER>}
Cassius\emph{</PER>}
."


0 ... \emph{<PER>}
Roman Emperor Julius Caesar\emph{<PER>}
 ... \emph{<PER>}
Brutus\emph{</PER>}
 ...
1 ... \emph{<PER>}
Brutus\emph{</PER>}
 ... \emph{<PER>}
Roman Emperor Julius Caesar\emph{<PER>}
 ...
0 ... \emph{<PER>}
Roman Emperor Julius Caesar\emph{<PER>}
 \emph{<PER>}
Cassius\emph{</PER>}
 ...
1 ... \emph{<PER>}
Cassius\emph{</PER>}
 ... \emph{<PER>}
Roman Emperor Julius Caesar\emph{<PER>}
 ...
0 ... \emph{<PER>}
Cassius\emph{</PER>}
 ... \emph{<PER>}
Brutus\emph{</PER>}

0 ... \emph{<PER>}
Brutus\emph{</PER>}
```

```
 ... \emph{<PER>}
Cassius\emph{</PER>}
```

Examples for jSRE can be generated just for each combination of entities representing the direction of the relation through different positive labels. In this case the entity_label has to be "T" for both the candidates: if the relation is between the first and the second candidate (according to the token id order that is the sentence order) the example will be labelled 1, otherwise it will be labelled 2. If there is no relation between the 2 candidates the example will be labelled 0. In the example above we obtain only 3 examples (1 is negative).

```
2 ... \emph{<PER>}
Roman Emperor Julius Caesar\emph{<PER>}
 ... \emph{<PER>}
Brutus\emph{</PER>}
 ...
2 ... \emph{<PER>}
Roman Emperor Julius Caesar\emph{<PER>}
 ... \emph{<PER>}
Cassius\emph{</PER>}
 ...
0 ... \emph{<PER>}
Brutus\emph{</PER>}
 ... \emph{<PER>}
Cassius\emph{</PER>}
 ...
```

jSRE will consider this as the example set for a multiclassification problem.

## Running jSRE

This section explains how to use the jSRE software. jSRE implements the class of shallow linguistic kernels described in [1].

jSRE consists of a training module (Train) and a classification module (Predict). The classification module can be used to apply the learned model to new examples. See also the examples below for how to use Train and Predict.

Train is called with the following parameters:

```
java -mx128M org.itc.irst.tcc.sre.Train [options] example-file model-file
```

Arguments:

| | |
|---|---|
| example-file | file with training data |
| model-file | file in which to store resulting model |

Options:

| | |
|---|---|
| -h | this help |
| -k string | set type of kernel function (default SL): |
| | LC: Local Context Kernel |
| | GC: Global Context Kernel |
| | SL: Shallow Linguistic Context Kernel |
| -m int | set cache memory size in MB (default 128) |
| -n [1..] | set the parameter n-gram of kernels SL and GC (default 3) |
| -w [1..] | set the window size of kernel LC (default 2) |
| -c [0..] | set the trade-off between training error and margin (default 1/[avg. x*x']) |

The input file example-file contains the training examples in the format described in . The result of Train is the model which is learned from the training data in example-file. The model is written to model-file. To make predictions on test examples, Predict reads this file.

Predict is called with the following parameters:

```
java org.itc.irst.tcc.sre.Predict [options] test-file model-file output-file
```

Arguments:

| | |
|---|---|
| example-file | file with test data |
| model-file | file from which to load the learned model |
| output-file | file in which to store resulting predictions |

Options:

| | |
|---|---|
| -h | this help |

The test examples in example-file are given in the same format as the training examples (possibly with -1 as class label, indicating unknown). For all test examples in example-file the predicted values are written to output-file. There is one line per test example in output-file containing the value of the classification on that example.

**Case of Study: The relation located_in**

Suppose to have in located_in.train and located_in.test the training and test set, respectively, tagged in SRE format for the relation *located_in* . To train a model for *located_in* , run:

```
java -mx256M org.itc.irst.tcc.sre.Train -m 256 -k SL -c 1 examples/located_in.train examples/located_in
```

The standard output is:

```
train a relation extraction model
read the example set
find argument types
arg1 type: LOC
arg2 type: LOC
create feature index
embed the training set
save the embedded training set
save feature index
save parameters
run svm train
.*
```

```
optimization finished, #iter = 1628
obj = -58.42881586324897, rho = 0.8194511083494147
nSV = 439, nBSV = 10
.*
optimization finished, #iter = 354
obj = -13.875607571278666, rho = 0.0232461933948966
nSV = 146, nBSV = 0
*
optimization finished, #iter = 857
obj = -32.435195153048916, rho = -1.2010373459490367
nSV = 306, nBSV = 2
Total nSV = 658
```

To predict *located_in* , run:

```
java org.itc.irst.tcc.sre.Predict examples/located_in.test examples/located_in.model examples/located_i
```

The standard output is:

```
predict relations
read parameters
read the example set
read data set
find argument types
arg1 type: LOC
arg2 type: LOC
read feature index
embed the test set
save the embedded test set
run svm predict
Accuracy = 90.98039215686275% (232/255) (classification)
Mean squared error = 0.14901960784313725 (regression)
Squared correlation coefficient = 0.5535585550902604 (regression)
tp      fp      fn      total   prec    recall  F1
65      10      13      255     0.867   0.833   0.850
```

The output files located_in.output contains the predictions.

To see the list of extracted located_in, run:

```
java org.itc.irst.tcc.sre.RelationExtractor examples/located_in.test examples/located_in.output
```

A fragment of the output is:

```
1 relations found in sentence 2456
0 Whitefish ===> Montana  (1)

1 relations found in sentence 28
```

```
1 Naples ===> Campania  (1)

1 relations found in sentence 1359
2 Riga ===> Latvia  (1)

1 relations found in sentence 130
3 Hot_Springs_National_Park ===> Ark.  (1)

1 relations found in sentence 2412
4 Addis_Ababa ===> Ethiopia.  (1)

...

2 relations found in sentence 1486
46 Port_Arther ===> Texas.  (1)
47 Galveston ===> Texas.  (1)

1 relations found in sentence 5921
48 Zambia <=== Kafue_River (2)

1 relations found in sentence 5169
49 New_York <=== Dakota (2)

...
```

## History

- v 1.0

## Bibliography

[1] Claudio Giuliano, Alberto Lavelli, Lorenza Romano. *Exploiting Shallow Linguistic Information for Relation Extraction from Biomedical Literature* . In *Proceedings of the 11th Conference of the European Chapter of the Association for Computational Linguistics (EACL 2006)* , Trento, Italy, 3-7 April 2006. [PDF]

[2] Claudio Giuliano, Alberto Lavelli, Daniele Pighin and Lorenza Romano. *FBK-IRST: Kernel Methods for Semantic Relation Extraction* . In *Proceedings of the 4th Interational Workshop on Semantic Evaluations (SemEval-2007)* , Prague, 23-24 June 2007.