SUPWSD: A Flexible Toolkit for Supervised Word Sense Disambiguation

Simone Papandrea, Alessandro Raganato and **Claudio Delli Bovi**
Department of Computer Science
Sapienza University of Rome
papandrea.simone@gmail.com
{raganato,dellibovi}@di.uniroma1.it

Abstract

In this demonstration we present SUPWSD, a Java API for supervised Word Sense Disambiguation (WSD). This toolkit includes the implementation of a state-of-the-art supervised WSD system, together with a Natural Language Processing pipeline for preprocessing and feature extraction. Our aim is to provide an easy-to-use tool for the research community, designed to be modular, fast and scalable for training and testing on large datasets. The source code of SUPWSD is available at http://github.com/SI3P/SupWSD.

1 Introduction

Word Sense Disambiguation (Navigli, 2009, WSD), is one of the long-standing challenges of Natural Language Understanding. Given a word in context and a pre-specified sense inventory, the task of WSD is to determine the intended meaning of that word depending on the context. Several WSD approaches have been proposed over the years and extensively studied by the research community, ranging from knowledge-based systems to semi-supervised and fully supervised models (Agirre et al., 2014; Moro et al., 2014; Taghipour and Ng, 2015b; Iacobacci et al., 2016). Nowadays a new line of research is emerging, and WSD is gradually shifting from a purely monolingual (i.e. English) setup to a wider multilingual setting (Navigli and Moro, 2014; Moro and Navigli, 2015). Since scaling up to multiple languages is considerably easier for knowledge-based systems, as they do not require sense-annotated training data, various efforts have been made towards the automatic construction of high-quality sense-annotated corpora for multiple languages (Otegi et al., 2016; Delli Bovi et al., 2017), aimed at overcoming the so-called *knowledge acquisition bottleneck* of supervised models (Pilehvar and Navigli, 2014). These efforts include the use of Wikipedia, which can be considered a full-fledged, manually sense-annotated resource for numerous languages, and hence exploited as training data (Dandala et al., 2013).

Beside the automatic harvesting of sense-annotated data for different languages, a variety of multilingual preprocessing pipelines has also been developed across the years (Padr and Stanilovsky, 2012; Agerri et al., 2014; Manning et al., 2014). To date, however, very few attempts have been made to integrate these data and tools with a supervised WSD framework; as a result, multilingual WSD has been almost exclusively tackled with knowledge-based systems, despite the fact that supervised models have been proved to consistently outperform knowledge-based ones in all standard benchmarks (Raganato et al., 2017). As regards supervised WSD, It Makes Sense (Zhong and Ng, 2010, IMS) is indeed the de-facto state-of-the-art system used for comparison in WSD, but it is available only for English, with the last major update dating back to 2010.

The publicly available implementation of IMS also suffers from two crucial drawbacks: (i) the design of the software makes the current code difficult to extend (e.g. with classes taking as input more than 15 parameters); (ii) the implementation is not optimized for larger datasets, being rather time- and resource-consuming. These difficulties hamper the work of contributors willing to update it, as well as the effort of researchers that would like to use it with languages other than English.

In this paper we present SUPWSD, whose objective is to overcome the aforementioned drawbacks, and facilitate the use of a supervised WSD software for both end users and researchers. SUP-

Proceedings of the 2017 EMNLP System Demonstrations, pages 103–108
Copenhagen, Denmark, September 7–11, 2017.

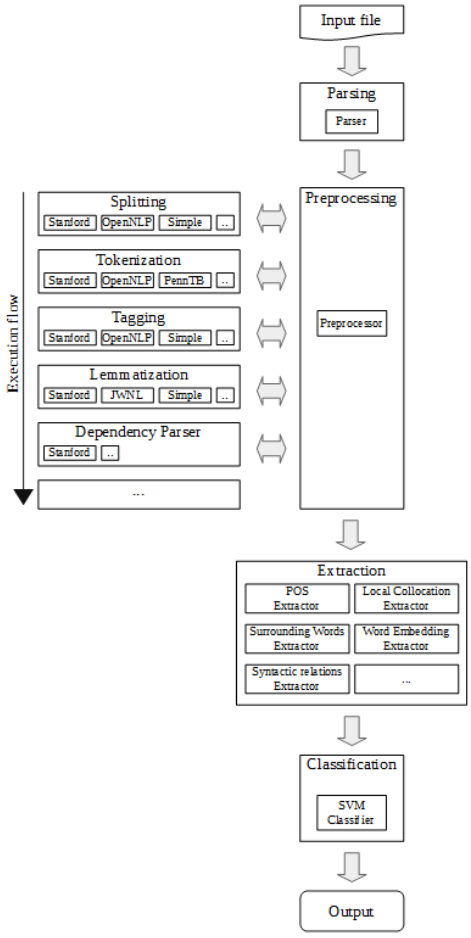

Figure 1: Architecture design of SUPWSD.

WSD is designed to be modular and highly flexible, enabling contributors to extend it with ease. Its usage is simple and immediate: it is based on a jar file with only 2 commands and 3 parameters, along with an XML configuration file for specifying customized settings. SUPWSD supports the most widely used preprocessing tools in the research community: Stanford coreNLP (Manning et al., 2014), openNLP[1], and TreeTagger (Schmid, 2013); as such, SUPWSD can directly handle all the languages supported by these tools. Finally, its architecture design relies on commonly used design patterns in Java (such as Factory and Observer among others), which make it flexible for a programmatic use and easily expandable.

2 SUPWSD: Architecture

In this section we describe the workflow of SUPWSD. Figure 1 shows the architecture design of our framework: it is composed of four main modules, common for both the training and testing phase: (i) input parsing, (ii) text preprocessing, (iii) features extraction and (iv) classification.

Input parsing. Given either a plain text or an XML file as input, SUPWSD first parses the file and extracts groups of sentences to provide them as input for the subsequent text preprocessing module. Sentence grouping is used to parallelize the preprocessing module's execution and to make it less memory-intensive. Input files are loaded in memory using a lazy procedure (i.e. the parser does not load the file entirely at once, but processes it according to the segments of interest) which enables a smoother handling of large datasets. The parser specification depends on the format of the input file via a Factory patterns, in such a way that new additional parsers can easily be implemented and seamlessly integrated in the workflow (c.f. Section 3). SUPWSD currently features 6 different parsers, targeted to the various formats of the Senseval/SemeEval WSD competition (both all-words and lexical sample), along with a parser for plain text.

Text preprocessing. The text preprocessing module runs the pre-specified preprocessing pipeline on the input text, all the way from sentence splitting to dependency parsing, and retrieves the data used by the feature extraction module to construct the features. This module consists of a five-step pipeline: sentence splitting, tokenization, part-of-speech tagging, lemmatization and dependency parsing. SUPWSD currently supports two preprocessing options: *Stanford* and *Hybrid*. Both can be switched on and off using the configuration file. The former (default choice) provides a wrapper for the Stanford NLP pipeline, and selects the default Stanford model for each component. The latter, instead, enables the user to customize their model choice for each and every preprocessing step. For instance, one possible customization is to use the openNLP models for tokenization and sentence splitting, and the Stanford models for part-of-speech tagging and lemmatization. In addition, the framework enables the user to provide an input text where preprocessing information is already included.

The communication between the input parsing and the text preprocessing modules (Figure 1) is handled by the `Analyzer`, a component that handles a fixed thread pool and outputs the feature information collected from the input text.

[1] opennlp.apache.org/

```
<?xml version="1.0" encoding="UTF-8"?>
<supWSD xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="supWSD.xsd">
    <working_directory>path</working_directory>
    <parser mns="file">lexical|senseval|semeval7|semeval13|semeval15|plain</parser>
    <preprocessing>
        <splitter model="">stanford|open_nlp|simple|none</splitter>
        <tokenizer model="">stanford|open_nlp|penn_tree_bank|simple|none</tokenizer>
        <tagger model="">stanford|open_nlp|tree_tagger|simple|none</tagger>
        <lemmatizer model="">stanford|jwnl|tree_tagger|simple|none</lemmatizer>
        <dependency_parser model="">stanford|none</dependency_parser>
    </preprocessing>
    <extraction>
        <features>
            <pos_tags cutoff="0">true|false</pos_tags>
            <local_collocations cutoff="0">true|false</local_collocations>
            <surrounding_words cutoff="0" window="1">true|false</surrounding_words>
            <word_embeddings strategy="EXP|FRA|AVG" window="10" vectors="file" vocab="file"
                                                    cache="[0...1]">true|false</word_embeddings>
            <syntactic_relations>true|false</syntactic_relations>
        </features>
    </extraction>
    <classifier>liblinear|libsvm</classifier>
    <writer>all|single|plain</writer>
    <sense_inventory dict="file">wordnet|babelnet|none</sense_inventory>
</supWSD>
```

Figure 2: The XML configuration file used by SUPWSD.

Features extraction. The feature extraction module takes as input the data extracted at preprocessing time, and constructs a set of features that will be used in the subsequent stage to train the actual SUPWSD model. As in the previous stage, the user can rely on the configuration file (Figure 2) to select which features to enable or disable. SUPWSD currently supports five standard features: (i) *part-of-speech tag* of the target word and part-of-speech tags surrounding the target word (with a left and a right window of length 3); (ii) *surrounding words*, i.e. the set of word tokens (excluding stopwords from a pre-specified list) appearing in the context of the target word; (iii) *local collocations*, i.e. ordered sequences of tokens around the target word; (iv) pre-trained *word embedding*, integrated according to three different strategies, as in Iacobacci et al. (2016);[2] (v) *syntactic relations*, i.e. a set of features based on the dependency tree of the sentence, as in Lee and Ng (2002). SUPWSD allows the user to select appropriate cutoff parameters for features (i) to (iii), in order to filter them out according to a minimum frequency threshold.

Classification. The classification module constitutes the last stage of the SUPWSD pipeline. On the basis of the feature set constructed in the previous stage, this module leverages an off-the-shelf machine learning library to run a classification algorithm and generate a model for each sense-annotated word type in the input text. The current version of SUPWSD relies on two widely used machine learning frameworks: LIBLINEAR[3] and LIBSVM[4]. The classification module of SUPWSD operates on top of these two libraries.

Using the configuration file (Figure 2) the user can select which library to use and, at the same time, choose the underlying sense inventory. The current version of SUPWSD supports two sense inventories: WordNet (Miller et al., 1990)[5] and BabelNet (Navigli and Ponzetto, 2012)[6]. Specifying a sense inventory enables SUPWSD to exploit the Most Frequent Sense (MFS) back-off strategy at test time for those target words for which no training data are available.[7] If no sense inventory is specified, the model will not provide an answer for those target words.

3 SUPWSD: Adding New Modules

In this section we illustrate how to implement new modules for SUPWSD and integrate them into the framework at various stages of the pipeline.

Adding a new input parser. In order to integrate a new XML parser, it is enough to extend the `XMLHandler` class and implement the methods `startElement`, `endElement` and `characters` (see the example in Figure 3). With the global variable `mAnnotationListener`, the programmatic user can directly specify when to transmit the parsed text to the text preprocessing module. Instead, in order to integrate a general parser for custom text, it is enough to extend the `Parser`

[2] We implemented a cache mechanism in order to deal efficiently with large word embedding files.

[3] http://liblinear.bwaldvogel.de

[4] https://www.csie.ntu.edu.tw/~cjlin/libsvm

[5] https://wordnet.princeton.edu

[6] http://babelnet.org

[7] The MFS is based on the lexicographic order provided by the sense inventory (either WordNet or BabelNet).

```
public class NewXMLHandler extends XMLHandler {

    @Override
    public void startElement(String uri, String localName,
            String name, Attributes attributes) throws SAXException {

        NewLexicalTags tag=NewLexicalTags.valueOf(name.toUpperCase());

        switch (tag) {
            ...
        }
        this.push(tag);

    }

    @Override
    public void endElement(String uri, String localName, String name)
            throws SAXException {

        NewLexicalTags tag=NewLexicalTags.valueOf(name.toUpperCase());

        switch (tag) {
            ...
            case TEXT:
                this.mAnnotationListener.notifyAnnotations(...);
        }
        this.pop();

    }

    @Override
    public void characters(char ch[], int start, int length)
            throws SAXException {

        String sentence = new String(ch, start, length);
        switch ((NewLexicalTags)this.get()) {
            ...
        }

    }
}
```

Figure 3: An example of XML parser.

class and implement the `parse` method. An example is provided by the `PlainParser` class that implements a parser for a plain textual file.

Adding a new preprocessing module. To add a new preprocessing module into the pipeline, it is enough to implement the interfaces in the package `modules.preprocessing.units`. It is also possible to add a brand new step to the pipeline (e.g. a Named Entity Recognition module) by extending the class `Unit` and implementing the methods to load the models asynchronously.

```
public abstract class FeatureExtractor {

    private final int mCutOff;

    public FeatureExtractor(int cutoff){

        this.mCutOff=cutoff;
    }

    public final int getCutOff(){

        return this.mCutOff;
    }

    public abstract void load() throws Exception
    public abstract void unload();
    public abstract Class<? extends Feature> getFeatureClass();
    public abstract Collection<Feature> extract(Lexel lexel,
            Annotation annotation);
}
```

Figure 4: The abstract class modeling a feature extractor.

Adding a new feature. A new feature for SUPWSD can be implemented with a two-step procedure. The first step consists in creating a class

```
public abstract class Classifier<T,V> {

    public abstract Object train(AmbiguityTrain ambiguity);
    protected abstract double[] predict(T model, V[] featuresNodes);
    protected abstract V[] getFeatureNodes(SortedSet<Feature> features);

    public final Collection<Result> evaluate(AmbiguityTest ambiguity,
            Object model,String cls) {

}
```

Figure 5: The abstract class modeling a classifier.

that extends the abstract class `Feature`. The builder of this class requires a unique key and a name. It is also possible to set a default value for the feature by implementing the method `getDefaultValue`. The second step consists in implementing an extractor for the new feature via the abstract class `FeatureExtractor` (Figure 4). Each `FeatureExtractor` has a cut-off value and declares the name of the class through the method `getFeatureClass`.

Adding a new classifier. A new classifier for SUPWSD can be implemented by extending the generic abstract class `Classifier` (Figure 5), which declares the methods to train and test the models. Feature conversion is carried out with the generic method `getFeatureNodes`.

```
$ supWSD.jar train config.xml corpus keys
$ supWSD.jar test  config.xml corpus keys
```

Figure 6: Command line usage for SUPWSD.

4 SUPWSD: Usage

SUPWSD can be used effectively via the command line with just 4 parameters (Figure 6): the first parameter toggles between the train and test mode; the second parameter contains the path to the configuration file; the third and fourth parameters contain the paths to the dataset and the associated key file (i.e. the file containing the annotated senses for each target word) respectively.

Figure 2 shows an example configuration file for SUPWSD. As illustrated throughout Section 2, the SUPWSD pipeline is entirely customizable by changing these configuration parameters, and allows the user to employ specific settings at each stage of the pipeline (from preprocessing to actual classification). The `working directory` tag encodes the path in the file system where the trained models are to be saved. Finally, the `writer` tag enables the user to choose the preferred way of printing the test results (e.g. with or without confidence scores for each sense).

SUPWSD can also be used programmatically through its Java API, either using the toolkit (the

Tr. Corpus	System	Senseval-2	Senseval-3	SemEval-07	SemEval-13	SemEval-15
SemCor	IMS	70.9	69.3	61.3	65.3	69.5
	SUPWSD	71.3	68.8	60.2	65.8	70.0
	IMS+emb	71.0	69.3	60.9	**67.3**	71.3
	SUPWSD+emb	**72.7**	**70.6**	63.1	66.8	71.8
	IMS_{-s}+emb	72.2	70.4	62.6	65.9	71.5
	$SUPWSD_{-s}$+emb	72.2	70.3	**63.3**	66.1	71.6
	Context2Vec	71.8	69.1	61.3	65.6	**71.9**
	MFS	65.6	66.0	54.5	63.8	67.1
SemCor + OMSTI	IMS	72.8	69.2	60.0	65.0	69.3
	SUPWSD	72.6	68.9	59.6	64.9	69.5
	IMS+emb	70.8	68.9	58.5	66.3	69.7
	SUPWSD+emb	**73.8**	**70.8**	**64.2**	**67.2**	71.5
	IMS_{-s}+emb	73.3	69.6	61.1	66.7	70.4
	$SUPWSD_{-s}$+emb	73.1	70.5	62.2	66.4	70.9
	Context2Vec	72.3	68.2	61.5	**67.2**	**71.7**
	MFS	66.5	60.4	52.3	62.6	64.2

Table 1: F-scores (%) of different models in five all-words WSD datasets.

main class `SupWSD`, provided with the two static methods `train` and `test`, shares the same usage of the command line interface) or using an HTTP RESTful service.

5 Evaluation

We evaluated SUPWSD on the evaluation framework of Raganato et al. (2017)[8], which includes five test sets from the Senseval/Semeval series and two training corpus of different size, i.e. SemCor (Miller et al., 1993) and OMSTI (Taghipour and Ng, 2015a). As sense inventory, we used WordNet 3.0 (Miller et al., 1990) for all open-class parts of speech. We compared SUPWSD with the original implementation of IMS, including the best configurations reported in Iacobacci et al. (2016) which exploit word embedding as features. As shown in Table 1, the performance of SUPWSD consistently matches up to the original implementation of IMS in terms of F-Measure, sometimes even outperforming its competitor by a considerable margin; this suggests that a neat and flexible implementation not only brings benefits in terms of usability of the software, but also impacts on the accuracy of the model.

5.1 Speed Comparisons

We additionally carried out an experimental evaluation on the performance of SUPWSD in terms of execution time. As in the previous experiment, we compared SUPWSD with IMS and, given that both implementations are written in Java, we tested their programmatic usage within a Java program. We relied on a testing corpus with 1M words and more than 250K target instances to disambiguate, and we used both frameworks on SemCor and OMSTI as training sets. All experiments were performed using an Intel i7-4930K CPU 3.40GHz twelve-core machine. Figures in Table 2 show a considerable gain in execution time achieved by SUPWSD, which is around 3 times faster than IMS on Semcor, and almost 6 times faster than IMS on OMSTI.

	IMS	SUPWSD
train Semcor/sec.	$\sim$ 360	$\sim$ 120
train Semcor+OMSTI/sec.	$\sim$ 3000	$\sim$ 510
test/sec.	$\sim$ 110	$\sim$ 22

Table 2: Speed comparison for both the training and testing phases.

6 Conclusion and Release

In this demonstration we presented SUPWSD, a flexible toolkit for supervised Word Sense Disambiguation which is designed to be modular, highly customizable and easy to both use and extend for end users and researchers. Furthermore, beside the Java API, SUPWSD provides an HTTP RESTful service for programmatic access to the SUPWSD framework and the pre-trained models.

Our experimental evaluation showed that, in addition to its flexibility, SUPWSD can replicate or outperform the state-of-the-art results reported by

[8] http://lcl.uniroma1.it/wsdeval

the best supervised models on standard benchmarks, while at the same time being optimized in terms of execution time.

The SUPWSD framework (including the source code, the pre-trained models, and an online demo) is available at http://github.com/SI3P/SupWSD. We release the toolkit here described under the GNU General Public License v3.0, whereas the RESTful service is licensed under a Creative Commons Attribution-Non Commercial-Share Alike 3.0 License.

Acknowledgments

The authors gratefully acknowledge the support of the Sapienza Research Grant 'Avvio alla Ricerca 2016'.

References

Rodrigo Agerri, Josu Bermudez, and German Rigau. 2014. Ixa pipeline: Efficient and ready to use multilingual nlp tools. In *LREC*, pages 3823–3828.

Eneko Agirre, Oier Lopez de Lacalle, and Aitor Soroa. 2014. Random Walks for Knowledge-Based Word Sense Disambiguation. *Computational Linguistics*, 40(1):57–84.

Bharath Dandala, Rada Mihalcea, and Razvan Bunescu. 2013. Multilingual word sense disambiguation using wikipedia. In *Proceedings of the Sixth International Joint Conference on Natural Language Processing*, pages 498–506.

Claudio Delli Bovi, José Camacho-Collados, Alessandro Raganato, and Roberto Navigli. 2017. Eurosense: Automatic harvesting of multilingual sense annotations from parallel text. In *Proc. of ACL*.

Ignacio Iacobacci, Mohammad Taher Pilehvar, and Roberto Navigli. 2016. Embeddings for Word Sense Disambiguation: An Evaluation Study. In *Proc. of ACL*, pages 897–907.

Yoong Keok Lee and Hwee Tou Ng. 2002. An empirical evaluation of knowledge sources and learning algorithms for word sense disambiguation. In *Proc. of EMNLP*.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60.

George A. Miller, R.T. Beckwith, Christiane D. Fellbaum, D. Gross, and K. Miller. 1990. WordNet: an online lexical database. *International Journal of Lexicography*, 3(4):235–244.

George A. Miller, Claudia Leacock, Randee Tengi, and Ross T. Bunker. 1993. A semantic concordance. In *Proc. of HLT*, pages 303–308.

Andrea Moro and Roberto Navigli. 2015. SemEval-2015 Task 13: Multilingual All-Words Sense Disambiguation and Entity Linking. In *Proc. of SemEval*, pages 288–297.

Andrea Moro, Alessandro Raganato, and Roberto Navigli. 2014. Entity Linking meets Word Sense Disambiguation: a Unified Approach. *TACL*, 2:231–244.

Roberto Navigli. 2009. Word sense disambiguation: A survey. *ACM Computing Surveys*, 41(2):10.

Roberto Navigli and Andrea Moro. 2014. Multilingual word sense disambiguation and entity linking. In *COLING (Tutorials)*, pages 5–7.

Roberto Navigli and Simone Paolo Ponzetto. 2012. BabelNet: The Automatic Construction, Evaluation and Application of a Wide-Coverage Multilingual Semantic Network. *Artificial Intelligence*, 193:217–250.

Arantxa Otegi, Nora Aranberri, Antonio Branco, Jan Hajic, Steven Neale, Petya Osenova, Rita Pereira, Martin Popel, Joao Silva, Kiril Simov, and Eneko Agirre. 2016. QTLeap WSD/NED Corpora: Semantic Annotation of Parallel Corpora in Six Languages. In *Proceedings of LREC*, pages 3023–3030.

Llus Padr and Evgeny Stanilovsky. 2012. Freeling 3.0: Towards wider multilinguality. In *Proceedings of the Language Resources and Evaluation Conference (LREC 2012)*, Istanbul, Turkey. ELRA.

Mohammad Taher Pilehvar and Roberto Navigli. 2014. A large-scale pseudoword-based evaluation framework for state-of-the-art word sense disambiguation. *Computational Linguistics*, 40(4).

Alessandro Raganato, Jose Camacho-Collados, and Roberto Navigli. 2017. Word Sense Disambiguation: A Unified Evaluation Framework and Empirical Comparison. In *Proc. of EACL*.

Helmut Schmid. 2013. Probabilistic part-of-speech tagging using decision trees. In *New methods in language processing*, page 154. Routledge.

Kaveh Taghipour and Hwee Tou Ng. 2015a. One Million Sense-Tagged Instances for Word Sense Disambiguation and Induction. In *Proceedings of CoNLL*, pages 338–344.

Kaveh Taghipour and Hwee Tou Ng. 2015b. Semi-supervised word sense disambiguation using word embeddings in general and specific domains. In *HLT-NAACL*, pages 314–323.

Zhi Zhong and Hwee Tou Ng. 2010. It makes sense: A wide-coverage word sense disambiguation system for free text. In *Proc. of ACL System Demonstrations*.