Fextor: a Feature Extraction Framework for Natural Language Processing. A case study in Word Sense Disambiguation, Relation Recognition and Anaphora Resolution*

Bartosz Broda, Paweł Kędzia, Michał Marcińczuk, Adam Radziszewski,
Radosław Ramocki, Adam Wardyński

Institute of Informatics
Wrocław University of Technology
Wybrzeże Wyspiańskiego 27
50-370 Wrocław, Poland,
{bartosz.broda, pawel.kedzia, michal.marcinczuk, adam.radziszewski,
radoslaw.ramocki, adam.wardynski}@pwr.wroc.pl

Abstract. Feature extraction from text corpora is an important step in Natural Language Processing (NLP), especially for Machine Learning (ML) techniques. Various NLP tasks have many common steps, e.g. low level act of reading a corpus and obtaining text windows from it. Some high-level processing steps might also be shared, e.g. testing for morpho-syntactic constraints between words. An integrated feature extraction framework removes wasteful redundancy and helps in rapid prototyping. In this paper we present a flexible feature extraction framework called Fextor. We describe assumptions about the feature extraction process and provide general overview of software architecture. This is accompanied by examples of applications in hugely different NLP tasks. Namely, we show the application of Fextor in: word sense disambiguation, recognition of inter-chunk syntactic relations, semantic relations between named entities, as well as anaphora resolution.

Keywords: feature extraction, word sense disambiguation, relation recognition, anaphora resolution, shallow parsing, derivational relations, named entities

1 Introduction

Feature extraction process is a preliminary step of Machine Learning (ML) techniques. When working on theoretical aspects, one often uses standardised benchmarking datasets, where features are already provided. E.g., the iris flower dataset [1] allows to put the focus solely on the performance of the algorithm.

* This work was financed by the National Centre for Research and Development (NCBiR) project SP/I/1/77065/10.

However, employing ML for real-world problem solving requires the consideration of feature extraction process, first and foremost. In some domains, the nature of features is constrained. For example, when dealing with measurements taken by some device, features can only include data from the sensors of the measuring equipment. On the other hand, there are also domains that provide more freedom when describing the given problem in terms of features.

One of such domains is Natural Language Processing (NLP). In many NLP tasks we deal with textual data from which the features must be extracted, as ML algorithms cannot process text directly. The challenge is to select features that are informative and discriminative. Often an initial set of features is established and some experiments are conducted to improve the set, modyfing, adding or removing features. Having an easy to use and flexible framework for feature extraction can help and speed up the process considerably. This is especially important when dealing with many complex feature types, where being able to describe features in some high-level way is a necessity. Moreover, many steps of feature extraction process are common among different NLP tasks. Reading a corpus, selection of text window around interesting words in text are examples of tasks that can and should be abstracted away while constructing features.

It is hard to find an open source feature extraction framework that could be applied to Slavic languages, esp. to Polish. On the other hand, there are a few NLP frameworks, which contain some form of feature extraction (see Sec. 2). One of our major requirements for feature extraction framework was flexibility. This was motivated by the wide range of NLP research areas we are dealing with: from part-of-speech tagging through word sense disambiguation to anaphora resolution. We have gained some experience with feature extraction while working separately on different problems, thus we had different opinions on what the feature extraction process should look like. Nevertheless, after a few fruitful discussions and brain-storming sessions, common patterns have emerged. We decided to undertake the difficult task of constructing an integrated feature extraction framework called Fextor[1].

We had a few design goals that guided the entire process of constructing the software. The *flexibility* mentioned earlier was very important. Everyone of us needed to solve a different problem, thus without flexibility the undertaking would fail. Opposite to flexibility stands *simplicity*: we joined our efforts to create Fextor in order to reduce total cost of feature extraction. The ability to describe any feature we would need was also required, so we strived for *expressiveness*. The tools had to work with our standard software libraries, most importantly: `wccl` and `corpus2` [2]. We are constructing practical tools, so we need to process huge corpora in an *efficient* way. We have already constructed an integrated ML framework within LexCSD [3], which enables usage of different classifiers from different software packages, thus the integration with LexCSD[2] was naturally on the list of requirements.

[1] Fextor home page: `http://nlp.pwr.wroc.pl/fextor`

[2] LexCSD home page: `http://nlp.pwr.wroc.pl/lexcsd`

This paper is organised as follows: in the next section we give an overview of existing approaches to feature extraction in NLP. Next, we give an overview of Fextor architecture: we introduce the concepts of *slicers*, *iterators*, *contexts* and *feature generators*. In Section 4 we show some applications. Some of the applications are already developed and some are being under development. The described applications include: word sense disambiguation, recognition of inter-chunk syntactic relations and semantic relations between named entities, anaphora resolution and classification of derivational relations. In Section 5 we summarise the paper and point direction of further works.

2 Related Works

There exist several common frameworks that may be used to solve NLP problems using ML classifiers. Those frameworks usually provide some degree of support for feature extraction.

NLTK [4] is an open source suite containing a large library of Python modules for various NLP tasks. It also includes implementations of various ML algorithms. Although the abundance of practical modules is quite impressive, the user must actually write custom code for feature extraction. What is more, the framework is (somewhat implicitly) targeted at processing English, and hence, no support is given to deal systematically with structured tagsets that are characteristic for inflective languages.

Freeling [5] is another open source suite for language analysis equipped with some ML components. The set of available classifiers is quite limited in comparison with NLTK. What is more, the suite is somewhat loosely coupled, e.g. POS tagging and Word Sense Disambiguation modules are based on completely different APIs. Nevertheless, one component seems particularly interesting for our considerations, namely the *Fex* feature extraction module [6]. The module is used in Freeling for named entity recognition and coreference resolution. Fex is able to process a corpus in a specially prepared input format into a list of feature vectors. Each sentence is processed separately. Features are expressed in a simple domain language. The language allows to extract values of word forms, tags, chunk tags and grammatical roles from different positions in the input corpus (note that this information must be explicitly provided as per-word labels in the input corpus). Fex language expressions may retrieve value of a particular category (tag, form, etc.) from a position relative to the word being processed; it is also possible to retrieve ranges of values or complex values, composed of values of different categories. Unfortunately, POS tags are treated as atomic symbols and no mechanism is offered to decompose them into values of particular grammatical categories (e.g. case of a noun), not to mention calculation of complex morphosyntactic functions (e.g. tests for agreement between a noun and its adjective modifier, typical for Slavic languages). The language also does not seem to support iteration over larger units, such as existing syntactic annotations or named entities.

WCCL [2] is another open source toolkit for feature generation. The toolkit is targeted at Polish[3] and distinguishes itself with the support for positional tagsets and ability to 'understand' complex tags. As in the case of Fex, WCCL comes with a specialised domain language for writing functional expressions evaluated on annotated sentences with some token highlighted as the centre being processed at the moment. The language is quite expressive, allowing for extraction of simple and complex features. Simple features include orthographic forms, lemmas and grammatical class and values of grammatical categories from tokens. Complex features include the following (the list is not exhaustive):

- test for set-theoretic relations between two sets of strings or sets of symbols defined in the currently used tagset (WCCL supports ambiguity in the input corpus, e.g. some tokens may be assigned more than one tag, resulting in sets of possible values of some grammatical categories);
- constraint satisfaction search, i.e. a given token range is sought for a token satisfying the given predicate and some function is finally evaluated on the token found (e.g. the grammatical case retrieved from a noun being a likely verb object);
- tests for morphological agreement on given grammatical categories, either between two given tokens or between whole token range;
- joining features with help of standard logical connectives and functional-style `if` clauses.

In spite of the mentioned expressiveness of WCCL, there are two limitations particularly important to our considerations. First, as in Fex, WCCL features cannot reach beyond the boundaries of the sentence being processed. This is particularly unfortunate for application in anaphora resolution and word sense disambiguation, where the features typically employed refer to much broader context. The other drawback is that the toolkit is limited to 'token-oriented' feature generation, that is to say, one iterates over subsequent tokens and given functional expressions are evaluated on subsequent tokens along with their local neighbourhood. This is natural for tasks that may be cast as sequence labelling problems, where the sequence being labelled consists of tokens, e.g. morphosyntactic tagging, chunking. Other use cases may require features describing entities other than single tokens, e.g. named entities, syntactic chunks or even pairs of such annotations (to classify inter-annotation relations).

3 Fextor Architecture

Fextor is a Python 2.6 package which consists of several modules: *document* — represents processed document, *reader* — allows to read documents, *iterators* — to iterate over elements of interest in a document, *context* — a trimmed part of the document representing local context, *features* — all available features.

[3] WCCL has been developed for Polish, although it is able to process any language as long as the tagset conforms to a few additional requirements, cf. [7].

Generally, it takes input files as arguments, while details regarding types of examples to take from documents and features to extract from the examples are set in a configuration file. The results are written in CSV format to the standard output or to a selected file. The process of feature generation may be sketched as follows:

- Read the document and make its internal representation;
- Iterate over the `Document` and return pointers to elements of interest (e.g. tokens) where iterator stopped. Type of the `Iterator` determines on which elements the iterator should stop, e.g., `OnlyBaseIterator` can stop only on the words, whose base form occurs in the given list;
- For each `Pointer` from iterator:
  - cut the `Context` — size of the context depends on the slicer used, e.g., *n*-th tokens from left/right, sentence, document, . . . ;
  - make the `Pointer` to the element of interest in the `Context` (it may be different than in the document);
  - for each feature specified in the config file, pass pair of `Context` and `Pointer`, apply the feature function and get the resulting feature value;
- Return the results to the user (to the standard output or a file).

The input, be it a single file or a multi-document corpus, is read into an internal representation. Next, the document (or documents) is processed in an appropriate way, as decided by the user in the configuration file. The result of processing are values returned by selected feature extractors. In case of a corpus, the result constitutes all values extracted from all the documents from the corpus.

The class governing the execution flow is `Fextor`. What follows is a functional overview of the architecture, roughly as implemented in the `Fextor` object.

3.1 Fextor I/O

List of input files that constitute a training corpus is provided in the arguments passed to Fextor. The files are read using Corpus2, a C++ module that offers data structures and routines for reading and writing annotated corpora in different formats, supporting configurable tagsets [7].

Two formats are supported so far — *poliqarp* and *document.* The former is a compact, indexed binary representation of a morphosyntactically annotated corpus, as generated by the Poliqarp corpus indexing and querying system [8]. The latter is an XML-based format, so-called *CCL format* (as supported by Corpus2 and WCCL), being a simple extension of XCES with the possibility to annotate chunk-style annotations, their heads and inter-annotation relations. As the relations may hold between annotations from different sentences (typical example being anaphora), while the rest of annotation is limited to sentence boundaries, relations may be read from a separate file. The *document* format also supports batch processing mode, when input files are text files that list the actual XML documents.

The internal representation of input files is handled through the `Document` class. This class holds information about paragraphs, sentences, tokens, relations and annotations from the given document. Tokens in the document are numbered. The document interface allows to get annotations from the document and get all relations from the document. Internal representation of the document is created by a *Reader* that uses Corpus2 classes. Created document is the basis for further processing and for features extraction — this step may be understood as a first phase in the process of feature extraction.

The Fextor output is produced with Python's `csv` module, using `excel` dialect, although with semicolon (`;`) rather than the comma used as the delimeter (more in line with the default CSV file format used by recent spreadsheet software, after which the dialect takes its name). Essentially, the first line is a semicolon-separated list of feature names, and what follows are semicolon-separated lists of extracted values for corresponding features, one example per line.

3.2 Iterators

Once a document is read, it is passed on to an iterator, specified in the Fextor configuration file directly as a Python class. Also, the configuration file can specify parameters that will be passed to the iterator.

The iterators pass through the document and return pointers to examples, one by one as Python generators do. We currently support three types of *pointers*: *token*, *annotation* and *annotation pair*.

`TokenPointer` points to a single token from text. `AnnotationPointer` conversely corresponds to an *annotation*, which is essentially a sequence of tokens belonging to a named *annotation channel*. In our approach, the annotations are organised in independent *channels*; each channel may contain a number of non-overlapping annotations, being sequences of consecutive tokens (as in *chunking* task). For instance, one may have a `chunk_np` channel containing NP chunks, `person_nam` channel containing named entities describing personal names. An annotation may also be given additional properties (key–value string pairs); besides, there is a possibility to annotate annotation's syntactic head (on of its tokens is then highlighted as head). We also provide `AnnotationPairPointer`, which encompasses two selected annotations; this pointer is especially useful when classifying relations holding between two annotations.

Only one iterator, with one type of example is supported in a single Fextor run i.e. features designed for `TokenPointer`s won't work for `AnnotationPointer`s (for code reuse one could e.g. provide a wrapper that takes head of annotation and invokes some token feature on it).

`EveryTokenIterator` is an example of an iterator which iterates over all tokens in order of their occurrence in the document. Additionally, Fextor offers iteration over annotations from selected channels (`NamedAnnotIterator`) or pairs of annotations as specified with names of the channels (`SentAnnotPairIterator` - limited to sentence boundaris) or with a relation name that should connect the pair (`RelationIterator`).

Fextor outputs extracted features one line per example in the same order as the examples are coming from the selected iterator (in the order of documents coming from `DocumentProvider` i.e. as they were provided in input).

3.3 Slicers and Context

Along with the notion of a pointer, we have *contexts*. A context is a list of tokens, sentences, paragraphs, relations and annotations in the neighborhood of a pointer. It is essentially a fragment of document "cut" around a pointer with the use of a `Slicer` object. In general, slicer is a mechanism used to trim the given context. Size of the returned context depends on the type of slicer used. Sample slicers are:

- `DocumentSlicer` — Returns copy of the document.
- `TokenSlicer` — Context is trimmed to the given range of tokens. N tokens on the right and M on the left site of the given pointer.
- `SentenceSlicer` — Context is trimmed to the boundaries of a sentence that contains the given pointer. Note that the division into sentences is taken intact from the document, no additional sentence splitting is performed.
- `MultiSentenceSlicer` — Context is trimmed to the given range of sentences: N sentences on the right side and M to the left of the given pointer.

A slicer receives a document and a pointer to a fragment in the document. The slicer returns a pair of: context and mapping of the given pointer in document to pointer in the generated context, type of the returned pointer is the same as the used pointer.

The idea is that a feature extraction may be focused on (or limited to) a context around a pointer, so instead of document as a whole, just a pointer and corresponding context is provided (although as mentioned above, context may as well encompass whole document if need be). E.g. WCCL features naturally take a context of a sentence. The type of a slicer required by a feature is specified in the feature definition in the configuration file.

3.4 Feature Generators

The last step of processing is to generate the values of the features for every element returned by iterator. Classes that perform feature generation implement one of the following interfaces:

- `TokenFeatureGenI` if the extraction centers around tokens;
- `AnnotationFeatureGenI` in the case of annotation processing;
- `AnnotationPairFeatureGenI` for pairs of annotations.

As already mentioned, only one type of elements is supported throughout a single Fextor run.

The list of features to be extracted is defined as a `features` parameter in the main extractor section in the configuration file. Each feature has a dedicated configuration section. The section contains name of Python class performing the feature generation and a set of parameters required to initialize and run the generator. This allows to write more generic classes that are subsequently parametrized, e.g. we have classes that directly support WCCL functional expressions. The last element defined in the feature section is the name of context slicer used by the feature generator.

`PosFeature` is a sample implementation of feature generator which extract a set of parts of speech tags from specified context. As it is token feature the generator implements the `TokenFeatureGenI` interface. In case when left or right boundary (specified as relative offset from the current token) is outside the context, then for the missing tokens the '`None`' value is taken. `PosFeature` can perform words filtering according to specified list of stopwords (the `stopwords` parameter). For words present on the list the generator returns empty string. For non-empty sets the generator return a list of part of speech tags separated by semicolon. Below is a sample definition of a feature using the `PosFeature` generator:

```
[wsd_pos]
class      = fextor.features.tokens.PosFeature
slicer     = fextor.contexts.slicer.TokenSlicer(-20, 20)
flex       = adj,adja,adv,aglt, ..., tnum,tsym,winien
tagset     = kipi
stopwords = stopwords.txt
conversion = numeric_set
```

Where: *wsd_pos* — is a unique feature name; *class* — name of Python class used as a generator, *slicer* — type of slicer used to make contexts, here `TokenSlicer` with a range of 20 words from the left and the right side around the pointer; *flex* — a list of POS values of words which are to be processed; *tagset* — the tagset used in the corpus read; *stopwords* — path to a file with stopwords; *conversion* — type of feature value conversion used in Fextor output, as expected by the fextor2lexcsd application. Other `TokenFeatureGenI` features, are: `BaseFeature` — to extract base form of the words, `OrthFeature` — to extract words in form as occurs in the text.

3.5 Interfacing with LexCSD

To facilitate the use of LexCSD [3] which contains a framework for Machine Learning, Fextor package includes `fextor2lexcsd` converter to obtain the LexCSD matrix format out of the CSV format described above.

As described in Sec. 3.1, the Fextor output is a CSV file containing (in successive columns) values generated by the features. These values can be transformed to the new predetermined types of values. Each feature declared in the config file should contain information about the desired type of conversion. The following types of conversion are supported:

- `none` — no conversion is done.
- `numeric` — each generated value has unique number. All occurrence of this value are replaced by this number.
- `binary` — each feature is expanded to a set of new features, each corresponding to its subsequent values. The new features are binary: $F_n^v = 1$ when the original feature f_n has value of v, 0 otherwise. For example, assuming three features `A B C` with the following values:

```
A B C
-----
1 3 4
2 2 4
```

  Binary conversion will yield the following result:

```
A:1 A:2 B:2 B:3 C:4
-------------------
 1   0   0   1   1
 0   1   1   0   1
```

- `binary_set` – each generated value is treated as set with values separated by the comma. Then for all values from this set `binary` conversion is made. For example:

```
   A      B
--------------
{1,2,3} {2,3}
 {2,4}   {1}
```

  is transformed to:

```
A:1 A:2 A:3 A:4 B:1 B:2 B:3
-----------------------
  1  1   1   0   0   1   1
  0  1   0   1   1   0   0
```

- `sparse_binary` – same as `binary`, but value of 0 are not set.
- `sparse_binary_set` – same as `binary_set`, but value of 0 are not set.
- `continuous_vector` – this is an arbitrary fixed size vector of real numbers.

4 Example Applications

Feature extraction is not a goal on its own. Thus, in this section we will show a few applications of Fextor. Some of them are deployed and some are being under development. We have already achieved good results for Word Sense Disambiguation (WSD) and classification of derivational relations between words. The results for inter-chunk relation recognition are promising. The remaining tasks are still under development, so we will confine our discussions to the feature sets we have designed and the role of Fextor in solving the tasks.

4.1 Word Sense Disambiguation

Many words have more than one sense (lexical meaning), but usually only one of them is activated in a given context. The typical example of such words are: *bank* (river bank or financial institution), *ring* (making a call or physical object) or *line* (36 entries in WordNet [9], e.g., queue, cable, shape). The task of Word Sense Disambiguation (WSD) is to choose the right sense for a word in a context. With availability of manually annotated corpora WSD can be described as a classification problem. That is, given the occurrence of ambiguous word in text the classifier has to assign a sense label on the basis of features extracted from surrounding context.

As we had some experience with ad hoc feature extraction for WSD [10, 3] we already had some idea what features we should extract. Also, lots of ideas for WSD features can be found in the literature, especially the important works of [11, 12]. Starting with extraction of bag of words features in vector space model is usually not a bad idea in WSD as this feature type gives the most coverage [11]. In this model the features are represented as a high-dimensional vector, where every dimension corresponds to occurrences of different words in contexts of ambiguous words. The task of feature generator is to mark occurrences of words in a surrounding text window. Words are filtered by their parts of speech as not every part of speech is a good discriminator (e.g., nouns are typically useful, prepositions not so much). The text window is usually large, e.g., 100 words around the target word.

The word order is lost in a bag-of-words representation. Thus another potentially useful feature is a dense vector that holds the information about words occurring on given position relatively to the ambiguous word. Now, the vector's dimensions correspond to the position, and the value represents a numerical identifier of a word occurring on that position. The text window is shorter than in the bag-of-words representation (e.g., 10 words). With even shorter text window (of 2 or 3 words) and some additional statistical filtering one can obtain a good approximation of *collocations*, which where shown to improve WSD performance [13]. Another simple modification of aforementioned feature type leads to another feature: instead of marking occurrence of words in short text window we can mark occurrences of parts of speech.

With the help of WCCL language we can extract more complex features. So far we have experimented with morpho-syntactic constraints similar to the ones used in extraction of Measures of Semantic Relatedness (MSRs) from text corpora [14]. Those features check for a morphological agreement between (ambiguous) noun and an adjective, try to capture a potential predicate for a noun, seek nouns in coordination and check for modification of ambiguous noun by another noun in genitive. So far the preliminary results with employment of morpho-syntactic features are not encouraging. This might be caused by the fact that those operators were written by hand with the goal of the highest precision as possible. Thus the coverage suffered and the features are rather sparse. As so far we have focused only on nouns we haven't tried to employ more sophisticated WCCL features that were employed for construction of verbal MSRs [15].

The currently employed features give state-of-the art results for Polish. That is, results we achieve with help of Fextor are comparable to results achieved in other works [16, 10, 3]. We are working on the extension of already employed feature set. We want to employ a Polish wordnet called Słowosieć [17], MSR directly and additional forms of morpho-syntactic processing using WCCL or more sophisticated parsers [18, 19].

4.2 Recognition of inter-chunk syntactic relations

Another application of Fextor is that of shallow parsing. More specifically, we are working on a simple ML module that recognises selected syntactic relations between syntactic chunks, as defined in the *Polish Corpus of Wrocław University of Technology* (*KPWr*) [20, 21]. There is already a tool suitable for recognition of the chunks, namely the memory-based chunker proposed in [22]. The task described here is that of recognition of partial predicate-argument structure, which is limited to relations holding between VP and NP chunks.

The task may be formulated as a classification problem: given a sentence with two chunks highlighted, classify what type of syntactic relation holds between them. We consider three options: no relation, subject or object. We assume that the input sentence has been tokenised, morphosyntactically tagged and annotated with NP and VP chunks. What is more, each chunk is augmented with information about the location of its syntactic head (always one-token-long). The preliminary experiments described here make use of reference NP/VP annotation, therefore the evaluation presented in this section does not include chunking errors.

In terms of feature extraction, the use case requires iteration over annotation pairs (the annotations being chunks). During training, for each sentence a set of all the possible VP–NP pairs is generated (at least in our baseline model; the set could be limited by some general syntactic constraints). Each such pair, along with the whole sentence as context is employed to generate feature values for classification; the decision class is the desired relation type. The performance phase consists in generating pairs in the same fashion, feature generation and having the trained classifier predict the relation type between the chunks. In terms of Fextor configuration, we employ *sentence annotation pair iterator*, features exploiting WCCL expressions, as well as *annotation pair sentence slicers*. Below are snippets from the configuration file used:

```
[Extractor]
iterator = fextor.iterators.SentAnnotPairIterator
first_channels = chunk_vp
second_channels = chunk_np
features = class_hd1 class_hd2 np_btw ... label

[class_hd1]
```

```
class      = fextor.features.pairs.WCCLPairFeatureGen
slicer     = fextor.contexts.slicer.AnnotationPairSentenceSlicer()
operator   = class[$Hd1]
type       = symset
file       =

[class_hd2]
class      = fextor.features.pairs.WCCLPairFeatureGen
slicer     = fextor.contexts.slicer.AnnotationPairSentenceSlicer()
operator   = class[$Hd2]
type       = symset
file       =

[np_btw]
; the number of chunk_np annots between ann1 and ann2
class      = fextor.features.pairs.DistanceInAnnotations
slicer     = fextor.contexts.slicer.AnnotationPairSentenceSlicer()
channel    = chunk_np
```

The SentAnnotPairIterator generates all annotation pairs from each sentence, where first annotation is of chunk_vp type, while the second one is a chunk_np. Most of the features employed are based on WCCL functions. The role of Fextor here is to set position variables to point to the expected tokens, e.g. $Hd1 is automatically set to the location of the head of the first annotation (VP in this case); this allows for using concise and expressive functional expressions. E.g., the class_hd1 feature is responsible for returning the possible values of the grammatical class (POS) of the VP syntactic head. Some of our features are based on functionality that is not available in WCCL at all, e.g. np_btw returns the number of NP chunks that occur in the range between the annotations of the considered pair.

A reasonable feature set for this task will include features that characterise both chunks themselves, their local context, as well as the material that comes in-between. As the chunks may be of different length (the same holds for the intervening material), it is necessary to focus on features that refer to particular tokens (at least syntactic heads of both chunks), but also to capture some abstract properties of units bigger than single tokens. The feature set employed here consists of the following items:

1. lemma of heads of both annotations;
2. POS of both heads;
3. grammatical case of the second annotation head (which is an NP);
4. does the second annotation start with a preposition? (the annotation schema of KPWr puts both real NPs and actual PPs into one group called NP);
5. set of POS values collected from all the tokens belonging to the first annotation and the same for the second one;
6. set of POS values collected from all the tokens occupying the range between the annotations;

7. set of values of grammatical case collected from all the tokens occupying the in-between range (some tokens are not specified for case at all, e.g. adverbs and punctuation; such tokens do not contribute to the set);
8. a set of string codes, indicating whether some of important items appeared in the in-between range; this is based on dictionary look-up, where the dictionary was prepared manually and assigns 45 lemmata to one of the following categories: coordinating conjunct, relative pronoun, comma or ambiguous between the two first;
9. how many NP and VP chunks appear in the in-between range;
10. does the NP follow the VP or vice-versa.

Our experiments consisted in using Fextor equipped with the above feature set and configuration file in tandem with the LexCSD ML framework [3]. This allowed us to conveniently test several classifiers, coming from different software packages. Our data set contained 4028 relation instances taken from the part of the KPWr that has been annotated so far. Our experiments are based on 10-fold cross-validation. The results of our preliminary experiments are presented in Table 1. P, R, F columns present respectively values of *precision*, *recall* and *F-measure*. The results labelled TiMBL: k15 correspond to the Tilburg Memory-Based Learning package [23] with $k = 15$ neighbours, using the overlap metric. The results labelled TiMBL: k15,mM are also obtained using TiMBL, but using Modified Value Difference metric. The last row contains results obtained using the LibLINEAR classifier [24] assuming the `C3` regression function with $C = 0.25$.

Classifier	**Subject**			**Object**		
	P	R	F	P	R	F
Naive Bayes	61.38%	54.67%	57.33%	58.30%	87.56%	69.87%
Decision Trees (J48)	69.00%	45.38%	53.64%	65.18%	75.34%	69.33%
RIPPER (JRip)	74.57%	45.23%	55.37%	66.37%	74.43%	69.07%
TiMBL: k15	73.44%	58.08%	63.88%	66.98%	81.70%	73.42%
TiMBL: k15,mM	74.67%	58.74%	64.88%	72.72%	76.46%	74.41%
LibLINEAR: S3,C=0.25	73.10%	65.77%	68.93%	69.15%	83.71%	75.50%

Table 1. Results for syntactic relation recognition.

The results obtained so far for syntactic relation recognition are not striking but anyway promising. Note that the functionality already implemented in Fextor allows to test more advanced features, e.g., also examining the NPs and VPs appearing beyond the range constituted by both annotations and the in-between material.

4.3 Recognition of Semantic Relations Between Named Entities

The next application of Fextor is recognition of semantic relations between named entities. The set of semantic relation categories to be recognised is limited to a predefined set. The task is also limited to relations between named entities present in the same sentence, thus this application is very similar to the task of inter-chunk syntactic relation recognition. The other assumption is that the relations must be supported by some premises stated in sentence — we do not intend to recognise relations supported only by external knowledge. We assume that the named entities will be recognized beforehand with the NER tool presented in [25] extended to 56 categories of proper names.

The recognition of semantic relations between named entities requires iteration over selected pairs of named entities according to a predefined dictionary of pairs of named entity categories. Such enumeration is implemented in the `SentAnnotPairIteratorByDict` iterator. Below is a sample definition of the iterator for *affiliation* relation:

```
[Extractor]
iterator = fextor.iterators.SentAnnotPairIteratorByDict
dict = reference_dict.pkl
relation = affiliation
features = r1 r2 r3 r4 r5 r6 ... context docname relations
```

where `dict` is a path to the dictionary of valid pairs of named entities and `relation` is a name of current relation category. The `context` and `docname` features are used to identify any pair of annotations within set of documents after the classification is done.

In the preliminary experiment we have utilized first-order logic rules obtained by applying Inductive Logic Programming over the data represented as a set of predicates (the description of the predicates used to describe the data is presented in [26]). The feature value is *true* if the rule can be proved and *false* otherwise. The rules are in fact short patterns over the sequence of tokens and token dependency tree around the annotations. The feature generator utilizes external module which transforms every sentence into set of predicates and Yap interpreter for testing the rules.

Below is a sample feature definition which utilizes one of the rules generated for *composition* relation:

```
[r12]
class = fextor.features.ilp.IlpRulesFeature
slicer = fextor.contexts.slicer.AnnotationPairSentenceSlicer()
tagset = nkjp
predicates = predicates.txt
rules = relation(A,B,composition) :-
   annotation_token(B,C), jump_left(D,C), jump_left(E,D),
   token_base(E,word_w), jump_left(F,E), token_base(F,meta_COMMA),
   token_dependency(G,C,adj).
```

We have tested two models of feature space. In the first model, namely *sep-feat*, the feature space for every category of relations contained only rules generated for that category. In the other model, namely *join-feat*, the feature space for every category of relations was the same and contained all features generated for all categories of relations. We have tested a set of basic classifiers available in the LexCSD enviroment which can handle binary features, including: BayesianLogisticRegression, BFTree, ComplementNaiveBayes (CNB), DecisionTable, LMT, NaiveBayes. The classifiers were trained on the training set and evaluated on the tune set[4]. Then, we have selected the classifiers which obtained the best results on the tune set and evaluated them on the testing set.

In Table 2 we present the comparison of results obtained by the hand-crafted rules, ILP rules and classifiers using ILP-rules as features. In case of 5 categories of relations we obtained better results than ILP rules. The highest improvement was obtained for *origin* relation by 20.95 points of F-measure. The other two categories obtained slightly lower results by 1.39 and 0.16 points of F-measure for *composition* and *alias* relation respectively. For the last relation category, i.e. *neighbourhood*, we obtained much worse results.

Relation	**Rules**	**ILP**	**Classifier**			
	F [%]	F [%]	Model	Type	F [%]	Change
affiliation	28.90	46.12	sep-feat	DecisionTable	**52.73**	+6.61
alias	24.35	**46.03**	join-feat	BFTree	45.87	-0.16
composition	42.86	**65.63**	sep-feat	NaiveBayes	63.64	-1.39
creator	12.50	32.35	join-feat	CLR+CNB+RT	**41.03**	+8,68
location	26.00	29.58	sep-feat	BLR	**33.46**	+3.88
nationality	30.77	44.44	sep-feat	RandomTree	**54.55**	+10.11
neighbourhood	10.53	**33.33**	sep-feat	DecisionTable	15.38	-17.95
origin	57.14	41.27	join-feat	BFTree	**62.22**	+20.95

Table 2. Comparision of results obtained by hand-crafted rules, ILP rules and classifiers using ILP-rules as features.

We observed that many incorrect relations were recognized in sentences which contain a pair of named entities connected with a relation and a set of irrelevant named entities not connected with any relation. Some of the rules generated by ILP had form of two unlinked patterns matching the context of source and target named entities. Many of the false relations were recognized between named entities appearing far away from each other. We expect, that named entities appearing far away from each other are rare to be in any relation. We could eliminate them by introducing some features which will reflect the distance between the named entities.

In the preliminary experiment we have utilized only one type of features. However, the set of features can be extended by features used in the inter-chunk syntactic relations recognition task and also:

1. relative position of named entities (A after B, A before B, but also A inside B, etc.),

[4] The training, tune and testing sets are described in [26].

2. location of named entities inside chunks (for source and target annotation),
3. syntactic relations for chunks containing the named entities (for source and target annotation),
4. wordnet-based generalisation of words (direct and indirect hyperonyms of words),
5. bag of words in *before*, *between* and *after* context (according to [27]),
6. verbs and prepositions closest to named entities (for source and target annotation),

The extended set of features is going to be tested in the future experiments.

4.4 Anaphora Resolution

Anaphora is another example of a relation held between two items in text, namely when one fragment (*anaphor*) points back to a previously mentioned item in the text (*antecedent*). As such, *anaphora resolution* (resolving an anaphor to the expression it refers to) can be treated in similar manner as abovementioned examples concerning syntactic or semantic relations. Specifically, anaphora resolution can be presented as a classification of pairs of expressions, whether given relation holds or not. One important distinction though is that anaphora in general transcends sentence boundaries.

One type of anaphoric relation is direct, identity-of-reference anaphora, where the anaphor and the antecedent have the same referent in the real world. In essence, they are coreferential, and this type of anaphora overlaps with the issue of coreference resolution. A machine learning approach to coreference resolution of noun phrases has been presented in [28] and expanded upon in other publications, such as [29].

The approach represents the problem as a classification task as suggested above, however special care is taken in order to avoid dealing with all possible combinations of suspected anaphor/antecedent pairs. Positive cases are taken from pairs of anaphor and its closest antecedent. To limit the number of negative cases, they are taken only from pairs of actual anaphor with a noun phrase that is not its antecedent, but is textually placed between the anaphor and its real closest antecedent. A set of features describes the pairs and a model is built to predict whether relation holds or not, based on the values of the features.

Such approach should also extend to other anaphoric relations, such as bridging anaphoras, where the relation between anaphor and antecedent is indirect, e.g. anaphor is a part of antecedent or in its possession.

Feature extraction for this process can be naturally facilitated by Fextor. The example selection can be performed by a dedicated iterator, or with a separate preprocessing step that would mark interesting pairs, allowing for a simple iterator.

Initially we are basing our feature selection on [29], focusing on

- lexical features such as textual similarity,
- grammatical features such as agreement on gender and number

- semantic features such as WordNet semantic class,
- positional features such as distance in tokens or sentences.

It is also conveniently possible to try to employ features selected for classification of other types of linguistic relations as mentioned in other examples herein.

4.5 Classification of Derivational relations

Classification of derivational relations between words is a task of assigning single class to a pair of words. The first word is called a *derivative* and the second one is referred to as a *derivative base*. In our experiments there were four coarse-grained derivational relations (classes): femininity, inhabitant, markedness and semantic role. Markedness can be subdivided into three subclasses: diminutivity, augmentativity and young being. Respectively semantic role can be subdivided into: agent of hidden predicate and location of hidden predicate. More details about Polish derivational relations and their automatic generation can be found in [30].

There are multiple possible knowledge sources one can use in the task of derivational relation classification, e.g., corpora, wordnet and textual association between derivative and its base form. As Fextor was designed to extract features from corpora by specialized iteration method one additional step had to be introduced to the processing pipeline, unlike in previously described applications. We addressed the problem by construction of an artificial corpus file. The 'fabricated' corpus is made up from only one sentence which contains N tokens. Every token represents one instance of derivational relation, so there are N instances of relations. In addition each token contains special set of properties describing particular instance of relation. Following properties are needed for feature generation:

- derivative and derivative base words,
- relation instance class,
- identifiers of derivative and derivative base synsets in plWordnet,
- every possible morphological analysis of derivative and its base.

To generate features we need to iterate over every token in the artificial corpus.

Fextor was used to generate following set of features:

- plWordnet domain identifiers for derivative and base,
- bag of plWordnet synset identifiers describing semantic classes for derivative and its base,
- suffix of derivative, which is not included in its base form,
- length of derivative's suffix,
- bag of part of speech, grammatical numbers, genders and cases for derivative and its base,

- distributional semantics vector for derivative and its base (the simplest model could be a cooccurrence with any word in fixed sliding window, generated by corpora iteration), vectors are taken from precalculated matrix which is available as a resource for feature generator.

In order to evaluate our approach we used instances of 7 derivational relation subtypes from *plWordNet 1.6*. For our experiment 10-fold cross-validation scheme was applied. Pairs for each relation subtype were randomly divided into 10 subsets. One subset per relation subtype was used for testing in each iteration.

Table 3 presents cross-validation results for coarse-grained and fine-grained class subdivision. We used multiclass *SVM* classifier from *LibLINEAR*.

Relation	**Precision**	**Recall**	**F-score**
femininity	95.09	98.36	96.70
inhabitant	91.26	87.85	89.52
markedness	89.70	98.16	93.74
diminutivity	89.31	97.17	93.08
augmentativity	70.80	62.02	66.12
young being	53.85	26.92	35.90
semantic role	84.79	93.23	88.81
agent of hidden predicate	86.74	96.09	91.17
location of hidden predicate	83.33	83.33	83.33

Table 3. Cross-validation results of multiclass semantic classification (coarse-grained relations are in italic).

4.6 Other possible usages

In the above sections we presented Fextor usage scenarios corresponding to NLP tasks we are currently involved in. There is, however, a wide range of other possible applications, where the inclusion of the presented framework could bring practical profits. The possible application areas include the following items:

- Sequence labelling tasks, such as POS tagging, chunking, named entity recognition. Some of the tasks could benefit from features that could not be expressed in WCCL, e.g. bag-of-word features describing local contexts stretching beyond sentence boundaries.
- The `AnnotationPointer` may be used to iterate over any type of annotations already placed in text. The annotations may also correspond to larger units of text, e.g. previously recognised definitions, clauses in sentences or perhaps selected whole sentences annotated as spots of interest. Fextor could be used to classify such stretches of text with respect to various criteria, e.g. sentiment, question type, dialogue act classification.

5 Conclusions

Feature extraction from text corpora is an important step in Natural Language Processing based on Machine Learning. In this paper we have described a flexible approach to feature extraction, which was implemented in the *Fextor* environment. We have shown architecture of the system and a few example applications: anaphora resolution, classification of derivational relations, recognition of inter-chunk syntactic relations, recognition of semantic relations between named entities and word sense disambiguation.

There are a few possibilities for further development. Fextor was designed for Polish, but after a few extension it should be possible to adapt it to other languages — especially from the Slavic family. We also need to finish the deployment of Fextor in recognition of syntactic and semantic relations. The software is being released under GNU GPL licence; we still need to finish writing the documentation, which at the present stage is rather rudimentary.

References

1. Anderson, E.: The species problem in iris. Annals of the Missouri Botanical Garden **23**(3) (1936) 457–509
2. Radziszewski, A., Wardyński, A., Śniatowski, T.: WCCL: A morpho-syntactic feature toolkit. In: Proceedings of the Balto-Slavonic Natural Language Processing Workshop, Springer (2011)
3. Broda, B., Piasecki, M.: Evaluating LexCSD in a Large Scale Experiment. Control and Cybernetics **40**(2) (2011)
4. Bird, S., Loper, E.: Nltk: The natural language toolkit. In: Proceedings of the ACL demonstration session, Barcelona, Association for Computational Linguistics (2004) 214–217
5. Padró, L., Collado, M., Reese, S., Lloberes, M., Castellón, I.: FreeLing 2.1: Five years of open-source language processing tools. In Chair), N.C.C., Choukri, K., Maegaard, B., Mariani, J., Odijk, J., Piperidis, S., Rosner, M., Tapias, D., eds.: Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10), Valletta, Malta, European Language Resources Association (ELRA) (2010)
6. Roth, D., Cumby, C., Sammons, M., Yih, W.T.: A relational feature extraction language (fex). Technical report, University of Illinois at Urbana Champaign (2004)
7. Radziszewski, A., Śniatowski, T.: Maca — a configurable tool to integrate Polish morphological data. In: Proceedings of the Second International Workshop on Free/Open-Source Rule-Based Machine Translation. (2011)
8. Janus, D., Przepiórkowski, A.: Poliqarp: An open source corpus indexer and search engine with syntactic extensions. In: Proceedings of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions, Association for Computational Linguistics (2007) 85–88
9. Fellbaum, C., et al.: WordNet: An electronic lexical database. MIT press Cambridge, MA (1998)
10. Baś, D., Broda, B., Piasecki, M.: Towards Word Sense Disambiguation of Polish. In: Proceedings of the International Multiconference on Computer Science and Information Technology — 3rd International Symposium Advances in Artificial Intelligence and Applications (AAIA'08). (2008) 65–71

11. Agirre, E., Edmonds, P., eds.: Word Sense Disambiguation: Algorithms and Applications. Springer (2006)
12. Navigli, R.: Word sense disambiguation: A survey. ACM Comput. Surv. **41**(2) (2009) 1–69
13. Ng, T., Lee, H.: Integrating multiple knowledge sources to disambiguate word senses: An examplar-based approach. In: Proceedings of the Thirty-Fourth Annual Meeting of the Association for Computational Linguistics. (1996) 40–47
14. Piasecki, M., Szpakowicz, S., Broda, B.: Extended Similarity Test for the Evaluation of Semantic Similarity Functions. In: Proceedings of the 3rd Language and Technology Conference, October 5–7, 2007, Pozna'n, Poland, Pozna'n, Wydawnictwo Pozna'nskie Sp. z o.o. (2007) 104–108
15. Broda, B., Derwojedowa, M., Piasecki, M., Szpakowicz, S.: Corpus-based Semantic Relatedness for the Construction of Polish WordNet. In (ELRA), E.L.R.A., ed.: Proceedings of the Sixth International Language Resources and Evaluation (LREC'08), Marrakech, Morocco (2008)
16. Młodzki, R., Przepiórkowski, A.: The wsd development environment. In Vetulani, Z., ed.: Proc. 4rd Language and Technology Conference, Poznań, Poland. (2009)
17. Piasecki, M., Szpakowicz, S., Broda, B.: A WordNet from the Ground Up. Oficyna wydawnicza Politechniki Wroclawskiej (2009)
18. Wróblewska, A.: Polish dependency bank. Linguistic Issues in Language Technology **7**(1) (2012)
19. Przepiórkowski, A.: Powierzchniowe przetwarzanie języka polskiego. Akademicka Oficyna Wydawnicza EXIT, Warsaw (2008)
20. Radziszewski, A., Maziarz, M., Wieczorek, J.: Shallow syntactic annotation in the Corpus of Wrocław University of Technology. Cognitive Studies **12** (2012)
21. Broda, B., Marcińczuk, M., Maziarz, M., Radziszewski, A., Wardyński, A.: Kpwr: Towards a free corpus of polish. In Calzolari, N., Choukri, K., Declerck, T., Doğan, M.U., Maegaard, B., Mariani, J., Odijk, J., Piperidis, S., eds.: Proceedings of LREC'12, Istanbul, Turkey, ELRA (2012)
22. Maziarz, M., Radziszewski, A., Wieczorek, J.: Chunking of Polish: guidelines, discussion and experiments with Machine Learning. In: Proceedings of the 5th Language & Technology Conference LTC 2011, Poznań, Poland (2011)
23. Daelemans, W., Zavrel, J., Ko van der Sloot, A.V.d.B.: TiMBL: Tilburg Memory Based Learner, version 6.3, reference guide. Technical Report 10-01, ILK (2010)
24. Fan, R.E., Chang, K.W., Hsieh, C.J., Wang, X.R., Lin, C.J.: LIBLINEAR: A library for large linear classification. Journal of Machine Learning Research **9** (2008) 1871–1874
25. Marcińczuk, M., Janicki, M.: Optimizing CRF-based Model for Proper Name Recognition in Polish Texts. In: Proceedings of the 13th International Conference on Intelligent Text Processing and Computational Linguistics. Volume 7181 of Lecture Notes in Computer Science (LNCS)., Springer-Verlag (2012)
26. Marcińczuk, M., Ptak, M.: Preliminary study on automatic induction of rules for recognition of semantic relations between proper names in polish texts. In: Proceedings of the 15th International Conference on Text, Speech and Dialogue (to appear). Lecture Notes in Artificial Intelligence (LNAI), Springer-Verlag (2012)
27. Bunescu, R.C.: Learning for information extraction: from named entity recognition and disambiguation to relation extraction. Ph.d., The University of Texas at Austin (2007)
28. Soon, W.M., Chung, D., Lim, D.C.Y., Lim, Y., Ng, H.T.: A machine learning approach to coreference resolution of noun phrases (2001)

29. Ng, V., Gardent, C.: Improving machine learning approaches to coreference resolution. In: ACL. (2002) 104–111
30. Piasecki, M., Ramocki, R., Maziarz, M.: Automated Generation of Derivative Relations in the Wordnet Expansion Perspective. In: Proceedings of the 6th Global Wordnet Conference, Matsue, Japan (2012)