Technical Report

for

Version 4.0

(April 23, 2013)

by Alexander Volokh

Contents

1 Overview

MDParser stands for multilingual dependency parser and is a data-driven system, which can be used to parse text of an arbitrary language for which training data is available. The parser is able to create both unlabeled and labeled dependency structures. The number of possible relation types depends on the granularity of the training data.

The models of the system are based on various features, which are extracted from the words of the sentence, including word forms and part of speech tags. Therefore in order to process previously unannotated text MDParser additionally includes some preprocessing components:

- a sentence splitter, since the parser constructs a dependency structure for individual sentences
- a tokenizer, in order to recognise the elements between the dependency relations will be built
- a part of speech tagger, in order to determine the part of speech tags, which are one of the most important influencing factors for constructing the dependency structure.

MDParser is an especially fast system and therefore it is particularly suitable for processing very large amounts of data. Thus it can be used as a part of bigger applications in which dependency structures are desired.

MDParser has already been tested for several languages, including German, English, Finnish and Hindi. It is currently able to achieve quite competitive results, considering that it is based on a fast linear classification approach and a deterministic parsing strategy. A more detailed evaluation and comparison with other systems can be found in my dissertation[1].

2 Preprocessing Components

MDParser's sentence splitter and tokenizer are based on the Java class BreakIterator. The BreakIterator class implements methods for finding the location of boundaries in text for any language, which guarantees MDParser's multilinguality. However, for some languages its performance is not good. E.g. it does not recognise German ordinal numbers like 2. or 3. and treats the point as a sentence boundary. The package DE.DFKI.LT.SENTENCESPLITTER provides support for training your own sentence splitter, which I have done for German. For other languages I use the default BreakIterator functionality.

The part of speech tagger I use in MDParser was developed by my colleague Sven Schmeier and it is based on SVMTool[2] - a generator for sequential taggers. Unfortunately, it is not open source and I am not familiar with the technicalities.

[1] Currently available here http://www.dfki.de/~avolokh/phd.pdf ; will be published later
[2] http://www.lsi.upc.edu/~nlp/SVMTool/

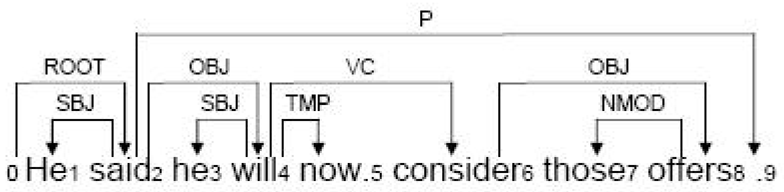

Figure 1: Dependency Structure

However, it is also trainable for any language and thus guarantees MDParser's multilinguality as well. It shows a good performance, but since it is not open source it turned out to have two disadvantages: a) the constructor of the tagger class requires the path to the model as a String. This is problematic if you want to put the models in an archive (e.g. jar or apk), because there you only can work with InputStream and b) it does not support multithreading, which becomes the bottleneck of MDParser if you run it on a machine with many cores.

3 Dependency Structure

In this section I want to show how a dependency analysis looks like and what formal properties I assume. Dependency grammar works only with surface forms of the words in the sentence, without assuming any phrasal units. This is a very special property of dependency grammar, since there is no way to treat particular groupings of words (such as chunks or constituents) in a specific manner. Even though these surface forms do not necessarily have to be identical with the word forms we observe in a sentence, but usually one assumes this kind of a definition of words. The only exception will be the special root node (0), which does not have any surface form. Words are interconnected by special meaningful relations called dependencies and form a structure called dependency structure. Here is an example of dependency structure:

Dependency and dependency structure are central notions for the theory, therefore it is necessary to list the properties I assume for them for my parser. The implementation of the classes is located in the package PARSER and is called DEPENDENCY or DEPENDENCYSTRUCTURE respectively.

3.1 Dependency Relations

Dependency relations are:

- **binary** (only defined for pairs of words)
- **directed / antisymmetric** ($A \rightarrow B \neq B \rightarrow A$; the direction between the words is important, furthermore I assume that for any pair of words

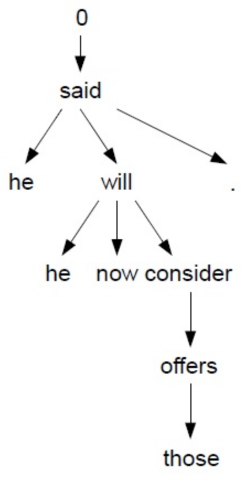

Figure 2: Dependency Structure as a Tree

a relation only in one direction is allowed at the same time)

- **anti-reflexive** (if $A \rightarrow B$, then $A \neq B$; no word can govern itself)
- **anti-transitive** (if $A \rightarrow B$ and $B \rightarrow C$, then $A \nrightarrow B$; dependency relations are only direct)
- **labeled** ($A \xrightarrow{l} B$; the relation between two words is typed with the label l)

Since dependencies are binaries, I will often use the notation used for pairs - (A, B), which is an abbreviation for $A \rightarrow B$. The term I will use for A is *head* or *parent* and B will be called *dependent* or *child*. Even though dependencies are not transitive, it is still sometimes useful to know whether some element is dominated by another indirectly, i.e. in several steps. Therefore I will use the notation $A \xrightarrow{*} B$ and will call B *subordinate* of A.

3.2 Dependency Structures

Here are the properties of the constructed dependency structures:

- **connectedness** (all words should be dominated by at least one other word)
- **single-headiness** (all words, except the special 0 word have to have exactly one head)
- **rooted** (each sentence has exactly one token whose head is 0)

- **acyclicity** (there should be no way that $A \xrightarrow{*} A$, in other words there should be no cycle)
- **projectivity** (if $A \rightarrow B$, then all words between A and B have to be subordinate of A; in terms of the dependency graph as seen in Figure 1, it means that there are no crossing branches)

4 Parsing Algorithm

There are two dominant approaches for dependency parsing. The one general approach is to construct a tree which maximises the probabilities of individual dependency relations within it. It is called graph-based and there are a lot of different strategies based on this idea. Another approach is called transition-based and its idea is to perform a sequence of actions which in the end lead to the correct dependency tree. Both approaches achieve very competitive results and it is impossible to say which one is better. However, the transition-based approach is particularly suitable for fast systems and it does not require much resources, since only a relatively small amount of local information is required in order to choose the locally best transition in each step, whereas for the graph-based approach the whole tree is constructed in one step and a lot of information has to be considered all at once.

MDParser is a transition-based system and it is based on a slightly modified version of Covington's parsing strategy. The module responsible for choosing the best operation in every step is called *oracle*. At first, during the training phase, the perfect oracle is simulated by using the training data, during which the system is driven by the already given gold standard result. During this phase the system learns which operation is likely to be chosen in which situation. These situations depend on the current state of the system and its auxiliary data structures, and are called *configurations*. The result is a model, which is then used to make predictions about the most likely action, when the gold standard result is unavailable, which happens in the parsing phase.

Algorithmus 1 Parsing Algorithm

```
for (int j=1; j < sentence.length;j++) {
  for (int i=j-1; i >= 0; i--) {
    if (permissible(j,i) || permissible (i,j) {
       select correct operation (oracle,j,i)
    }
  }
}
```

The obvious strategy is to examine each pair of words in the entire sentence of length n, linking them as head-dependent or dependent-head if necessary. The Covington's strategy does it in a very intelligent way: according to this algorithm one goes through all words j starting from the first and finishing at the last (line 1). One then scans through all potential candidates left to it. Since the head or dependents of j are more likely to be close to it than far away one works backward from j to $j-1$, $j-2$ and so on in order to find them earlier (line 2). In case $i=0$ the word j can be made the root of the sentence. For each pair (j,i) the correct decision should be predicted. This is done by consulting the oracle (line 4), given the current configuration.

This algorithm turned out to work best with four basic operations:

- LINK1(j, i)
  - Makes the token i the head of the token j
- LINK2(j, i)
  - Makes the token j the head of the token i
- TERMINATE
  - i is set to -1 (i.e. one basically goes over to the next j)
- SHIFT
  - Does nothing(i.e. i is decremented by 1)

In the past after the basic operations, which allow to find head-modifier relations between words, one had to use a different model to predict the relation type (label) for these dependencies. However, the newer machine learning techniques, which allow real multi-class classification enable us to do both operations in one step. Thus the classifier does not learn only when to do LINK1 or LINK2 operations, but it rather learns operations like LINK1#SBJ, LINK1#OBJ, LINK1#NMOD etc.

This algorithm will definitely terminate after $\frac{n(n-1)}{2}$ steps in the worst-case, which is already very good. However, the line 3 checks the pair of words (j, i) for permissibility, i.e. whether a dependency relation between them does not violate the dependency structure constraints introduced earlier. In many cases this a pair of words is not permissible and therefore can be skipped immediately without consulting the oracle.

The implementation of Covington's algorithm can be found in the DE.DFKI.LT.MDPARSER.ALGORITHM.COVI class. Originally, I have experimented with other algorithms as well, which explains some other parsing algorithm classes, which are still availale, however, Covington's algorithm turned out to be especially effective for fast parsing (cf. [3]) and that is why only Covington's algorithm should be used.

4.1 Permissibility

The newly introduced *permissible* function guarantees, that the current configuration will not violate the dependency structure well-formedness, if the oracle predicts an operation LINK1 or LINK2.

The function receives the values for j and i, as well as the partially constructed dependency structure. It then checks whether LINK1, LINK2 or both operations are allowed, i.e. will not violate the definition of a proper dependency tree in the current step. Therefore the well-formedness should be checked. It is

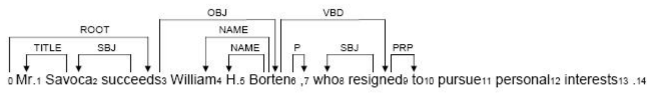
OBJ
VBD
ROOT
NAME
TITLE
SBJ
NAME
P
SBJ
PRP
0 Mr.1 Savoca2 succeeds3 William4 H.5 Borten6 ,7 who8 resigned9 to10 pursue11 personal12 interests13 .14

Figure 3: Partially Constructed Dependency Structure

a trivial task to check for most of the properties, like whether the dependency relation is anti-reflexive (check $i \neq j$) or anti-symmetric (if we check for (i, j), then (j, i) should not be contained in the already constructed tree). It is also very easy to check, whether (i, j) introduces a cycle, by checking recursively all words dominated by i (if i itself is among them, then the new structure is cyclic). For the root condition one should keep track whether a dependency $(0, j)$ has already been found and prohibit all further ROOT operations.

One of the non-trivial constraints is that of projectivity. In order to illustrate the problem we should imagine that we are parsing a sentence as shown in Figure 2.

If we are currently at the eleventh token ($j = 11$), then only relations between 11 and the tokens 10, 9, 6, 3 and 0 would not introduce crossing branches. Whereas it is very easy to understand this idea, it is a bit harder to implement it (at least not as easy as the other constraints). My algorithm for checking whether a dependency relation between j and i introduces non-projectivity into the tree looks like this:

Algorithmus 2 Projectivity Check

```
if (distance(j,i) == 1) {
  return true;
} else {
  int smaller = j;
  int bigger = i;
  if (j > i) {
    smaller = i;
    bigger = j;
  }
  for (int i=bigger-1; i > 0; i–) {
    if (i != smaller) {
      if (i > smaller && head(i) < smaller) {
        return false;
      } else if (i < smaller && head(i) > smaller) {
          return false;
      }
    }
  }
}
return true;
```

This algorithm checks whether the edge between two words with indexes j and i is projective or not. An edge of length 1 is always projective (lines 1-3). Otherwise one has to check which token has a smaller index (lines 3-9). Let us assume i is the smaller and j is the bigger one. Then for every word between i and j (line 10), there should be no word which has a parent which is smaller than i (lines 12-14) and for every word between 0 and i there should be no word which has a head which is bigger than i (lines 14-16).

It is important to note that in some languages non-projective relations are quite common. In this case one can simply exclude the projectivity check from the permissibility function and the system will also be able to capture such relations. The current models, however, assume projectivity.

The other more complex check is that of acyclicity:

Algorithmus 3 Acyclicity check

```
1  int curHead = j;
2  boolean[] possibleEnds = new boolean[sentence.length+1];
3  possibleEnds[i] = true;
4  Stack<Integer> toCheck = new Stack<Integer>();
5  toCheck.add(curHead);
6  while (!toCheck.isEmpty()) {
7    curHead = toCheck.pop();
8    int curHeadHead = heads[curHead];
9    if (curHeadHead == 0) {
10     return true;
11   }
12  if (possibleEnds[curHeadHead]) {
13    return false;
14  } else {
15      toCheck.add(curHeadHead);
16      possibleEnds[curHead] = true;
17   }
18 }
19 return true;
```

This algorithm goes through all tokens, for which holds that i is their subordinate, i.e. i is reachable in the tree by following down the dependency edges. It starts with the current head of i (line 5) and then its head (line 15) etc. until it reaches the artificial node 0 (lines 9-11). If any token comes on the stack for the second time, we know there is a cycle and terminate (lines 12-14). Otherwise if the token 0 can be reached, then we know that there is no cycle on the path.

Depending on the of a constraint, i.e. whether it describes a property of a single dependency relation or of a whole dependency structure the implementation of the permissibility function can be found in the DE.DFKI.LT.MDPARSER.ALGORITHM.DEPENDENCY or DE.DFKI.LT.MDPARSER.ALGORITHM.DEPENDENCYSTRUCTURE classes.

5 Learning

In this section I am going to introduce the machine learning approach I am using for learning my models. I use a strategy called *MCSVM_ CS (SolverType.MCSVM_ CS)[2]* from the LibLinear library[1]. The values of the parameters C and *epsilon* are 0.1 and 0.3, respectively.

The class de.dfki.lt.mdparser.parser.Trainer contains the functionality for creating the training data and learning the model out of it.

First of all it should be said that the training data for the parser is usually so big for most languages that you can not train one model for the whole data because of the hardware limitations. Therefore it is sometimes sensible to reduce the size of the problem or to split it into many smaller ones.The splitting is done in a such way that the training data is split according to the value of some feature, e.g. POS tag. This way one gets one model for verbs, one model for nouns, one model for adjectives etc. Splitting requires one parameter: the minimum size of one piece of training data. This parameter is necessary, because some feature values are so rare that it is not possible to train an accurate model for them. E.g. some POS tags like LS (List item marker), RP (Particle) or EX (Existential *there*) are not frequent enough and should not get a separate model, but should rather be combined in one.

There are two approaches to create the training data: the first one is to first print it to the disk and then read in the files and train models one by one. The other one is to keep the training data in memory and train models without I/O operations. It is clear that the second option is faster, however, because of the size of the training data it often requires too much memory. Especially, since MDParser is able to use several cores in parallel to train several models at the same time, it is better to not have unncessary training data in memory to enable this functionality. Thus despite the fact that the I/O operations take additional time, it is worth it, because the following learning phase becomes much more efficient.

6 Feature Model

The performance of a transition-based parser relies heavily on its ability to predict correct transitions. In this ssection I am going to list all features I have used in MDParser for this task. These features are only based on word forms, POS tags and dependency labels, because this information is available in any dependency treebank for any language. Some languages contain additional information about lemmas or different morphological knowledge, but I did not exploit it, since it is not available for every language and it also would require additional preprocessing components, which are able to deliver this fine-grained information, which are also not easy to get.

1. wfj $\Rightarrow$ returns the word form of the token j.

2. pj $\Rightarrow$ returns the part of speech of the token j.

3. WFJP1 $\Rightarrow$ returns the word form of the token $j+1$.
4. PJP1 $\Rightarrow$ returns the part of speech of the token $j+1$.
5. WFJP2 $\Rightarrow$ returns the word form of the token $j+2$.
6. PJP2 $\Rightarrow$ returns the part of speech of the token $j+2$.
7. WFJP3 $\Rightarrow$ returns the word form of the token$j+3$.
8. PJP3 $\Rightarrow$ returns the part of speech of the token $j+3$.
9. WFI $\Rightarrow$ returns the word form of the token i.
10. PI $\Rightarrow$ returns the part of speech of the token i.
11. PIP1 $\Rightarrow$ returns the part of speech of the token $i+1$.
12. WFHI $\Rightarrow$ returns the word form of the head of the token i.
13. PHI $\Rightarrow$ returns the part of speech of the head of the token i.
14. DEPI $\Rightarrow$ returns the dependency label of the head of the token i.
15. DEPLDI $\Rightarrow$ returns the dependency label of the left-most dependent of the token i.
16. DEPRDI $\Rightarrow$ returns the dependency label of the right-most dependent of the token i.
17. DEPLDJ $\Rightarrow$ returns the dependency label of the left-most dependent of the token j.
18. DIST $\Rightarrow$ returns the distance between the tokens j and i. For $i = 0$ the feature returns 0, for the distance 1 the feature returns 1, for distances 2 or 3 the feature returns 2, for distances 4 or 5 the value 3 is returned, for distances 6, 7, 8 or 9 the value 4 and for all other distances the value 5 is returned.
19. MERGE2(PI,PIP1)$\Rightarrow$ returns the concatenation of PI and PIP1 features.
20. MERGE2(WFI,PI)$\Rightarrow$ returns the concatenation of WFI and PI features.
21. MERGE3(PJP1,PJP2,PJP3)$\Rightarrow$ returns the concatenation of PJP1, PJP2 and PJP3 features.
22. MERGE2(DEPLDJ,PJ)$\Rightarrow$ returns the concatenation of DEPLDJ and PJ features.
23. MERGE3(PI,DEPRDI,DEPLDI)$\Rightarrow$ returns the concatenation of PI, DEPRDI and DEPLDI features.
24. MERGE2(DEPI,WFHI)$\Rightarrow$ returns the concatenation of DEPI and WFHI features.

25. MERGE3(PHI,PJP1,PIP1)⇒ returns the concatenation of PHI, PJP1 and PIP1 features.

26. MERGE3(WFJ,WFI,PJP3)⇒ returns the concatenation of WFJ, WFI and PJP3 features.

27. MERGE3(DIST,PJ,WFJP1)⇒ returns the concatenation of DIST, PJ and PJP1 features.

Feature templates 1-13 are lexical (word forms and POS tags), 14-17 are based on dependency labels and template 18 takes the distance between two words into account. Templates 19-27 combine the templates 1-18 between each other.

This is a model with a lookahead of 3, which uses a lot of static features templates (all word form templates and all POS templates are static by default). Most of the features are suitable for memoisation, because they have the same values not only for one, but for several configurations. The implementation of the feature model can be found in the DE.DFKI.LT.MDPARSER.FEATURES.COVINGTONFEATUREMODEL class.

7 Models and Alphabets

In this section I will briefly describe how the result of the training procedure looks like and how it is used during the application. The training result consists of two components: model and alphabet, the DE.BWALDVOGEL.LIBLINEAR.MODEL class is part of the liblinear library, whereas alphabet is implemented in the DE.DFKI.LT.MDPARSER.FEATURES.ALPHABET class.

Model: The model file contains the following important information. Line 1 contains the name of the learning algorithm, in our case it is MCSVM_CS. Line 2 contains the number of different classes for which the weights were learned. Line 3 lists the indexes of the classes. Line 4 contains the number of features for which at least one non-zero weight is present. It is important to note that the original models trained by Liblinear contain all features, independently of their weights. That means that even if a feature does not have any significance, because all the weights are zero, it is still part of the model. This is suboptimal, since loading such models requires more memory and it does not affect the quality of classification. Therefore the models of MDParser are postprocessed and compactised by removing all such features. The functionality for that is provided by the DE.DFKI.LT.MDPARSER.MODEL.MODELEDITOR class. Line 5 contains the information about the chosen bias, which is the constant added to each feature vector. In MDParser it is always set to -1, which means that the bias is not used. From line 7 onwards the model contains the weights for the features. Every feature has exactly n weights, where n is the number of different classes, as specified in the line 2.

Alphabet: In order to access the weight of a feature in the model one has to know its index, so one can look up the weight from the corresponding line in the model file. Therefore a mapping between string values of features and unique integers is necessary, e.g. 7457 WFJ=NEW or 35788 wfjp1=reviving. Additionally, the correspondence between class indexes and operation names has to be stored, e.g. 1 shift, 2 terminate, 3 i#NMOD, 4 j#PMOD, 5 j#NMOD, 6 j#P etc. The already mentioned ModelEditor class also removes the entries for features, which have zero weights in the model and then adjusts the other indexes (e.g. if a feature is removed, all other features with a higher index gets an index which is 1 smaller), because it is beneficial to have lower numbers when creating arrays for features.

8 Usage

In this section I will describe the requirements for running the system and how it is used.

MDParser is implemented in Java and thus Java JRE 1.6 is the foremost requirement.

MDParser can be run:

- as a jar file the command line (java -Xmx1000m -jar mdpfull.jar mdpfull.xml). This corresponds to DE.DFKI.LT.MDPARSER.TEST.MDPARSER main class.
- from your own java project by adding the jar to the build path and addressing the methods of the DE.DFKI.LT.MDPARSER.TEST.MDPARSER class (e.g. parseSentence(String text, String language, String inputFormat) or parseText(String text, String language, String inputFormat)) text = text string to be parsed; language = {english,german} inputFormat = {text,conll} (text = string has to be preprocessed (sentence splitting, tokenisation, pos tagging), conll = preprocessing is done, only dependency parsing is necessary
- as an Android app for demonstration purposes (the server for German runs on http://lns-87009.sb.dfki.de:3389/xmlrpc and for English on http://lns-87009.sb.dfki.de:3379/xmlrpc), from time to time when the lns-87009 machine is restarted one has to restart these servers (DE.DFKI.LT.MDPARSER.XMLRPC.MDPSERVER class)
- using server/client architecture (DE.DFKI.LT.MDPARSER.XMLRPC.MDPSERVER and DE.DFKI.LT.MDPARSER.XMLRPC.MDPCLIENT classes). The server basically loads the model, which is a time consuming step (can take 10-20 seconds) and keeps it in memory. The client can then access these models whenever necessary without having to wait for models to be loaded every time.

MDParser has a lot of parameters. In order not to change them from code and also allow flexible usage with JAR files, most of these parameters are loaded from properties file (Java Properties class). Here are the examples of all properties files used for MDParser.

8.1 MDPServer

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE properties SYSTEM
"http://java.sun.com/dtd/properties.dtd"> <properties version="1.0">
<entry key="port">3379</entry>
<entry key="language">english</entry>
<entry key="algorithm">covington</entry>
<entry key="modelsFile">englishConll.zip</entry>
<entry key="modelFilePOSTaggerEnglish">resources/BROWN_MEDLINE/MEDLINE-
BROWN-FINAL</entry>
<entry key="modelFilePOSTaggerGerman">resources/NEGRA/NEGRA</entry>
</properties>
```

8.2 MDPClient

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE properties SYSTEM
"http://java.sun.com/dtd/properties.dtd"> <properties version="1.0">
<entry key="server">http://lns-87009.sb.dfki.de:3379/xmlrpc</entry>
<!– server's address (including port) lns-87009.sb.dfki.de–>
<entry key="inputType">file</entry>
<!– file - 'inputFile' is only one file –>
<!– dir - 'inputFile' is a whole directory with files to be parsed –>
<entry key="inputFile">input/ambig.txt</entry>
<!– path to the input file if inputType is 'file'–>
<!– path to the input dir if inputType is 'dir'–>
<entry key="language">german</entry> <!– language of the input file –>
<!– currently supported: english or german –>
<entry key="taggedFile">temp/1.txt</entry>
<!– path to the tokenized and POS-tagged file –>
<!– if inputType is 'file' the property should be a file –>
<!– if inputType is 'dir' the property should be a dir –>
<!– used only if mode is tag or outputFormat is 'stanford' –>
<entry key="mode">parse</entry>
<!– tag = only tokenize and tag with POS the input –>
<!– parse = tokenize, tag with POS and parse the input –>
<entry key="outputFile">temp/1.conll</entry>
<!– path to the output file if inputType=file and path to the output directory
if inputType=dir –>
<entry key="inputFormat">text</entry>
<!– conll = input file is in CONLL format –>
<!– text = input file is plain text –>
```

```
<entry key="outputFormat">conll</entry>
<!– conll = output file is in CONLL format –>
<!– conllxml = output file is an xml file in CONLL format –>
<!– triple = output file is in triple format –>
<!– stanford = two output files(taggedFile(lemma/pos)and outputFile(dependencies))
are in STANFORD format –>
<entry key="numberOfThreads">1</entry>
<!– currently not used, but set in the code–>
</properties>
```

8.3 MDParser

```
<?xml version="1.0" encoding="UTF-8"?> <!DOCTYPE properties SYSTEM
"http://java.sun.com/dtd/properties.dtd"> <properties version="1.0">
<entry key="inputType">file</entry>
<!– file - ’inputFile’ is only one file (required for the train mode) –>
<!– dir - ’inputFile’ is a whole directory with files to be parsed –>
<entry key="inputFile">input/g4.txt</entry>
<!– path to the input file if inputType is ’file’–>
<!– path to the input dir if inputType is ’dir’–>
<entry key="language">german</entry>
<!– language of the input file –>
<!– currently supported: english or german –>
<entry key="taggedFile">temp/1.txt</entry>
<!– path to the tokenized and POS-tagged file –>
<!– if inputType is ’file’ the property should be a file –>
<!– if inputType is ’dir’ the property should be a dir –>
<!– used only if mode is tag or outputFormat is ’stanford’ –>
<entry key="mode">parse</entry>
<!– tag = only tokenize and tag with POS the input –>
<!– parse = tokenize, tag with POS and parse the input –>
<!– train = train a new model –>
<entry key="outputFile">temp/1.conll</entry>
<!– path to the output file if inputType=file and path to the output directory
if inputType=dir –>
<entry key="inputFormat">text</entry>
<!– conll = input file is in CONLL format (required for the train mode) –>
<!– text = input file is plain text –>
<entry key="outputFormat">conll</entry>
<!– conll = output file is in CONLL format –>
<!– conllxml = output file is an xml file in CONLL format –>
<!– triple = output file is in triple format –>
<!– stanford = two output files(taggedFile(lemma/pos)and outputFile(dependencies))
are in STANFORD format –>
<entry key="modelsFile">germanConll.zip</entry>
<!– file to which the parser models are saved (must be a zip-file) –>
```

```
<entry key="modelFilePOSTaggerEnglish">resources/BROWN_MEDLINE/MEDLINE-
BROWN-FINAL</entry>
<!– path to the POS tagger model for English–>
<entry key="modelFilePOSTaggerGerman">resources/NEGRA/NEGRA</entry>
<!– path to the POS tagger for German –>
</properties>
```

References

[1] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9:1871–1874, 2008.

[2] S. Sathiya Keerthi, S. Sundararajan, Kai-Wei Chang, Cho-Jui Hsieh, and Chih-Jen Lin. A sequential dual method for large scale multi-class linear SVMs. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '08, pages 408–416, New York, NY, USA, 2008. ACM.

[3] Alexander Volokh and Günter Neumann. Dependency parsing with efficient feature extraction. In *KI*, pages 253–256, 2012.