Automatic Analysis of Flaws in Pre-Trained NLP Models

Richard Eckart de Castilho
Ubiquitous Knowledge Processing Lab (UKP-TUDA)
Department of Computer Science
Technische Universität Darmstadt
`http://www.ukp.tu-darmstadt.de/`

Abstract

Most tools for natural language processing (NLP) today are based on machine learning and come with pre-trained models. In addition, third-parties provide pre-trained models for popular NLP tools. The predictive power and accuracy of these tools depends on the quality of these models. Downstream researchers often base their results on pre-trained models instead of training their own. Consequently, pre-trained models are an essential resource to our community. However, to be best of our knowledge, no systematic study of pre-trained models has been conducted so far. This paper reports on the analysis of 274 pre-models for six NLP tools and four potential causes of problems: *encoding*, *tokenization*, *normalization*, and *change* over time. The analysis is implemented in the open source tool *Model Investigator*. Our work 1) allows model consumers to better assess whether a model is suitable for their task, 2) enables tool and model creators to sanity-check their models before distributing them, and 3) enables improvements in tool interoperability by performing automatic adjustments of normalization or other pre-processing based on the models used.

1 Introduction

As natural language processing (NLP) has become a pervasive technology in many research and application domains, NLP tools and the pre-trained models that ship with them or that are provided by third parties have become essential resources. Often, researchers and other users do not have access to text corpora from which they could train suitable models themselves or simply prefer the convenience of using existing models. Users of centrally provided NLP infrastructures may not even have the option of using models other that those offered within the platform. However, this entails that any problems with these models, propagate into subsequent NLP components in a pipeline or into data analytics and negatively influence them. Worst of all, the consumers of pre-trained models may not even be aware of the problems the models exhibit.

We hope to raise more awareness through this paper by reporting on an analysis of pre-trained models for typical sources of problems: *encoding*, *tokenization*, *normalization*, and *changes* between different released versions (Section 3). The analysis is based on a dataset of 274 pre-trained models for six popular NLP tools. To the best of our knowledge, this is the first systematic analysis of pre-trained models for flaws and other sources of problems. We have observed that some models have been distributed with serious flaws for years and have informed the model creators of the problems.

The open source tool *Model Investigator*[1] that was created and used to perform this analysis (Section 2). It can be used by model creators to perform automatic sanity checks on their models and by model consumers to obtain detailed information about new models as they become available. The modular tool can easily be extended to support pre-trained models of new NLP tools.

Finally, we investigate the negative impact of flawed models in a case study: models trained with a bad character encoding (Section 4).

[1] `https://github.com/UKPLab/coling2016-modelinspector`

1.1 The role of models in NLP pipelines

Before going into the details, we briefly motivate why the analyzed issues are typical sources of problems in NLP pipelines, in particular when mixing components and models from different vendors.

Combining multiple NLP tools into pipelines is a typical task that is hindered by the fact that the tools tend to come with different APIs and input/output formats. There is a range of frameworks and platforms such as UIMA (Ferrucci and Lally, 2004), GATE (Cunningham et al., 2011), WebLicht (Hinrichs et al., 2010), LAP (Lapponi et al., 2013), DKPro Core (Eckart de Castilho and Gurevych, 2014) and LAPPS Grid (Ide et al., 2014) that address this problem by each respectively providing common data exchange models between NLP tools. Tool wrappers ensure that the data is transformed between the common exchange model and the tool-specific input/output formats, e.g. between the UIMA CAS exchange model and the input/output format used by the Stanford Parser. This process of wrapping NLP tools coming from entirely different contexts requires paying close attention to all kinds of variations in data representation, tagsets being used, etc. This requires not at last having detailed information about models and their characteristics.

A tool wrapper needs to know how to transform the data, e.g. which kinds normalization to apply to the text before passing it on to the tool, which character encoding to use, etc. This information can be hard-coded into the wrapper, but this is inflexible. It can be provided via parameters that a user needs to supply, but this is error prone. It could also be automatically obtained directly from the model file the tool is configured to use, but models are typically treated as black boxes. Extracting the necessary information is often not possible via the standard API. In many cases, we had to use invasive methods to extract e.g. lexicon information directly from *private variables*[2] in the model data objects.

Also, the interaction between components in a pipeline can have an impact. For example tokenization is typically the initial step in a pipeline and the resulting tokens are used by the other components. If the pipeline tokenization scheme does not correspond to the one that was used when a model was created, a negative impact on the pipeline results can be expected.

Some NLP tools have been around for many years and have seen multiple updates to their code and to their models. The models may have changed in significant, yet typically undocumented ways, e.g. with respect to normalization or lexicon size, which can have an impact on existing pipelines being upgraded to newer model versions. One might expect that there have been improvements that would produce better results when upgrading an old experimental setup to using a newer version of a tool or model. A positive change could be that newer models have been trained on more data and offer a larger vocabulary coverage. However, changes may also easily lead to decreased performance, e.g. if the normalization steps used when creating the model were changed and the pipeline setup is not adjusted accordingly.

2 The *Model Investigator* tool

To perform our analysis, we have implemented the open-source software *Model Investigator*. For the analysis, *Model Investigator* relies on extracting the lexicon information encoded in the models, i.e. the tokens the model was trained on. The lexicon is not always explicitly represented in a model. In some cases, it has to be extracted from different types of features representations according to the machine learning approach used by the respective tool.

Model Investigator can be used either as a command line tool or as a software library. Its architecture consists of three main component types: *model analyzers*, *checks*, and *reports*.

Model Analyzers A model analyzer extracts the lexicon information from a given model file. To support a new NLP tool, a new model analyzer needs to be implemented. The method of extracting the lexicon is specific for each tool. While some models contain an explicit lexicon, for others it needs to be extracted from an inventory of feature names that are used by the machine learning part of the tool.

Typically, the model analyzer loads a model into memory using the deserialization provided by the respective tool. As most of the analyzed tools have been implemented for the Java Virtual Machine (JVM),

[2]Many programming languages allow to control the visibility of data and functionality. The *private* functions and variables are meant not to be used by third parties, usually because they may be subject to change between different version of a software package without further notice. Without special effort, such functions and variables cannot be accessed.

we used the JVM-based language Groovy to implement our tool. Groovy has the benefit being able to ignore the typical Java access-modifiers (*private*, *protected*) and greatly facilitates accessing the often access-restricted data structures containing the lexicon information. Through integration with the Maven Central software repository, we could make it easy to automatically test a given model against different versions of a NLP tool, e.g. to test with which versions of the Stanford Parser and CoreNLP packages a given model is compatible. This way, we could inspect the models of all of the analyzed tools except TreeTagger, which is not implemented in Java. To analyze the TreeTagger models, we had to reverse-engineer the undocumented format of the binary model files to extract the lexicon information.

Checks The checks are then used to test the extracted lexicon for potential problems. A range of checks related to *encoding*, *tokenization*, and *normalization* have been implemented (cf. Sections 3.1-3.3). New checks can be added easily. The checks performed by the tool fall into two categories: *boolean checks* (e.g. if escaped and non-escaped brackets appear at the same time) and *frequency checks* (e.g. how many lexicon words contain encoding problems). As most of the models contain a small number of problems, we report only those frequency check results that affect more than 1% of the lexicon entries. Not all models are versioned and not all models change across releases of a tool. Thus, as the model version we use the date of the first change detected by calculating and comparing a checksum over the different model files in a series. If exactly the same model is contained in multiple versions of a NLP software package, we count it only once.

Reports The primary output of *Model Investigator* are detailed per-model analysis reports that are written as machine-processable JSON files. In addition, *Model Investigator* also generates comparative diachronic reports for models which are available in multiple versions. We leveraged this to analyze the ways in which models in a series *change* from one release to the next (cf. Section 3.4). Overview reports are generated as CSV files which can be opened using most spreadsheet applications.

3 Model Analysis

This section examines different types of issues related to encoding, tokenization, and normalization that were observed in the analyzed models. We exclude questions related to coverage because typically a frequency cutoff is applied during the training process and thus the lexicon extracted from a model usually does not give a complete account of the tokens in the training corpus.

As the basis for our analysis, we have compiled a dataset[3] of 274 pre-trained models for six NLP tools: POS tagger and named entity recognizer from Apache OpenNLP[4]; POS tagger, parser, and named entity recognizer from Stanford CoreNLP (Manning et al., 2014); TreeTagger (Schmid, 1994) (Table 1).

In order to track changes over time, we have organized the 274 models into *series*. Each series represents what should be considered as a set of different versions of essentially the same model. Such a series would e.g. cover the English PCFG models from the Stanford Parser that were included with different releases of the Stanford Parser package and the Stanford CoreNLP package. We have analyzed 48 series of size > 1.

3.1 Character Encoding

Problem description Special care has to be taken when creating a model with respect to 1) the encoding of the corpora used as training data and 2) with respect to the default encoding used on the platform where the model is created. Older (western) corpora often use variations of the ISO 8859 encoding, whereas newer corpora tend to use a Unicode encoding, often UTF-8. As for the platforms that models are trained on: the default (western) encoding for Windows is ISO 8859; UTF-8 is commonly the default on Linux and OS X (depending on the version, also Mac OS Roman). This heterogeneity of encodings makes training a model an error-prone task that requires careful attention. We observe two typical problems related to encoding:

[3]Some TreeTagger models that were collected in the past are no longer publicly available from the TreeTagger homepage. Since the license of the TreeTagger models does not allow for redistribution, we can unfortunately not make them available.

[4]`http://opennlp.apache.org`

Tool ID	Product	Tool	Languages	Series	Models
C-TAG	CoreNLP	POS tagger	6	23	70
O-TAG	OpenNLP	POS tagger	8	22	22
T-TAG	TreeTagger	POS tagger	18	18	23
C-NER	CoreNLP	Named Entity Recognizer	3	15	33
O-NER	OpenNLP	Named Entity Recognizer	3	15	15
C-PAR	CoreNLP	Parser	6	25	111
Total			21	118	274

Table 1: Analyzed tools, series, and models

Type A) Non-UTF-8 data is read as UTF-8. Non-ASCII characters that are not valid in UTF-8 are replaced with the Unicode replacement character (U+FFFD). This is a destructive operation as the original character information is lost. There are variation of this problem that use other characters as a replacement, e.g. the question mark. We did not analyze these variations here.

Type B) UTF-8 is read as ISO 8859 and re-encoded into UTF-8. In this case, UTF-8 characters are treated as one or more ISO 8859 characters, depending on the number of bytes used to encode the character in UTF-8. E.g. ”á” (UTF-8) becomes ”Ã¡” (ISO 8859-1). This is a non-destructive operation as the ISO 8859 byte sequence can be recovered and then re-interpreted as UTF-8.

Implementation The checks for type A flaws count the numbers of entries in the model lexicon containing question marks and Unicode replacement characters respectively. The check for a type B flaw is more sophisticated. Here, a set of *seed characters* is generated from those characters which differ between the different ISO 8859 variants, i.e. non-ASCII characters in the code range from 0xA1 to 0xFF. These seed characters are converted from the source encoding to a UTF-8 sequence which is then again re-encoded using the source encoding. The check then counts the number of model lexicon entries containing these re-encoded seed characters.

Result We observe serious type A flaws in 5 models of 3 series. For type B flaws, we checked against ISO 8859-(1,2,5,6) and used only the results for the encoding with the most hits. 7 models of 4 series exhibit serious type B encoding problems likely due to misconfiguration during model training. In both cases, we report only analysis results on the latest model version exhibiting the problem in the respective series (Table 2), which is also the latest version available at the time of writing. Additionally, we list first version affected version in each series from our dataset. It can be seen that the flaws in the models exist for many years and have even be carried over to newer versions. A small number (2-50) of type A and B errors was observed in 12 additional models and are likely spurious mistakes in the underlying corpora.

The problems listed in Table 2 have been reported to the respective model creators. Some have acknowledged the problems and started looking into providing new fixed models.

			First	Most		Affected	
			affected	recent		lexicon entries*	
Tool ID	Lang	Model	version	version	Type	Rel.	Abs.
C-NER	de	hgc	20120522	20150130	A	16.37 %	2616
C-NER	de	dewac	20120522	20150130	A	16.47 %	2800
C-PAR	es	pcfg	20140826	20150108	B	18.82 %	6492
C-PAR	es	sr	20140828	20141023	B	19.76 %	4495
C-PAR	es	sr-beam	20140828	20141023	B	19.51 %	5654
O-TAG	de	maxent	20120616	20120616	B	16.70 %	2442

Table 2: Models with encoding problems (* affected entires for most recent version).

Recommendation Developers of NLP tools should allow configuring the encoding independent of the platform a model is created on. Model creators must pay attention to configure the tools for the proper encoding of the source data. *Model Investigator* can be used to verify the encoding of the trained model.

3.2 Tokenization

Problem description Tokenization is typically the first step in an NLP pipeline and most subsequent steps build on the generated units. This is true when training models, but also when applying them. Thus, it is important that the same tokenization scheme is used at both stages. However, models are often trained on manually segmented corpora which may be hard to reproduce automatically (cf. (Dridan and Oepen, 2012)). Another common case is that the model creator did not document the tokenization scheme and the unaware model consumer chooses to uses a different one. This leads to the question whether information about this scheme can be extracted from a model and can be used to advise the model consumer whether a particular given automatic segmentation tool is suitable in combination with this model. As a subsequent step, the model consumer may be informed whether different models used in a pipeline use similar or radically different segmentation schemes.

Implementation As a heuristic for the suitability of a tokenizer for a given model, we apply the tokenizer to each lexicon word in turn and count cases in which the tokenizer further splits the lexicon word. For instance, if the lexicon word is [*1-million-plus*] and a given tokenizer splits it into [*1*, -, *million*, -, *plus*], this is counted. In addition to tokenizers from already introduced tools, we used the Java BreakIterator, JTok,[5] LanguageTool,[6] and LingPipe.[7] Due to the modular structure of *Model Investigator*, support for additional tokenizers can be easily added.

Result Exemplary results for all English models (max. over all versions/series) are shown in Table 3. A high result should indicate that a tokenizer is *not* suitable. However, further interpretation of the results is difficult. Obviously the heuristic has two major problems: 1) a trivial tokenizer that does nothing at all produces optimal results; 2) most tokenizers are context sensitive but in our test each lexicon entry is analyzed individually without context. Further in-depth analysis correlating results to different token shapes (e.g. numbers, dates, chemical compounds, etc.) may yield additional insight as to whether a user should be concerned about a high split rate or not.

Tokenizer	O-NER	O-TAG	C-NER	C-PAR	C-TAG	T-TAG
Java BreakIterator	0.05	4.94	5.17	7.72	61.32	0.24
JTok	0.03	2.88	1.14	1.44	49.68	0.08
LanguageTool	0.05	7.71	8.37	10.66	14.72	0.26
LingPipe	0.01	15.55	10.34	18.77	64.08	10.44
OpenNLP	0.02	0.74	1.13	0.77	17.59	0.11
CoreNLP	0.02	3.00	0.99	2.06	3.43	0.15

Table 3: Percentage of split lexicon entries for English pre-trained models of six NLP tools

Recommendation Model consumers should consider using *Model Investigator* to extract the lexicon from a pre-trained model and to apply their tokenizer of choice to it. If the tokenizers splits a significant number of the lexicon entries into smaller tokens, the tokenizer may not be suitable for use with the given model. Based on our analysis of pre-trained models and tokenizers, we conjecture a ratio of more than 20% split lexicon entries as unnecessarily high and potentially problematic.

3.3 Normalization

Problem description Normalization is a common step while preparing corpora and training models on them. The most common and prominent normalization may be the that of brackets and quotes introduced

[5]`http://github.com/DFKI-MLT/JTok`
[6]`http://languagetool.org`
[7]`http://alias-i.com/lingpipe`

by the Penn Treebank (PTB) (Marcus et al., 1993) and used in many corpora thereafter. Here, e.g. a right round bracket " (" gets normalized to "-RRB-". But we also observe other kinds of normalization in the analyzed models, e.g. the escaping of slashes and underscores. Many models contain a small number of XML entities (1-5), but we did not find this to be a systematically applied step. CoreNLP includes a normalization step to *americanize* English text which we did not further analyze here.

In most cases, normalization can be easily detected from the model lexicon. However, to the best of our knowledge, no NLP tool or wrapper actually does inspect the user-provided model to automatically configure suitable normalization steps for the given model. Instead, it is assumed that the user knows which normalization is required and configures the pipeline accordingly. In particular, if different models for the same tool use different normalization schemes, the user may forget to adjust the settings accordingly when changing models.

If normalization is applied inconsistently during model training, it can negatively affect the results. For example, finding PTB-normalized and non-normalized brackets in a model lexicon may be a indicator of an inconsistently normalized corpus or of a bug in the normalization code.

A particular problem are *invisible* characters such as the non-breaking space (NBSP) and the zero-width soft hyphen (SHY). A visual scan of the corpus data does not reveal these. Normally, it cannot be assumed that such characters are used systematically in input data. For most users, models trained on corpora containing such characters will produce sub-optimal results.

Implementation The checks for normalization problems count the number of model lexicon entries containing particular substrings or characters, e.g. entries containing SHY, NPSP, escaped slashes, or PTB brackets and quotes respectively.

Result We analyzed the models looking for typical normalization steps (Table 4) and problem indicators (Table 5). Inconsistent normalization of brackets appears to be a problem in some models, mostly for CoreNLP NER. Mixed quotes are detected in a large number of models. We assume this to be an artifact of the tokenization scheme not splitting adjacent quotes into separate tokens rather than being a normalization problem. SHYs appear systematically in the Spanish CoreNLP models which were all trained on AnCora 3.0 (Taulé et al., 2008) and also in the Spanish OpenNLP NER models. The latter were trained on data from the Spanish EFE News Agency used in CoNLL-2002 which was later integrated into AnCora. NBSPs appear in a many models but not systematically.

Normalization		Series	Models
PTB curly brackets	`-LCB-`, `-RCB-`	27	82
PTB round brackets	`-LRB-`, `-RRB-`	20	119
PTB square brackets	`-LSB-`, `-RSB-`	10	26
PTB quotes	` `` `, `''`	43	103
Slash escaped	\/	2	6

Table 4: Common normalizations

Issue	Series	Models	Max. affected lexicon entries
Mixed round brackets	7	10	-
Mixed square brackets	2	4	-
Mixed quotes	42	102	-
Contains SHY	7	10	4.40 %
Contains NBSP*	22	58	0.11 %

Table 5: Normalization issues (* no 1% cutoff)

Recommendation NBSPs should be replaced by regular a space character and SHYs should be removed from texts before training and analysis, actually even before tokenization. *Model Investigator* includes checks to test whether a trained model still contains problematic invisible characters in its lexicon and can be used to verify the final model.

3.4 Change between Model Versions

Problem description Some tools, such as CoreNLP and TreeTagger, are regularly updated or released in new versions. It is usually neither obvious nor documented if and in which way the models change as part of an update. Models packaged with different versions of a tool may actually be the same. For example the CoreNLP package has for a long time included a set of models with each released version of the package, but between two releases, only a few of these models changed. We also observe that models may change without any version number increasing. For example, the models available from the TreeTagger website are regularly updated, but their file name or version does not change in most cases. Users can only observe such changes if they download the models at different points in time and compare the binary files to each other bit-by-bit. If there *are* changes, they may have impact on the results. Negative impact could be caused e.g. by changes in the normalization. However, there could also be positive impact, e.g. if new models are trained on additional data and thus may provide additional coverage or accuracy.

Implementation To get an impression of the the change over time, we analyzed the oldest and the most recent model within each series and compared the results.

Results Of the 118 analyzed series, 48 consist of more than one model. Most of these remain largely stable in their lexicon size over time. Only 10 series exhibit a change of $> 1\%$ from the oldest to the newest model in the series (Table 6). This indicates that results based on most models should remain comparable irrespective of the model version used. On the other hand, it would have also been great to observe that training data is continually extended over the years.

			Affected lexicon entries		Δ	
Tool ID	Lang	Model	oldest	newest	Abs	Rel %
C-NER	en	all.3class.caseless.distsim.crf	32809	41092	8283	25.25
C-NER	en	all.3class.distsim.crf	24008	40572	16564	68.99
C-NER	en	conll.4class.distsim.crf	13677	13952	275	2.01
C-NER	en	muc.7class.distsim.crf	10364	10901	537	5.18
C-PAR	zh	factored	37687	55943	18256	48.44
C-PAR	zh	pcfg	37687	55943	18256	48.44
C-TAG	de	dewac	44581	47916	3335	7.48
C-TAG	de	fast	44581	47916	3335	7.48
C-TAG	de	hgc	44581	47916	3335	7.48
T-TAG	es	le	74289	84217	9928	13.36

Table 6: Series with lexicon size changes ($> 1\%$)

In terms of normalization, we find that some series switch from regular brackets to PTB brackets (round 4, square 5), from escaped slashes to unescaped slashes (2), and two series fix their issues of mixed PTB/normal brackets. It is also notable that in two cases (C-TAG *bidirectional-distsim*, *left3words-distsim*) the lexicon entries split by the JTok tokenizer drop by over 50% from 1666 to 800 due to a switch from escaped to unescaped slashes.

Recommendation Users should be careful when upgrading to a new version of a model and ensure that after the upgrade their setup still produces comparable results to before the upgrade. If the results change unexpectedly after the upgrade, *Model Investigator* can be used to get additional insight by generating a report of the characteristics of the old and new model and to comparing these to each other.

4 Case study: Impact of bad character encoding

In this section, we revisit the issue of models that were trained with a bad character encoding (cf. Section 3.1) and investigate the negative impact it has on the accuracy. We previously identified two series of models to be affected by a *type B* encoding problem: during the training of the model, an UTF-8 corpus was read as ISO 8859. Since this type of encoding problem is reversible, it is possible to transform the corpus tokens to the same broken encoding that is used in the model without any loss of information.

We measured the tagging accuracy of the German OpenNLP maxent POS tagger model once out-of-the-box and once transforming the tokens to the same broken encoding used in the model before passing them to the tagger. As evaluation datasets, we used the TIGER treebank (Brants et al., 2002) and on the TüBa D/Z treebank (Telljohann et al., 2004). Note that the German OpenNLP POS tagger model was originally trained on the TIGER corpus, so the accuracy on this corpus should not be taken as an indicator of the model quality, but only for the change in accuracy between broken and ”fixed” encoding. The tagging accuracy of the model increases by 1.26 on the TüBa D/Z corpus and by 1.38 on the TIGER corpus when the tokens are transformed to the broken model encoding before passing them to the tagger.

In the same way, we tested the broken Spanish CoreNLP PCFG parser model. Note that we evaluated only the the POS tagging accuracy of the model, not the parsing accuracy. Due to a lack of any other Spanish corpus with a comparable tagset, again, we evaluated the model's accuracy against its training corpus. The model was trained on the AnCore 3.0.0 corpus. We evaluated against the 3.0.1 version of AnCora[8] and measured an improvement 5.05 in tagging accuracy when ”fixing” the encoding.

					Encoding	Accuracy		
Tool ID	Lang	Model	Version	Corpus	defect in	broken	fixed	Δ
O-TAG	de	maxent	20120616	TIGER 2.2	model	97.59	98.97	1.38
O-TAG	de	maxent	20120616	TüBa D/Z 10	model	92.66	93.92	1.26
C-PAR	es	pcfg	20150108	AnCora 3.0.1	model	86.27	91.32	5.05

Figure 1: Comparison of applying models with broken encoding 1) as-is to a corpus (column *broken*) vs. 2) transforming the corpus tokens to the same broken representation used in the model before applying the model (column *fixed*).

5 Conclusion

In this paper, we have analyzed publicly distributed pre-trained models for NLP tools for issues related to *encoding*, *tokenization*, *normalization*, and *change*. We found several serious flaws which have been communicated to the respective model creators. To support model creators to perform automatic sanity checks on their models and to allow model consumers to get a better understanding of the models they use, we introduce a new tool called *Model Investigator*.

As future work, we plan to extend *Model Investigator* so it can be applied directly to training corpora before creating models. In our opinion, certain normalization steps (e.g. removing SHY or NBSP characters) should rather be fixed in the underlying corpora than during training or processing. Due to the frequency cutoff often used when training models, systematic issues on low-frequency tokens may go unnoticed and analyzing the corpora directly may, thus, reveal additional issues. Finally, more checks should be implemented to investigate deeper into potential source of problems raised by the increasing use of Unicode. For example, it should be investigated whether homoglyphs in Unicode (e.g. different dashes) are problematic.

Our techniques can also be embedded e.g. into UIMA or GATE tool wrappers to dynamically configure normalization steps or make up for bad encoding. This is another area we plan to investigate further in future work.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme (H2020-EINFRA-2014-2) under grant agreement No. 654021. It reflects only the author's views and the EU is not liable for any use that may be made of the information contained therein. It was further supported by the German Federal Ministry of Education and Research (BMBF) under the promotional reference 01UG1416B (CEDIFOR).

[8]The AnCora corpus is missing POS tags in several cases. To train the Spanish CoreNLP models, a heuristic was applied to fill in missing POS tags. It is implemented in the CoreNLP class *SpanishXMLTreeReader*. We used the same heuristic to fill in missing POS tags during evaluation.

References

Sabine Brants, Stefanie Dipper, Silvia Hansen, Wolfgang Lezius, and George Smith. 2002. The TIGER treebank. In *Proceedings of the Workshop on Treebanks and Linguistic Theories (TLT02)*, pages 24–41, Sozopol, Bulgaria, September.

Hamish Cunningham, Diana Maynard, Kalina Bontcheva, Valentin Tablan, Niraj Aswani, Ian Roberts, Genevieve Gorrell, Adam Funk, Angus Roberts, Danica Damljanovic, Thomas Heitz, Mark A. Greenwood, Horacio Saggion, Johann Petrak, Yaoyong Li, and Wim Peters. 2011. *Text Processing with GATE (Version 6).*

Rebecca Dridan and Stephan Oepen. 2012. Tokenization: Returning to a long solved problem – a survey, contrastive experiment, recommendations, and toolkit. In *Proceedings of ACL-2012*, pages 378–382, Jeju Island, Korea, July. ACL.

Richard Eckart de Castilho and Iryna Gurevych. 2014. A broad-coverage collection of portable nlp components for building shareable analysis pipelines. In *Proceedings of OIAF4HLT*, pages 1–11, Dublin, Ireland, August. ACL and Dublin City University.

David Ferrucci and Adam Lally. 2004. UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment. *Natural Language Engineering*, 10(3-4):327–348, September.

Marie Hinrichs, Thomas Zastrow, and Erhard Hinrichs. 2010. WebLicht: Web-based LRT Services in a Distributed eScience Infrastructure. In Nicoletta Calzolari, Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, Mike Rosner, and Daniel Tapias, editors, *Proceedings of LREC-2010*, pages 489–493, Valletta, Malta, May. ELRA.

Nancy Ide, James Pustejovsky, Christopher Cieri, Eric Nyberg, Di Wang, Keith Suderman, Marc Verhagen, and Jonathan Wright. 2014. The Language Application Grid. In Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Hrafn Loftsson, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis, editors, *Proceedings of LREC-2014*, Reykjavik, Iceland, May. ELRA.

Emanuele Lapponi, Erik Velldal, Nikolay A. Vazov, and Stephan Oepen. 2013. Towards large-scale language analysis in the cloud. In *Proceedings of the workshop on Nordic language research infrastructure at NODALIDA 2013; NEALT Proceedings Series 20*, number 89, pages 1–10, Oslo, Norway, July. Linköping University Electronic Press.

Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of ACL-2014*, pages 55–60, Baltimore, Maryland, USA, June. Association for Computational Linguistics.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a large annotated corpus of English: the Penn Treebank. *Comput. Linguist.*, 19(2):313–330, June.

Helmut Schmid. 1994. Probabilistic part-of-speech tagging using decision trees. In *Proceedings of International Conference on New Methods in Language Processing*, pages 44–49, Manchester, UK.

Mariona Taulé, M. Antònia Martí, and Marta Recasens. 2008. AnCora: Multilevel annotated corpora for Catalan and Spanish. In Nicoletta Calzolari, Khalid Choukri, Bente Maegaard, Joseph Mariani, Jan Odijk, Stelios Piperidis, and Daniel Tapias, editors, *Proceedings of LREC-2008*, Marrakech, Morocco, May. ELRA. http://www.lrec-conf.org/proceedings/lrec2008/.

Heike Telljohann, Erhard Hinrichs, and Sandra Kübler. 2004. The TüBa-D/Z treebank: Annotating German with a context-free backbone. In *Proceedings of LREC-2004*, pages 2229–2235, Lisbon, Portugal.