DeepNL: a Deep Learning NLP pipeline

Giuseppe Attardi
Dipartimento di Informatica
Università di Pisa
Pisa, Italy
attardi@di.unipi.it

Abstract

We present the architecture of a deep learning pipeline for natural language processing. Based on this architecture we built a set of tools both for creating distributional vector representations and for performing specific NLP tasks. Three methods are available for creating embeddings: feed-forward neural network, sentiment specific embeddings and embeddings based on counts and Hellinger PCA. Two methods are provided for training a network to perform sequence tagging, a window approach and a convolutional approach. The window approach is used for implementing a POS tagger and a NER tagger, the convolutional network is used for Semantic Role Labeling. The library is implemented in Python with core numerical processing written in C++ using parallel linear algebra library for efficiency and scalability.

1 Introduction

Distributional Semantic Models (DSM) that represent words as vectors of weights over a high dimensional feature space (Hinton et al., 1986), have proved very effective in representing semantic or syntactic aspects of lexicon. Incorporating such representations has allowed improving many natural language tasks. They also reduce the burden of feature selection since these models can be learned through unsupervised techniques from text.

Deep learning algorithms for NLP tasks exploit distributional representation of words. In tagging applications such as POS tagging, NER tagging and Semantic Role Labeling (SRL), this has proved quite effective in reaching state of art accuracy and reducing reliance on manually engineered feature selection (Collobert et al, 2011).

Word embeddings have been exploited also in constituency parsing (Collobert, 2011) and dependency parsing (Chen and Manning, 2014).

A further benefit of a deep learning approach is to allow performing multiple tasks jointly, and therefore reducing error propagation as well as improving efficiency.

This paper presents DeepNL, an NLP pipeline based on a common Deep Learning architecture: it consists of tools for creating embeddings, and tools that exploit word embeddings as features. The current release includes a POS tagger, a NER, an SRL tagger and a dependency parser.

Two methods are supported for creating embeddings: an approach that uses neural network and one using Hellinger PCA (Lebret and Collobert 2014).

2 NLP Toolkits

A short survey of NLP toolkits is presented by Krithika and Akondi (2014).

NLTK is among the most well-known and comprehensive NLP toolkits: it is written in Python and provides a number of basic processing facilities (tokenization, splitting, statistical analysis of corpora, etc.) as well as machine learning algorithms for classification and clustering. Currently it does not provide any tool based on word embeddings, however it can be interfaced to SENNA[1] or it can be used in conjunction with Gensim[2] which provides several algorithms for performing unsupervised semantic modeling from plain text, including word embeddings, random indexing, LDA (Latent Dirichlet Allocation).

The Stanford NLP Toolkit (Manning et al., 2014) is written in Java and provides tools for tokenization, sentence splitting, POS tagging, NER, parsing, sentiment analysis and temporal expression tagging. As a recent inclusion, it pro-

[1] http://ronan.collobert.com/senna/
[2] http://radimrehurek.com/gensim/

Proceedings of NAACL-HLT 2015, pages 109–115,
Denver, Colorado, May 31 – June 5, 2015.

vides a dependency parser based on neural network and word embeddings (Chen et al., 2014).

OpenNLP[3] is a machine learning library written in Java that supports the most common NLP tasks, such as tokenization, sentence segmentation, POS tagging, named entity extraction, chunking, parsing, and coreference resolution.

Typically each tool built with these libraries uses a different approach or an most suitable algorithm for the task: for example Sanford NLP uses Conditional Random Fields for NER while the POS tagger uses MaxEntropy and both require a set of rich features that need to be manually engineered.

DeepNL differs from these toolkits since it is based on a common deep learning architecture: all tools exploit the same core neural network and use mostly just word embeddings as features. For example the POS tagger and the NER tagger have an identical structure, and they differ only in the way they read/write documents and in the configuration of the discrete features used: the POS tagger uses word suffixes while the NER uses gazetteer dictionaries. Embeddings are used as features, providing a continuous rather than discrete representation of text.

The ability of creating suitable embeddings for various tasks is critical for the proper working of the tools in DeepNL; hence the toolkit integrates algorithms for creating word embeddings from text, either in unsupervised or supervised fashion.

3 Building Word Embeddings

Word embeddings provide a low dimensional vector space representation for words, where values in each dimension may represent syntactic or semantic properties.

DeepNL provides two methods for building embeddings, one is based on the use of a neural language model, as proposed by (Turian and Bengio; Collobert et al., 2011; Mikolov et al., 2010) and one based on spectral method as proposed by Lebret and Collobert (2013).

The neural language method can be hard to train and the process is often quite time consuming, since several iterations are required over the whole training set. Some researchers provide precomputed embeddings for English[4]. The Polyglot project (Al-Rafou et al., 2013) makes available embeddings for several languages, built from the plain text of Wikipedia in the respective language, and the Python code for computing them[5], that supports GPU computations by means of Theano[6].

Mikolov et al. (2013) developed an alternative solution for computing word embeddings, which significantly reduces the computational costs. They propose two log-linear models, called bag of words and skip-gram model. The bag-of-word approach is similar to a feed-forward neural network language model and learns to classify the current word in a given context, except that instead of concatenating the vectors of the words in the context window of each token, it just averages them, eliminating a network layer and reducing the data dimensions. The skip-gram model tries instead to estimate context words based on the current word. Further speed up in the computation is obtained by exploiting a mini-batch Asynchronous Stochastic Gradient Descent algorithm, splitting the training corpus into partitions and assigning them to multiple threads. An optimistic approach is also exploited to avoid synchronization costs: updates to the current weight matrix are performed concurrently, without any locking, assuming that updates to the same rows of the matrix will be infrequent and will not harm convergence.

The authors published single-machine multi-threaded C++ code for computing the word vectors[7]. A reimplementation of the algorithm in Python is included in the Genism library (Řehůřek and Petr Sojka, 2010). In order to obtain comparable speed to the C++ version, they use Cython for interfacing a coding in C of the core function for training the network on a single sentence, which in turn exploits the BLAS library for algebraic computations.

The DeepNL implementation is written in Cython[8] and uses C++ code which exploits the Eigen[9] library for efficient parallel linear algebra computations. Data is exchanged between Numpy arrays in Python and Eigen matrices by means of Eigen Map types. On the Cython side, a pointer to the location where the data of a Numpy array is stored is obtained with a call like:

[3] http://opennlp.apache.org/
[4] http://ronan.collobert.com/senna/
http://metaoptimize.com/projects/wordreprs/
http://www.fit.vutbr.cz/~imikolov/rnnlm/
http://ai.stanford.edu/~ehhuang/

[5] https://bitbucket.org/aboSamoor/word2embeddings
[6] http://deeplearning.net/software/theano/
[7] https://code.google.com/p/word2vec
[8] http://docs.cython.org/
[9] http://eigen.tuxfamily.org/

```
<FLOAT_t*>np.PyArray_DATA(self.nn.hid
den_weights)
```

and passed to a C++ method. On the C++ side this is turned into an Eigen matrix, with no computational costs due to conversion or allocation, with the code:

```
Map<Matrix> hidden_weights(
hidden_weights, numHidden, numInput)
```

which interprets the pointer to a double as a matrix with `numHidden` rows and `numInput` columns. Since Eigen by default uses column-major order while Numpy uses row-major order, the class `Matrix` above is declared as:

```
typedef Eigen::Matrix<double, Eig-
en::Dynamic, Eigen::Dynamic, Eig-
en::RowMajor> Matrix;
```

3.1 Word Embeddings through Hellinger PCA

Lebret and Collobert (2013) have shown that embeddings can be efficiently computed from word co-occurence counts, applying Principal Component Analysis (PCA) to reduce dimensionality while optimizing the Hellinger similarity distance.

Levy and Goldberg (2014) have shown similarly that the skip-gram model by Mikolov et al.(2013) can be interpreted as implicitly factorizing a word-context matrix, whose values are the pointwise mutual information (PMI) of the respective word and context pairs, shifted by a global constant.

DeepNL provides an implementation of the Hellinger PCA algorithm using Cython and the LAPACK library SSYEVR from Scipy[10].

Cooccurrence frequencies are computed by counting the number of times each context word $w \in \mathcal{D}$ occurs after a sequence of T words:

$$p(w|T) = \frac{p(w,T)}{p(T)} = \frac{n(w,T)}{\sum_n n(w,T)}$$

where $n(w, T)$ is the number of times word w occurs after a sequence of T words. The set $\mathcal{D}$ of context word is normally chosen as the the subset of the top most frequent words in the vocabulary $\mathcal{V}$.

The word co-occurrence matrix C of size $|\mathcal{V}|\times|\mathcal{D}|$ is built. The coefficients of C are square rooted and then its transpose is multiplied by it to obtain a symmetric square matrix of size $|\mathcal{V}|\times|\mathcal{V}|$, to which PCA is applied to obtain the desired dimensionality reduction.

3.2 Sentiment Specific Word Embeddings

For the task of sentiment analysis, semantic similarity is not appropriate, since antonyms end up at close distance in the embeddings space. One needs to learn a vector representation where words of opposite polarity are further apart.

Tang et al. (2014) propose an approach for learning sentiment specific word embeddings, by incorporating supervised knowledge of polarity in the loss function of the learning algorithm. The original hinge loss function in the algorithm by Collobert et al. (2011) is:

$$\mathcal{L}_{CW}(x, x^c) = max(0, 1 - f_\theta(x) + f_\theta(x^c))$$

where x is an ngram and x^c is the same ngram corrupted by changing the target word with a randomly chosen one, $f_\theta(\cdot)$ is the feature function computed by the neural network with parameters θ. The sentiment specific network outputs a vector of 2 dimensions, one for modeling the generic syntactic/semantic aspects of words and the second for modeling polarity.

A second loss function is introduced as objective for minimization:

$$\mathcal{L}_{SS}(x, x^c) = max(0, 1 - \delta_s(x)\, f_\theta(x)_1 + \delta_s(x)\, f_\theta(x^c)_1)$$

where δ_s is an indicator function reflecting the sentiment polarity of a sentence,

$$\delta_s(x) = \begin{cases} 1 \; if \; f^g(x) = [1,0] \\ 0 \; if \; f^g(x) = [0,1] \end{cases}$$

where $f^g(x)$ is the gold distribution for ngram x. The overall hinge loss is a linear combination of the two:

$$\mathcal{L}(x, x^c) = \alpha\, \mathcal{L}_{CW}(x, x^c) + (1 - \alpha)\, \mathcal{L}_{SS}(x, x^c)$$

The gradient for the output layer is given by the formula:

$$\begin{pmatrix} \frac{\partial \mathcal{L}}{\partial f_\theta(x)} \\ \frac{\partial \mathcal{L}}{\delta f_\theta(x^c)} \end{pmatrix}_0 = \begin{cases} \begin{pmatrix} -1 \\ 1 \end{pmatrix} \; if \; \mathcal{L}_{CW}(x, x^c) > 0 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \text{ otherwise} \end{cases}$$

$$\begin{pmatrix} \frac{\partial \mathcal{L}}{\partial f_\theta(x)} \\ \frac{\partial \mathcal{L}}{\delta f_\theta(x^c)} \end{pmatrix}_1 = \begin{cases} \begin{pmatrix} 1 \\ -1 \end{pmatrix} \; if \; \mathcal{L}_{SS}(x, x^c) > 0 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \text{ otherwise} \end{cases}$$

DeepNL provides an algorithm for training polarized embeddings, performing gradient descent

[10] https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.linalg.lapack.ssyevr.html

using an adaptive learning rate according to the AdaGrad method (Duchi et al, 2011). The algorithm requires a training set consisting of sentences annotated with their polarity, for example a corpus of tweets. The algorithm builds embeddings for both unigrams and ngrams at the same time, by performing variations on a training sentence replacing not just a single word, but a sequence of words with either another word or another ngram.

4 Deep Learning Architecture

DeepNL adopts a multi layer neural network architecture, as proposed in (Collobert et al., 2011):

1. Lookup layer. It maps word feature indices to a feature vector, as described below.
2. Linear layer. Fully connected network layer, represented by matrix M_1 and input bias b_1.
3. Activation layer (e.g. hardtanh)
4. Linear layer. Fully connected network layer, represented by matrix M_2 and input bias b_2
5. Softmax layer. Computes the softmax of the output values, producing a probability distribution of the outputs.

Overall, the network computes the following function:

$$f(x) = \text{softmax}(M_2\, a(M_1\, x + b_1) + b_2)$$

where $M_1 \in \mathbb{R}^{h \times d}$, $b_1 \in \mathbb{R}^d$, $M_2 \in \mathbb{R}^{o \times h}$, $b_2 \in \mathbb{R}^o$, are the parameters, with d the dimension of the input, h the number of hidden units, o the number of output classes, $a(\cdot)$ is the activation function.

4.1 Lookup layer

The first layer of the network transforms the input into a feature vector representation. Individual words are represented by a tuple of K discrete features, $w \in \mathcal{D}^1 \times \ldots \times \mathcal{D}^k$, where $\mathcal{D}^k$ is the dictionary for the k-th feature.

Each feature has its own *lookup table* $LT_{W^k}(\cdot)$, with a matrix of parameters to be learned $W^k \in \mathbb{R}^{d^k \times |\mathcal{D}^k|}$, where $\mathcal{D}^k$ is the dictionary for the k-th feature and d^k is a user specified vector size. The *lookup table* layer $LT_{W^k}(\cdot)$ associates a vector of weights to each discrete feature $f \in \mathcal{D}^k$:

$$LT_{W^k}(f) = \langle W^k \rangle_f^1$$

where $\langle W^k \rangle_f^1 \in \mathbb{R}^{d_k}$ is the f^{th} column of W and d_k is the word vector size (a hyper-parameter to be chosen by the user).

The feature vector for word w becomes the concatenation of the vectors for all features:

$$LT_{W^1}(w_1) LT_{W^2}(w_2) \cdots LT_{W^K}(w_k)$$

This vector of features for word w, is passed as input to the network. W^k, M_1, b_1, M_2 and b_2 are the parameters to be learned by backpropagation.

4.2 Feature Extractors

The library has a modular architecture that allows customizing a network for specific tasks, in particular its first layer, by supplying extractors for various types of features.

An extractor is defined as a class that inherits from an abstract class with the following interface:

```
class Extractor(object):
   def extract(self, tokens)
   def lookup(self, feature)
   def save(self, file)
   def load(self, file)
```

Method `extract`, applied to a list of tokens, extracts features from each token and returns a list of IDs for those features. The argument is a list of tokens rather than a single token, since features might depend on consecutive tokens. For instance a gazetteer extractor needs to look at a sequence of tokens to determine whether they are mentioned in its dictionary.

Method `lookup` returns the vector of weights for a given feature. Methods `save`/`load` allow saving and reloading the `Extractor` data to/from disk.

Extractors currently include an `Embeddings` extractor, implementing the word lookup feature, a `Caps`, `Prefix` and `Postfix` extractors for dealing with capitalization and prefix/postfix features, a `Gazetteer` extractor for dealing with the gazetteers typically used in a NER, and a customizable `AttributeFeature` extractor that extracts features from the state of a Shift/Reduce dependency parser, i.e. from the tokens in the stack or buffer as described for example in Nivre (2007).

5 Sequence Taggers

For sequence tagging, two approaches were proposed in Collobert at al. (2011), a window approach and a sentence approach. The window approach assumes that the tag of a word depends mainly on the neighboring words, and is suitable for tasks like POS and NE tagging. The sentence approach assumes that the whole sentence must be taken into account by adding a convolution layer after the first lookup layer and is more suitable for tasks like SRL.

We can train a neural network to maximize the log-likelihood over the training data. Denoting by θ the trainable parameters, including the network and the transition scores, we want to maximize the following log-likelihood with respect to θ:

$$\sum_{(x,t)\in T} \log p(t|x,\theta)$$

where *x* are all training sentences and *t* their corresponding tag sequence.

The score $s(x, t, \theta)$ of a sequence of tags *t* for a sentence *x*, with parameters θ, is given by the sum of the transition scores and the tag scores:

$$s(x,t,\theta) = \sum_{i=1}^{n} \left(T(t_{i-1},t_i) + f_\theta(x_i,t_i)\right)$$

where $T(i, j)$ is the score for the transition from tag i to tag j, and $f_\theta(t_i, x_i)$ is the output of the network at word x_i for tag t_i. The probability of a sequence *y* for sentence *x* can be expressed as:

$$p(y|x,\theta) = \frac{e^{s(x,y,\theta)}}{\sum_t e^{s(x,t,\theta)}}$$

If we define:

$$\underset{i}{\text{logadd}}\, x_i = \log \sum_i e^{x_i}$$

the log of the conditional probability of the correct sequence *y* is given by:

$$\log p(y|x,\theta) = s(x,y,\theta) - \underset{t}{\text{logadd}}\, s(x,t,\theta)$$

The probability can be computed iteratively by defining:

$$\partial_i(a) = \underset{t_i=a}{\text{logadd}}\, s(x_1^i, t_1^i, \theta)$$

$$= \underset{b}{\text{logadd}}(\partial_{i-1}(b) + T(b,a)) + f_\theta(a,i)\ \ \forall a$$

and finally

$$\underset{t}{\text{logadd}}\, s(x,t,\theta) = \underset{a}{\text{logadd}}\, \delta_{|x|}(a)$$

In order to avoid numeric overflows, the function logadd must be computed carefully, i.e. by subtracting the maximum value to the coefficients before performing exponentiation and then re-adding the maximum.

The computation of the gradients can be performed at once for the whole sequence exploiting matrix operations whose computation can be optimized and parallelized using suitable linear algebra libraries. We implemented two versions of the network trainer, one in Python using NumPy[11] and one in C++ using Eigen[12].

Here for example is the Python code for computing the δ in the above equation:

```
delta = scores
delta[0] += transitions[-1]
tr = transitions[:-1].T
for i in xrange(1, len(delta)):
  # sum by rows
  logadd = logsumexp(delta[i-1]+tr,
1)
  delta[token] += logadd
```

The array `scores[i, j]` contains the output of the neural network for the `i`-th element of the sequence and for tag `j`, `delta[i, j]` represents the sum of all scores ending at the `i`-th token with tag `j`; `transitions[i, j]` contains the current estimate of the probability of a transition from tag `i` to tag `j`.

6 Experiments

We tested the DeepNL sequence tagger on the CoNLL 2003 challenge[13], a NER benchmark based on Reuters data. The tagger was trained with three types of features: the word embeddings from SENNA, a "caps" feature telling whether a word is in lowercase, uppercase, title case, or had at least one non-initial capital letter, and a gazetteer feature, based on the list provided by the organizers. The window size was set to 5, 300 hidden variables were used and training was iterated for 40 epochs. In the following table we report the scores compared with the system by Ando et al. (2005) which uses a semi-supervised approach and with the results by the released version of SENNA[14]:

[11] http://www.numpy.org/
[12] http://eigen.tuxfamily.org/
[13] http://www.cnts.ua.ac.be/conll2003/ner/
[14] http://ml.nec-labs.com/senna/

System	F1
Ando et al. 2005	89.31
SENNA	89.51
DeepNL	89.38

Table 1. Performance on the NER task, using the CoNLL 2003 benchmark.

The slight difference with SENNA might be explained by the use of different gazetteers.

The same sequence tagger can be used for POS tagging. In this case the discrete features used are the same capitalization feature as for the NER and a suffix feature, which denotes whether a token ends with one of the 455 most frequent suffixes of length one or two characters in the training corpus.

Table 2 presents the results achieved by the POS tagger trained on the Penn Treebank, compared with the results of the reference system by Tuotanova et al. (2003), which uses rich features, and with the original SENNA implementation.

System	Precision
Toutanova et al. 2003	97.24
SENNA	97.28
DeepNL	97.12

Table 2. Performance on the POS task, using the Penn Treebank, sections 0-18 for training, sections 22-24 for testing.

Both these experiments confirm that word embeddings can replace the use of complex manually engineered features for typical natural language processing tasks.

7 Dependency Parsing

We have adapted to the use of embeddings our original transition based dependency parser DeSR (Attardi et al., 2009), that was already based on a neural network. The parser uses the neural network to decide which action to perform at each step in the analysis of a sentence. Looking at a short context of past analyzed tokens and next input tokens, it must decide whether the two current focus tokens can be connected by a dependency relation. In this case it performs a reduction, creating the dependency, otherwise it advances on the input. The original implementation used a large set of discrete features to represent the current context.

The deep learning version of the parser exploits word embedding as features and also creates a dense vector representation for the remaining discrete features. A specific extractor (`AttributeExtractor`) was built for this purpose.

8 Conclusions

We have presented the architecture of DeepNL, a library for building NLP applications based on a deep learning architecture. The implementation is written in Python/Cython and uses C++ linear algebra libraries for efficiency and scalability, exploiting multithreading or GPUs where available.

The implementation of DeepNL is available on GitHub[15].

The availability of a library that allows creating embeddings and training a deep learning architecture using them might contribute to the development of further tools for linguistic analysis.

For example we are planning to build a classifier for performing identification of affirmative, negative or speculative contexts in sentences.

We are also considering additional ways of creating embeddings, for example to generate context sensitive embeddings that could provide word representations that disambiguate among word senses.

Acknowledgements

Partial support for this work was provided by project RIS (POR RIS of the Regione Toscana, CUP n° 6408.30122011.026000160).

References

R. Al-Rfou, B. Perozzi, and S. Skiena. 2013. Polyglot: Distributed Word Representations for Multilingual NLP. arXiv preprint arXiv:1307.1662.

R. K. Ando, T. Zhang, and P. Bartlett. 2005. A framework for learning predictive structures from multiple tasks and unlabeled data. Journal of Machine Learning Research, 6:1817–1853.

G. Attardi, F. Dell'Orletta, M. Simi, J. Turian. 2009. Accurate Dependency Parsing with a Stacked Multilayer Perceptron. In *Proc. of Workshop Evalita 2009*, ISBN 978-88-903581-1-1.

Danqi Chen and Christopher D. Manning. 2014. Fast and Accurate Dependency Parser using Neural Networks. In: *Proc. of EMNLP 2014*.

R. Collobert et al. 2011. Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research*, 12, 2461–2505.

[15] https://github.com/attardi/deepnl

R. Collobert and J. Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML*, 2008.

R. Collobert. 2011. Deep Learning for Efficient Discriminative Parsing. In AISTATS, 2011.

P. S. Dhillon, D. Foster, and L. Ungar. 2011. Multiview learning of word embeddings via CCA. In *Advances in Neural Information Processing Systems* (NIPS), volume 24.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*.

S. Hartmann, G. Szarvas, and I. Gurevych. 2011. Mining Multiword Terms from Wikipedia, in M.T. Pazienza & A. Stellato (Eds.): Semi-Automatic Ontology Development: Processes and Resources, pp. 226-258, Hershey, PA, USA: IGI Global.

G.E. Hinton, J.L. McClelland, D.E. Rumelhart. Distributed representations. 1986. In: Parallel distributed processing: Explorations in the microstructure of cognition. Volume 1: Foundations, MIT Press, 1986.

L.B. Krithika and Kalyana Vasanth Akondi. 2014. Survey on Various Natural Language Processing Toolkits. World Applied Sciences Journal 32 (3): 399-402.

Rémi Lebret and Ronan Collobert. 2013. Word Embeddings through Hellinger PCA. *Proc. of EACL 2013*.

Omer Levy and Yoav Goldberg. 2014. Neural Word Embeddings as Implicit Matrix Factorization. In *Advances in Neural Information Processing Systems* (NIPS), 2014.

Christopher D. Manning and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. The MIT Press. Cambridge, Massachusetts.

Manning, Christopher D., Surdeanu, Mihai, Bauer, John, Finkel, Jenny, Bethard, Steven J., and McClosky, David. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pp. 55-60.

T. Mikolov, M. Karafiat, L. Burget, J. Cernocky, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. In *Proceedings of Workshop at ICLR*, 2013.

J. Nivre. 2007. Incremental non-projective dependency parsing, *Proceedings of Human LanguageTechnologies: The Annual Conference of the North American Chapter of the Association for Computational Linguistics* (*NAACL HLT*), Rochester, NY, pp. 396–403.

Radim Řehůřek and Petr Sojka. 2010. Software Framework for Topic Modelling with Large Corpora. In *Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks*, ELRA, Valletta, Malta, pp. 45–50.

K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Conference of the North American Chapter of the Association for Computational Linguistics & Human Language Technologies* (*NAACL-HLT*).