Representation of Word Sentiment, Idioms and Senses

Giuseppe Attardi

Dipartimento di Informatica
Università di Pisa
Largo B. Pontecorvo, 3
I-56127 Pisa, Italy
attardi@di.unipi.it

Abstract. Distributional Semantic Models (DSM) that represent words as vectors of weights over a high dimensional feature space have proved very effective in representing semantic or syntactic word similarity. For certain tasks however it is important to represent contrasting aspects such as polarity, different senses or idiomatic use of words. We present two methods for creating embeddings that take into account such characteristics: a feed-forward neural network for learning sentiment specific and a skip-gram model for learning sense specific embeddings. Sense specific embeddings can be used to disambiguate queries and other classification tasks. We present an approach for recognizing idiomatic expressions by means of the embeddings. This can be used to segment queries into meaningful chunks. The implementation is available as a library implemented in Python with core numerical processing written in C++, using a 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 [12], 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 plain 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 [8].

Word embeddings have been exploited also in constituency parsing [8] and dependency parsing [4]. Blanco et al. [3] exploit word embeddings for identifying entities in web search queries.

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 [14].

2 Building Word Embeddings

Word embeddings provide a low dimensional dense 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 [25, 8, 17] and one based on spectral method as proposed by Lebret and Collobert [14].

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 researcher provide precomputed embeddings for English[1]. The Polyglot project [1] makes available embeddings for several languages, built from the plain text of Wikipedia in the respective language, and the Python code for computing them[2], that supports GPU computations by means of Theano[3].

Mikolov et al. [19] 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[4]. A reimplementation of the algorithm in Python is included in the Genism library [22]. In order to obtain comparable speed to the C++ version, they use Cython for interfacing to 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.

[1] http://ronan.collobert.com/senna/, http://metaoptimize.com/projects/wordreprs/, http://www.fit.vutbr.cz/˜imikolov/rnnlm/, http://ai.stanford.edu/˜ehhuang/
[2] https://bitbucket.org/aboSamoor/word2embeddings
[3] http://deeplearning.net/software/theano/
[4] https://code.google.com/p/word2vec

2.1 Word Embeddings through Hellinger PCA

Lebret and Collobert [14] 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 [15] have shown similarly that the skip-gram model by Mikolov et al. [19] 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 SYSEVR.

Co-occurrence frequencies are computed by counting the number of times each context word w ∈ $\mathcal{D}$ 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 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 for obtaining the desired dimensionality reduction.

2.2 Context Sensitive Word Embeddings

The meaning of words often depends on their context. Our approach for learning word embeddings in context is inspired by the method for learning paragraph vectors [13]. We improve on their approach, avoiding the cost of computing at query time an embedding for the paragraph of the query. Our solution bears some resemblance to the approach in [11].

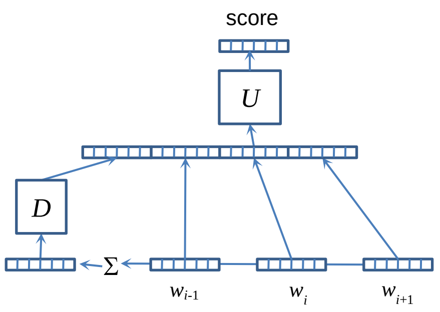

Figure 1. Overview of the model for context sensitive word embeddings. *U* and *D* are the matrices of weights to be learned.

Given a sequence of training words w_1, w_2, …, w_T, the objective function to maximize is the negative log likelihood:

$$\sum_{t=k}^{T-k} \log p(w_t | w_{t-k}, \dots, w_{t+k})$$

We add padding at sentence boundaries and substitute `<UNK>` for OOV words.

The prediction task is performed by a neural network with a softmax layer:

$$p(w_t | w_{t-k}, \dots, w_{t+k}) = \frac{e^{y_i}}{\sum_i e^{y_i}}$$

Each y_i is the score for output word i, computed as:

$$y = b + Uh(w_{t-k}, \dots, w_{t+k}; W, D)$$

where b, U are network parameters, W and D are the weight matrixes for words and paragraphs respectively. h concatenates the vectors of each word extracted from W and of their sum multiplied by D:

$$h(i_1, \dots, i_n; W, D) = W_1 || \dots || W_n || D \sum_i^n W_i$$

The combination of word vector and paragraph vector can be used for word sense disambiguation. A word i_k within paragraph $i_1,\dots,i_n$ is represented by the concatenation

$$W_{i_k} || D \sum_i^n W_i$$

2.3 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 far apart.

Tang et al. [24] 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. [6] 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 two 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 the subscript in $f_\theta(x)_1$ refers to the second element of the vector and $\delta_s(x)$ is an indicator function reflecting the sentiment polarity of a sentence, whose value is 1 if the sentiment polarity of x is positive and -1 if it is negative.

The overall hinge loss is a linear combination of the two:

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

DeepNL provides an algorithm for training polarized embeddings, performing gradient descent using an adaptive learning rate according to the AdaGrad method. 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.

3 Deep Learning Architecture

DeepNL adopts a multi-layer neural network architecture, as proposed in [6], consisting of five layers: a lookup layer, a linear layer, an activation layer (e.g. hardtanh), a second linear layer and a softmax layer. Overall, the network computes the following function:

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

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.

3.1 Lookup layer

The first layer of the network transforms the input into a feature vector representation. Individual words are represented by a vector of features, which is trained by back-propagation.

For each word $w \in \mathcal{D}$, an internal d-dimensional feature vector representation is given by the *lookup table* layer $LT_w(\cdot)$:

$$LT_W(w) = \langle W \rangle_w^1$$

where $W \in \mathbb{R}^{d \times |\mathcal{D}|}$is a matrix of parameters to be learned, $\langle W \rangle_w^1 \in \mathcal{D}$ is the w^{th} column of W and d is the word vector size (a hyper-parameter to be chosen by the user).

3.2 Discrete Features

Besides word representations, a number of discrete features can be used. Each feature has its own lookup table $LT_{W^k}(\cdot)$ with parameters $W^k \in \mathbb{R}^{d^k \times |\mathcal{D}^k|}$, where $\mathcal{D}^k$ is the dictionary for the k-th feature and d^k is a user specified vector size. The input to the network becomes the concatenation of the vectors for all features:

$$LT_{W^1}(w)LT_{W^2}(w)\cdots LT_{W^K}(w)$$

3.3 Sequence Taggers

For sequence tagging, two approaches were proposed in [6], 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 of the network, we want to maximize the following log-likelihood with respect to θ:

$$\sum_i \log p(t_i|c_i,\theta)$$

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

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

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

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

3.4 Experiments

We tested the DeepNL sequence tagger on the CoNLL 2003 challenge[5], a NER benchmark based on Reuters data. The tagger was trained with three types of features: 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. [2] which uses a semi-supervised approach and with the results by the released version of SENNA[6]:

[5] http://www.cnts.ua.ac.be/conll2003/ner/
[6] http://ml.nec-labs.com/senna/

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

The slight difference with SENNA is possibly due to the use of different suffixes.

4 Software Architecture

The DeepNL implementation is written in Cython and uses C++ code which exploits the Eigen[7] 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:

```
<FLOAT_t*>np.PyArray_DATA(self.nn.hidden_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 `nu-mInput` columns.

4.1 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. 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` extractor for dealing with capitaliza-

[7] http://eigen.tuxfamily.org/

tion 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 [20].

4.2 Parallel gradient computation

The computation of the gradients during network training requires computing the conditional probability over all possible sequences of tags, which grow exponentially with the length of the sequence. They can however be computed in linear time by accumulating them in a matrix, and then the matrix computation can be parallelized, as in the following code:

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

The computation can be optimized and parallelized using suitable linear algebra libraries. We implemented two versions of the network trainer, one in Python using NumPy[8] and one in C++ using Eigen[9].

5 Identification of Idiomatic Multiword Expressions

As an application of word embeddings, we experiment on the identification of idiomatic multiword expressions.

Multiword expressions are combinations of two or more words which can be syntactically and/or semantically idiosyncratic in nature. There are many varieties of multiword expressions: we concentrate on non-decomposable idioms, i.e. those idioms in which the meaning cannot be assigned to the parts of the MWE.

MWE identification is typically split into two phases: candidate identification and filtering.

For identifying potential candidates for MWE one can exploit the technique for discovering collocations [5] based on Pointwise Mutual Information.

[8] http://www.numpy.org/
[9] http://eigen.tuxfamily.org/

We adopt the simple variant proposed by Mikolov et al. (2013a), of computing a score for the likelihood of forming a collocation, using the unigram and bigram counts:

$$score(w_i, w_j) = \frac{count(w_i, w_j) - \delta}{count(w_i) \cdot count(w_j)}$$

The bigrams with score above a chosen threshold are then used as phrases. The δ is a discounting coefficient and prevents generating too many phrases consisting of very infrequent words. We also apply a cutoff on the frequency of bigrams, to avoid depending too much on the frequency of the individual words and in particular to limit the tendency of assigning higher scores to lower frequency words.

The process is repeated a few times by replacing the bigrams with a single token and decreasing the threshold value, in order to extract longer phrases.

As many have noted [16], just relying on statistical measures of frequency for identifying MWEs does not achieve very satisfactory results, since idiomatic phrases are not that much frequent in texts, hence data is sparse; therefore some sort of semantic knowledge is required.

Srivastava and Hovy [23] introduce a segmentation model for partitioning a sentence into linear constituents, called motifs, which is learned though semi-supervised learning. They then build embeddings for such motifs using the Hellinger PCA technique of Lebret and Collobert [14].

For deciding whether a candidate collocation is indeed a phraseme, we rely on the distinctive properties of idiomatic expressions: *non-composability*, i.e. their meaning is not obtainable as a composition of the meaning of its part; *non-substitutivity*, i.e. replacing near-synonyms for the parts of a phrase would produce something weird or nonsensical.

We assume that these two aspects should be fairly evident to the reader, otherwise he would not be able to distinguish a phraseme from a normal phrasal combination. Therefore, if we replace some of the words in the expression with similar words, we should end up with an apparently weird combination.

The basic idea in our experiments is to select replacement words that are similar according to their distance in the word embedding space. As a criterion for deciding if a phrase is unusual, we check first if no variant occurs in the corpus, otherwise we check whether the LM probability of all variants is below a given threshold.

5.1 Experiments

We carried out experiments on the corpus consisting of the plain text extracted from the English Wikipedia, for a total of 1,096,243,235 tokens, 4,456,972 distinct.

We created word embeddings on the corpus obtained by performing token combinations as described above, using a threshold of 500 on the first iteration and 300 on the following ones. We used a cutoff of 80 on the first iteration and 40 on the following ones. The vocabulary for this corpus consists of 225,000 words or phrases.

For evaluating our model, we used the WikiMwe corpus [10], which includes a gold evaluation set consisting of 2,500 expressions, annotated in four categories: non-compositional, collocation, regular natural language phrase and ungrammatical. Table 2 shows the results of our experiments.

Type	Precision	Recall	F1
MWE	53.61	58.73	55.05
Regular	51.36	66.60	58.00
Ungrammatical	**5.48**	**40.00**	**9.64**

Table 1. Results on the idetification of idiomatic expressions.

The results are encouraging since the best level of accuracy reported in Vincze et al. [26] is 55.75% F1 for noun compounds.

Table 2 shows a few examples of the output of our system on the WikiMwe test set.

Phrase	ngrams	LM prob.	type	Correct
dual gauge	232	-2.1	MWE	yes
art of being	0	-2.9	MWE	yes
protest against the war	0	-2.0	Colloc	yes
way to Damascus	0	-3.7	Colloc	yes
financial services	0	-1.8	Colloc.	no
androgenic alopecia	0	0.0	MWE	yes

Table 2. Samples of phrases and types assigned by our system.

An online demo of a similar system for the identification of Italian idiomatic phrases is available at: http://tanl.di.unipi.it/embeddings/mwe.

A potential application of the technique is the identification of chunks in search queries or in AdWords queries, in order to recognize expression whose intended meaning does not correspond to the combination of the individual words in the query.

6 Conclusions

We have presented the architecture of DeepNL, a library for building NLP applications based on a deep learning architecture.

The toolkit includes various methods for creating embedding, either generic embeddings and sentiment specific or context sensitive embeddings.

As an example of the effectiveness of the embeddings, we have explored their use in the identification of idiomatic word expressions.

The implementation is written in Python/Cython and uses C++ linear algebra libraries for efficiency and scalability, exploiting multithreading or GPUs where available. The code is available for download from: https://github.com/attardi/deepnl.

There are several potential applications for the library, in particular sentiment specific word embeddings might be applied to other classification tasks, for example detecting tweets that signal dangers or disasters.

Context sensitive word embeddings can be exploited in artificial tasks like word sense disambiguation or word sense similarity. Hopefully they should provide also benefits for more relevant tasks such as relation extraction, negation identification, data linking, ontology creation. We hope that the availability of the code will encourage exploring their use in such applications.

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

7 References

1. R. Al-Rfou, B. Perozzi, and S. Skiena. 2013. Polyglot: Distributed Word Representations for Multilingual NLP. arXiv preprint arXiv:1307.1662.
2. 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.
3. Roi Blanco, Giuseppe Ottaviano, Edgar Meij, 2015. Fast and Space-efficient Entity Linking in Queries, ACM WSDM 2015.
4. D. Chen and C. D. Manning. 2014. Fast and Accurate Dependency Parser using Neural Networks. In: *Proc. of EMNLP 2014*.
5. K. Church and P. Hanks. 1990. Word association norms, mutual information, and lexicography. *Computational Linguistics*. 16 (1): 22–29
6. R. Collobert et al. 2011. Natural Language Processing (Almost) from Scratch. *Journal of Machine Learning Research*, 12, 2461–2505.
7. R. Collobert and J. Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *ICML*, 2008.
8. R. Collobert. 2011. Deep Learning for Efficient Discriminative Parsing. In AISTATS, 2011.
9. 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.
10. 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.
11. Huang et al. 2012. Improving Word Representations via Global Context and Multiple Word Prototypes, *Proc. of the Association for Computational Linguistics 2012 Conference*.
12. 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.
13. Quoc Le and Tomas Mikolov. 2014. Distributed Representations of Sentences and Documents. In *Proceedings of the 31st International Conference on Machine Learning*, Beijing, China, 2014. JMLR:W&CP volume 32.
14. Rémi Lebret and Ronan Collobert. 2013. Word Embeddings through Hellinger PCA. *Proc. of EACL 2013*.

15. Omer Levy and Yoav Goldberg. 2014. Neural Word Embeddings as Implicit Matrix Factorization. In *Advances in Neural Information Processing Systems* (NIPS), 2014.
16. Christopher D. Manning and Hinrich Schütze. 1999. *Foundations of Statistical Natural Language Processing*. The MIT Press. Cambridge, Massachusetts.
17. T. Mikolov, M. Karafiat, L. Burget, J. Cernocky, and Sanjeev Khudanpur. 2010. Recurrent neural network based language model. In *INTERSPEECH 2010*, 11th Annual Conference of the International Speech Communication Association, Makuhari, Chiba, Japanfmikol.
18. 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.
19. Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Proceedings of NIPS*, 2013.
20. Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34(4):513-553.
21. Carlos Ramisch, Aline Villavicencio, and Christian Boitet. 2010. Multiword expressions in the wild? the mwetoolkit comes in handy. In Liu, Yang and Ting Liu, editors, *Proc. of the 23rd COLING* (COLING 2010) — Demonstrations, pages 57–60, Beijing, China.
22. 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.
23. S. Srivastava, E. Hovy. 2014. Vector space semantics with frequency-driven motifs. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, 634–643, Baltimore, Maryland, USA.
24. Tang et al. 2014. Learning Sentiment-SpecificWord Embedding for Twitter Sentiment Classification. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, pp. 1555–1565, Baltimore, Maryland, USA, June 23-25 2014.
25. Joseph Turian, Lev Ratinov, and Yoshua Bengio. 2010. Word representations: a simple and general method for semi-supervised learning. In *Proceedings of the 48th annual meeting of the association for computational linguistic*s, pp. 384-394. Association for Computational Linguistics.
26. Veronika Vincze, T. István Nagy, and Gábor Berend. 2011. Detecting noun compounds and light verb constructions: a contrastive study. In *Proceedings of the Workshop on Multiword Expressions: from Parsing and Generation to the Real World* (MWE '11). Association for Computational Linguistics, Stroudsburg, PA, USA, 116-121.