Originally published in: Zhao, Dongyan (Ed.): Proceedings of the 27th International Conference on Computational Linguistics: System Demonstrations. August 20-26, 2018, Santa Fe, New Mexico, USA. – Stroudsburg: Association for Computational Linguistics, 2018. Pp. 53-57.

Transparent, Efficient, and Robust Word Embedding Access with WOMBAT

Mark-Christoph Müller and **Michael Strube**
Heidelberg Institute for Theoretical Studies gGmbH
Schloss-Wolfsbrunnenweg 35
69118 Heidelberg, Germany
{mark-christoph.mueller|michael.strube}@h-its.org

Abstract

We present WOMBAT, a Python tool which supports NLP practitioners in accessing word embeddings from code. WOMBAT addresses common research problems, including unified access, scaling, and robust and reproducible preprocessing. Code that uses WOMBAT for accessing word embeddings is not only cleaner, more readable, and easier to reuse, but also much more efficient than code using standard in-memory methods: a Python script using WOMBAT for evaluating seven large word embedding collections (8.7M embedding vectors in total) on a simple SemEval sentence similarity task involving 250 raw sentence pairs completes in under ten seconds end-to-end on a standard notebook computer.

1 Motivation

Word embeddings are ubiquitous resources in current NLP which normally come as plain-text files containing collections of <string, real-valued vector> tuples. Each word embedding collection (WEC) is uniquely identified by its combination of 1) training *algorithm*, 2) training *parameters*, and 3) training *data*. The latter, in turn, is characterized by the *textual raw data* and the *preprocessing* that was applied to it.

Word embeddings are often used early on in the system pipeline: in a typical setup, a word embedding file is loaded up-front (*eager loading*), and vectors are looked up in memory and used as replacements for input words. This *native* approach to word embedding access has a couple of limitations with respect to transparency, efficiency, and robustness.

1. Writing code in which WECs are **easily and unambiguously identified** is difficult when each WEC is treated as a monolith in the file system.This way of identifying WECs completely disregards – and, in the worst case, obscures – the fact that these resources might *share* some of their meta data, resulting in different degrees of similarity between WECs: two or more WECs might be identical except for their training window sizes, or except for the fact that some additional postprocessing was applied to one of them. For intrinsic and extrinsic evaluation (Schnabel et al., 2015; Nayak et al., 2016) of the effect of different training parameters on WECs, these parameters need to be accessible explicitly, and not just on the level of file names.

2. Experiments with **large numbers of WECs** do not scale and are inefficient if entire files need to be loaded every time. Experiments involving large numbers of WECs are not uncommon: Baroni et al. (2014) employed 48 different WECs, while Levy et al. (2015) used as many as 672. More recently, Wendlandt et al. (2018) explored the (in)stability of word embeddings by evaluating WECs trained for all combinations of three algorithms (two of them involving a random component), five vector sizes (dimensions), and seven data sets. In order to include the effect of randomness, five sets of WECs with different initializations were trained for the two algorithms, resulting in 385 WECs altogether. Antoniak and Mimno (2018) focused on training *corpora*, in particular on the effect of three different sampling methods. They trained WECs for all combinations of these three methods, four algorithms, six data sets, and two segmentation sizes. To tackle the effect of randomness, they trained repeatedly for 50

Proceedings of the 27th International Conference on Computational Linguistics: System Demonstrations, pages 53–57
Santa Fe, New Mexico, USA, August 20-26, 2018.

times, producing a total of 7.200 WECs. None of these papers provides technical details on how WECs are handled, but the code that is available indicates that the native, eager loading approach seems to be prevalent. More sophisticated, selective access to stored WECs is required to speed up experimentation and also support more ad-hoc, explorative approaches.

3. Finally, **converting unrestricted raw data into units for WEC vector lookup** often amounts to guesswork because the original preprocessing code is not shared together with the WEC resource. Preprocessing – which can involve everything from lowercasing, tokenization, stemming or lemmatization, to stop word and special character removal, right up to detecting and joining multiword expressions – is often underestimated in NLP, and word embedding research is not an exception: For the well-known and widely used GloVe embeddings, the documentation simply states to first use "something like the Stanford Tokenizer"[1]. The 100B word data set used to train the GoogleNews embeddings[2] contains a considerable number of automatically detected multiword expressions. As a result, as many as 2.070.978 of the 3M vocabulary items are phrases joined with one or more "_" characters. Standard preprocessing without access to the same phrase extraction code cannot detect these items and will thus be blind to almost 70% of the GoogleNews WEC vocabulary. Any preprocessing code used in the creation of a WEC resource has to be considered an integral part of that resource. This is the only way to ensure that the resource is fully (re)usable, which in turn is a prerequisite for the reproducibility of experiments utilizing that resource. The topic of *reproducibility* has been around in e.g. computational biology for some time (Sandve et al., 2013), and is also gaining attention in NLP (see e.g. the 4REAL workshops in 2016 and 2018). Already in 2013, Fokkens et al. identified preprocessing, in particular tokenisation, as one of the major sources of errors in their attempts to reproduce NER results.[3]

While some word embedding APIs and toolkits do exist,[4] they mostly focus on providing interfaces for in-memory vector lookup or for higher-level similarity tasks. None of them adresses scalability or preprocessing issues.

2 WOMBAT in a Nut Shell

WOMBAT, the **WO**rd e**MB**edding d**AT**abase, is a light-weight Python tool for more transparent, efficient, and robust access to potentially large numbers of WECs. It supports NLP researchers and practitioners in developing compact, efficient, and reusable code. Key features of WOMBAT are 1. **transparent** identification of WECs by means of a clean syntax and human-readable features, 2. **efficient** *lazy*, on-demand retrieval of word vectors, and 3. increased **robustness** by systematic integration of executable preprocessing code. WOMBAT implements some *Best Practices* for research reproducibility (Sandve et al., 2013; Stodden and Miguez, 2014), and complements existing approaches towards WEC standardization and sharing.[5] The WOMBAT source code including sample WEC data is available at `https://github.com/nlpAThits/WOMBAT`.

WOMBAT provides a single point of access to *existing* WECs. Each plain text WEC file has to be imported into WOMBAT *once*, receiving in the process a set of `ATT:VAL` identifiers consisting of five system attributes (`algo, dims, dataset, unit, fold`) plus arbitrarily many user-defined ones.

```
from wombat import connector as wb_conn
wbc = wb_conn(path="/home/user/WOMBAT-data/", create_if_missing=True)
wbc.import_from_file("GoogleNews-vectors-negative300.txt",
                     "algo:w2v;dataset:googlenews;dims:300;fold:0;unit:token")
```

Importing the GoogleNews embeddings into WOMBAT.

The above code is sufficient to import the GoogleNews embeddings. The combination of identifiers, provided as a semicolon-separated string, must be unique, but the supplied order is irrelevant. In this

[1] `https://github.com/stanfordnlp/GloVe/blob/master/src/README.md`

[2] `https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing`

[3] Fokkens et al. (2013) do not address preprocessing for word embeddings, but their conclusions apply just the same.

[4] E.g. `https://radimrehurek.com/gensim/models/word2vec.html`, `https://github.com/3Top/word2vec-api`, `https://github.com/stephantul/reach`, `https://github.com/vecto-ai/vecto`

[5] E.g. `http://vectors.nlpl.eu/repository`, `http://wordvectors.org`, `https://github.com/JaredFern/VecShare`, `http://bit.ly/embeddings`

example, no additional user-defined attributes were assigned, as the publicly available GoogleNews WEC is sufficiently identifiable. For self-trained WECs, user-defined attributes for hyper-parameters including minimum frequency, window size, and training iterations are usually employed.

In WOMBAT, each WEC is stored in a single one-table relational database[6] with a *word* column and a *vector* column as a `float32` NumPy array, which significantly reduces the disk size, e.g. from 12.7 GB to 4.1 GB for GoogleNews. In order to maintain data integrity, the *word* column employs a `unique` database index to prevent multiple entries for the same word.

The most basic WOMBAT operation is the retrieval of embedding vectors from one or more WECs, which are specified by their identifiers. For this, WOMBAT provides a `get_vectors(...)` method supporting a grid search-friendly `ATT:{VAL1,VAL2,...,VALn}` identifier format, which is expanded into `n` atomic identifiers in the supplied order. In addition, several WEC identifiers can be concatenated with `&`. If input words are already preprocessed, they can directly be supplied as a nested Python `list`.

The following code retrieves vectors for the words `theory` and `computation` from all six GloVe WECs and from the GoogleNews WEC, in under two seconds on a standard notebook computer. The special identifier format is used here to specify all four GloVe 6B data sets, which share all properties except for `dims` (vector dimensionality). Other typical uses supported by this format include the evaluation of WECs trained with different hyper-parameters like e.g. `window`.

```
from wombat import connector as wb_conn
wbc = wb_conn(path="/home/user/WOMBAT-data/")
wecs = "algo:glove;dataset:6b;dims:{50,100,200,300};fold:1;unit:token&\
        algo:glove;dataset:42b;dims:300;fold:1;unit:token&\
        algo:glove;dataset:840b;dims:300;fold:0;unit:token&\
        algo:w2v;dataset:googlenews;dims:300;fold:0;unit:token"
v = wbc.get_vectors(wecs, {}, for_input=[["theory", "computation"]], raw=False, as_tuple=True)
```

Retrieving embedding vectors from several WECs.

More often, however, input text is *raw* and needs to be preprocessed into smaller units before word vectors can be retrieved. WOMBAT acknowledges the importance of preprocessing by providing a two-level mechanism for directly integrating user-defined preprocessing code. The first, obligatory level handles the actual preprocessing by piping each raw input line through a `process(...)` method. User-defined Python code implementing this method is directly inserted into the WOMBAT database. When vectors for raw input text are to be retrieved (`raw=True`), the correct preprocessing for each WEC is automatically applied in the background. While each WEC in WOMBAT could have its own preprocessing, the expected input format for many WECs (e.g. GloVe) is almost identical. Only `glove.840B.300d.txt`, e.g., is case-sensitive, while the others are not. This difference, encoded in the WOMBAT meta data as `fold:0` and `fold:1`, is accounted for automatically during preprocessing. Similarly, some WECs might exist in both an unstemmed and a stemmed variant, which can be distinguished by the values `token` and `stem` in the `unit` attribute. These values are also evaluated during preprocessing. The second, optional processing level analyses the token sequence produced by the first level and joins into phrases those adjacent tokens for which vocabulary items exist in the WEC. Currently this is done by a `gensim.models.phrases.Phraser` object, which initially needs to be trained on the tokenized textual raw data *before* WEC training, and which then needs to be applied to this data in order to enrich it with phrase information.

WOMBATs `get_vectors(...)` method returns data as a generic, nested Python data structure. Basically, it is a list containing one two-item tuple for every WEC, where the first item is the WEC identifier, and the second item is a nested structure containing the actual result, including the raw and preprocessed input and a list of `<word, vector>` tuples. By default, the ordering of this tuple list is undefined, but input ordering can optionally be maintained (`in_order=True`). For most tasks, however, ordering is irrelevant, which is why the more efficient `in_order=False` is the default.

[6] We currently use SQLite (`https://sqlite.org`).

```
[                                                  # top-level result container
 [                                                 # start of result for first WEC
  (
   'algo:glove;dataset:6b;dims:50;fold:1;unit:token', # normalized WEC identifier
   [
    (
     [],                                          # raw input as supplied to for_input (empty here since raw=False)
     ['theory','computation'],                    # tokens produced by preprocessing (if raw=True)
     [
      (                                           # result tuple for 'theory'
       'theory',                                  # token exactly as used in lookup
       [0.28217, 0.65819001, ... -0.39082, -0.1266]   # vector as NumPy array
      ),
      (                                           # result tuple for 'computation'
       'computation',                             # token exactly as used in lookup
       [-0.25176001, -0.028599, ... 0.31508, 0.25172] # vector as NumPy array
      )
     ]
    )
   ]                                              # end of result for first (and only) input list
  )
 ],                                               # end of result for first (and only) WEC
 ...                                              # potential result for second WEC
]
```

Schematic WOMBAT result format.

3 Sample Use Cases

At this point, the `wombat.analyse` library contains only a few methods (cf. below). Our focus has been on developing a stable, generic, and efficient code base, on top of which more complex and useful functionality (incl. further visualizations, nearest neighbors, etc.) can be implemented.

3.1 Global Sentence Similarity

In order to demonstrate WOMBAT in an actual end-to-end use case, we applied it to a sentence pair similarity ranking task, using the data set from task 1, track 5 of SemEval-2017 (Cer et al., 2017). The data set consists of 250 tab-separated, raw sentence pairs. Since we focus on preprocessing and vector retrieval, we implement a simple baseline approach only, in which sentences are represented as the average vector of their respective word vectors (excluding stop words) and the pairwise distances are computed as cosine distance. The result is a list containing, for each WEC, an ordered list of tuples of the form `<distance, sentence1, sentence2>`. The following code implements the whole process. The distance metric in the `pairwise_distances(...)` method is provided as a parameter, and can be set to any method for computing vector distances (or similarities, in which case the output ordering can be reversed with `reverse=True`).

```
import numpy, scipy.spatial.distance
from wombat import connector as wb_conn
from wombat.analyse import pairwise_distances
wbc = wb_conn(path="/home/user/WOMBAT-data/")
wecs = "algo:glove;dataset:6b;dims:{50,100,200,300};folded:1;unit:token&algo:glove;dataset:42b;dims:300;folded:1;unit:token&\
        algo:glove;dataset:840b;dims:300;folded:0;unit:token&algo:w2v;dataset:googlenews;dims:300;folded:0;unit:token"
infile = "STS.input.track5.en-en.txt"
pp_cache = {}
vecs_1 = wb.get_vectors(wecs, pp_cache, for_input=[numpy.loadtxt(infile, dtype=str, delimiter='\t', usecols=0)], raw=True)
vecs_2 = wb.get_vectors(wecs, pp_cache, for_input=[numpy.loadtxt(infile, dtype=str, delimiter='\t', usecols=1)], raw=True)
pd = pairwise_distances(vecs_1, vecs_2, metric=scipy.spatial.distance.cosine, reverse=False)
```

Global sentence similarity computation with WOMBAT.

The execution time for reading the input file (column 0 and column 1 separately), preprocessing, vector retrieval from seven WECs, vector averaging per sentence, pairwise distance computation, and sorting is under ten seconds on a standard notebook computer.

3.2 Word-Level Sentence Similarity

WOMBAT was originally developed in a research project dealing with scientific publication title similarity, which involved light-weight semantic matching based on WEC-based similarities. Figure 1 shows two sample outputs of WOMBATs `plot_heatmap(...)` method, which accepts as input the generic output vectors produced by `get_vectors(...)`. The two plots show the contribution of phrase-aware preprocessing in the comparison of two publication title strings: the left plot was fed with `<string, vector>` tuples which were created with phrases temporarily disabled, and shows a spurious maximal

similarity for the term *net* in the two titles. The right plot, in contrast, was fed with tuples which were created with phrases enabled, including a separate vector for *Petri net*. The plot shows a more differentiated, still high, but not maximal similarity between *Petri net* and *net*, resulting in a more accurate general representation of the to titles' similarities. Heat maps, of course, are standard visualization, but WOMBAT provides methods for their efficient, large-scale creation.

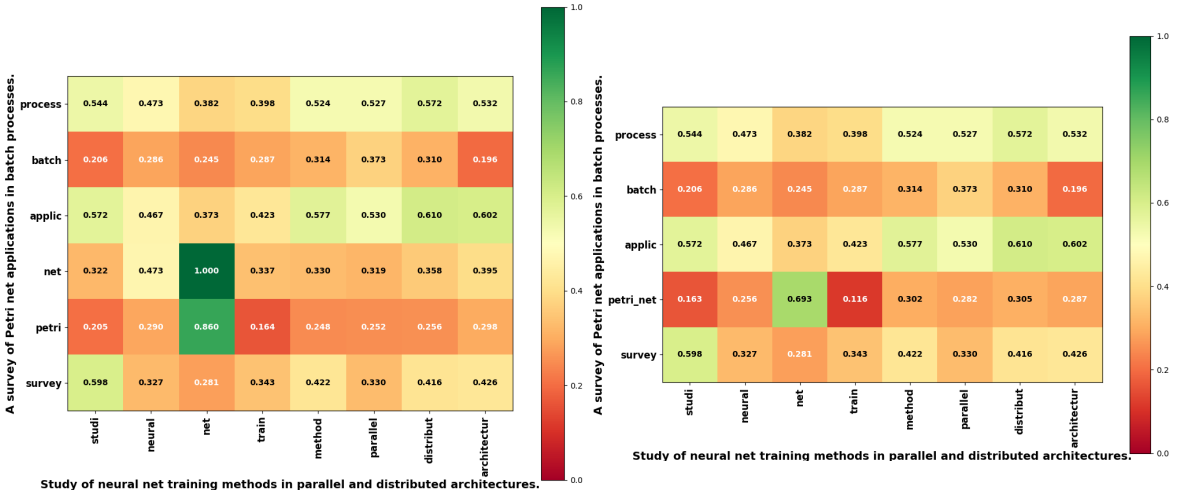

Figure 1: Word-level sentence similarity without (left) and with (right) phrase-aware preprocessing.

Acknowledgements The research described in this paper was conducted in the project *SCAD – Scalable Author Name Disambiguation*, funded in part by the Leibniz Association (grant SAW-2015-LZI-2), and in part by the Klaus Tschira Foundation. We thank the anonymous COLING reviewers for their useful suggestions.

References

Maria Antoniak and David Mimno. 2018. Evaluating the stability of embedding-based word similarities. *TACL*, 6:107–119.

Marco Baroni, Georgiana Dinu, and Germán Kruszewski. 2014. Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors. In *Proceedings of ACL 2014*, pages 238–247.

Daniel M. Cer, Mona T. Diab, Eneko Agirre, Iñigo Lopez-Gazpio, and Lucia Specia. 2017. Semeval-2017 task 1: Semantic textual similarity - multilingual and cross-lingual focused evaluation. *CoRR*, abs/1708.00055.

Antske Fokkens, Marieke van Erp, Marten Postma, Ted Pedersen, Piek Vossen, and Nuno Freire. 2013. Offspring from reproduction problems: What replication failure teaches us. In *Proceedings of ACL 2013*, pages 1691–1701.

Omer Levy, Yoav Goldberg, and Ido Dagan. 2015. Improving distributional similarity with lessons learned from word embeddings. *TACL*, 3:211–225.

Neha Nayak, Gabor Angeli, and Christopher D. Manning. 2016. Evaluating word embeddings using a representative suite of practical tasks. In *Proceedings of the 1st Workshop on Evaluating Vector-Space Representations for NLP*, pages 19–23.

Geir Kjetil Sandve, Anton Nekrutenko, James Taylor, and Eivind Hovig. 2013. Ten simple rules for reproducible computational research. *PLoS Computational Biology*, 9(10).

Tobias Schnabel, Igor Labutov, David M. Mimno, and Thorsten Joachims. 2015. Evaluation methods for unsupervised word embeddings. In *Proceedings of EMNLP 2015*, pages 298–307.

Victoria Stodden and Sheila Miguez. 2014. Best practices for computational science: Software infrastructure and environments for reproducible and extensible research. *Journal of Open Research Software*, 2(1):1–6.

Laura Wendlandt, Jonathan K. Kummerfeld, and Rada Mihalcea. 2018. Factors influencing the surprising instability of word embeddings. In *Proceedings NAACL-HLT 2018*, pages 2092–2102.