itembed

Release 0.5.0

Johan Berdat

Sep 03, 2021

CONTENTS

CHAPTER

ONE

GETTING STARTED

...

CHAPTER

TWO

MATHEMATICAL BACKGROUND

[WFC+17], [BK16]…

2.1 The Pair Paradygm

Item pairs are at the center of [MCCD13] and its derivatives. Instead of processing a whole sequence, only two items are considered at a single step. This section discusses how to select them and what they represent.

2.1.1 Input-Output

The most straightforward way to define an item pair is in the supervised case. The left-hand side is the input (a.k.a. feature item) and the right-hand side is the output (a.k.a. label item).

…

2.1.2 Skip-Gram

…

2.2 Why Negative Sampling?

2.2.1 Softmax Formulation

Let (a, b) a pair of items, where $a \in A$ is the source and $b \in B$ the target. The actual meaning depends on the use case, as discussed above.

The conditional probability of observing b given a is defined by a softmax on all possibilities, as it is a regular multi-class task:

$$P(b \mid a; \mathbf{u}, \mathbf{v}) = \frac{e^{\mathbf{u}_a^T \mathbf{v}_b}}{\sum_{b'} e^{\mathbf{u}_a^T \mathbf{v}_{b'}}}$$

The log-likelihood is therefore defined as:

$$\mathcal{L}(a, b; \mathbf{u}, \mathbf{v}) = -\log P(b \mid a; \mathbf{u}, \mathbf{v}) = -\mathbf{u}_a^T \mathbf{v}_b + \log \sum_{b'} e^{\mathbf{u}_a^T \mathbf{v}_{b'}}$$

$$\frac{\partial}{\partial \mathbf{u}_a} \mathcal{L}(a, b; \mathbf{u}, \mathbf{v}) = -\mathbf{v}_b + \sum_{b'} P(b' \mid a; \mathbf{u}, \mathbf{v}) \mathbf{v}_{b'}$$

However, this implies a summation over every $b' \in B$, which is computationally expensive for large vocabularies.

2.2.2 Noise Contrastive Estimation Formulation

Noise Contrastive Estimation (Gutmann and Hyvärinen [GH10]) is proposed by Mnih and Teh [MT12] as a stable sampling method, to reduce the cost induced by softmax computation. In a nutshell, the model is trained to distinguish observed (positive) samples from random noise. Logistic regression is applied to minimize the negative log-likelihood, i.e. cross-entropy of our training example against the k noise samples:

$$\mathcal{L}(a, b) = -\log P(y = 1 \mid a, b) + k\mathbb{E}_{b' \sim Q}\left[-\log P(y = 0 \mid a, b)\right]$$

To avoid computating the expectation on the whole vocabulary, a Monte Carlo approximation is applied. $B^* \subseteq B$, with $|B^*| = k$, is therefore the set of random samples used to estimate it:

$$\mathcal{L}(a, b) = -\log P(y = 1 \mid a, b) - k \sum_{b' \in B^* \subseteq B} \log P(y = 0 \mid a, b')$$

We are effectively generating samples from two different distributions: positive pairs are sampled from the empirical training set, while negative pairs come from the noise distribution Q.

$$P(y, b \mid a) = \frac{1}{k+1} P(b \mid a) + \frac{k}{k+1} Q(b)$$

Hence, the probability that a sample came from the training distribution:

$$P(y = 1 \mid a, b) = \frac{P(b \mid a)}{P(b \mid a) + kQ(b)}$$

$$P(y = 0 \mid a, b) = 1 - P(y = 1 \mid a, b)$$

However, $P(b \mid a)$ is still defined as a softmax:

$$P(b \mid a; \mathbf{u}, \mathbf{v}) = \frac{e^{\mathbf{u}_a^T \mathbf{v}_b}}{\sum_{b'} e^{\mathbf{u}_a^T \mathbf{v}_{b'}}}$$

Both Mnih and Teh [MT12] and Vaswani et al. [VZFC13] arbitrarily set the denominator to 1. The underlying idea is that, instead of explicitly computing this value, it could be defined as a trainable parameter. Zoph et al. [ZVMK16] actually report that even when trained, it stays close to 1 with a low variance.

Hence:

$$P(b \mid a; \mathbf{u}, \mathbf{v}) = e^{\mathbf{u}_a^T \mathbf{v}_b}$$

The negative log-likelihood can then be computed as usual:

$$\mathcal{L}(a, b; \mathbf{u}, \mathbf{v}) = -\log P(a, b; \mathbf{u}, \mathbf{v})$$

Mnih and Teh [MT12] report that using $k = 25$ is sufficient to match the performance of the regular softmax.

2.2.3 Negative Sampling Formulation

Negative Sampling, popularised by Mikolov et al. [MSC+13], can be seen as an approximation of NCE. As defined previously, NCE is based on the following:

$$P(y = 1 \mid a, b; \mathbf{u}, \mathbf{v}) = \frac{e^{\mathbf{u}_a^T \mathbf{v}_b}}{e^{\mathbf{u}_a^T \mathbf{v}_b} + kQ(b)}$$

Negative Sampling simplifies this computation by replacing $kQ(b)$ by 1. Note that $kQ(b) = 1$ is true when $B^* = B$ and Q is the uniform distribution.

$$P(y = 1 \mid a, b; \mathbf{u}, \mathbf{v}) = \frac{e^{\mathbf{u}_a^T \mathbf{v}_b}}{e^{\mathbf{u}_a^T \mathbf{v}_b} + 1} = \sigma\left(\mathbf{u}_a^T \mathbf{v}_b\right)$$

Hence:

$$P(a, b; \mathbf{u}, \mathbf{v}) = \sigma\left(\mathbf{u}_a^T \mathbf{v}_b\right) \prod_{b' \in B^* \subseteq B} \left(1 - \sigma\left(\mathbf{u}_a^T \mathbf{v}_b\right)\right)$$

$$\mathcal{L}(a, b; \mathbf{u}, \mathbf{v}) = -\log \sigma\left(\mathbf{u}_a^T \mathbf{v}_b\right) - \sum_{b' \in B^* \subseteq B} \log\left(1 - \sigma\left(\mathbf{u}_a^T \mathbf{v}_b'\right)\right)$$

For more details, see Goldberg and Levy's notes [GL14].

To compute the gradient, let us rewrite the loss as:

$$\mathcal{L}(a, b; \mathbf{u}, \mathbf{v}) = -\ell_{a,b,1} - \sum_{b' \in B^* \subseteq B} \ell_{a,b',0}$$

where

$$\ell_{a,b,y} = \log \sigma\left(y - \mathbf{u}_a^T \mathbf{v}_b\right)$$

Then:

$$\begin{aligned} \frac{\partial}{\partial \mathbf{u}_a} \ell(a, b, y) &= \frac{1}{y - \sigma(\mathbf{u}_a^T \mathbf{v}_b)} \left(-\sigma\left(\mathbf{u}_a^T \mathbf{v}_b\right)\left(1 - \sigma\left(\mathbf{u}_a^T \mathbf{v}_b\right)\right)\right) \mathbf{v}_b \\ &= \left(y - \sigma\left(\mathbf{u}_a^T \mathbf{v}_b\right)\right) \mathbf{v}_b \end{aligned}$$

And similarly:

$$\frac{\partial}{\partial \mathbf{v}_b} \ell(a, b, y) = \left(y - \sigma\left(\mathbf{u}_a^T \mathbf{v}_b\right)\right) \mathbf{u}_a$$

2.3 Additional Considerations

2.3.1 Normalization

By setting the denominator to 1, as proposed above, the model essentially learns to self-normalize. However, Devlin et al. [DZH+14] suggest to add a squared error penalty to enforce the equivalence. Andreas and Klein [AK15] even suggest that it should even be sufficient to only normalize a fraction of the training examples and still obtain approximate self-normalising behaviour.

2.3.2 Item distribution balancing

In word2vec, Mikolov et al. [MSC+13] use a subsampling approach to reduce the impact of frequent words. Each word has a probability

$$P(w_i) = 1 - \sqrt{\left(\frac{t}{f(w_i)}\right)}$$

of being discarded, where $f(w_i)$ is its frequency and t a chosen threshold, typically around 10^{-5}.

2.4 References

CHAPTER

THREE

DEVELOPER INTERFACE

This part of the documentation covers the public interface of itembed.

3.1 Preprocessing Tools

A few helpers are provided to clean the data and convert to the expected format.

`itembed.index_batch_stream`(*num_index*, *batch_size*)

Indices generator.

`itembed.pack_itemsets`(*itemsets*, *, *min_count=1*, *min_length=1*)

Convert itemset collection to packed indices.

Parameters

- **itemsets** (*list of list of object*) – List of sets of hashable objects.
- **min_count** (*int, optional*) – Minimal frequency count to be kept.
- **min_length** (*int, optional*) – Minimal itemset length.

Returns

- **labels** (*list of object*) – Mapping from indices to labels.
- **indices** (*int32, num_item*) – Packed index array.
- **offsets** (*int32, num_itemset + 1*) – Itemsets offsets in packed array.

Example

```
>>> itemsets = [
...     ["apple"],
...     ["apple", "sugar", "flour"],
...     ["pear", "sugar", "flour", "butter"],
...     ["apple", "pear", "sugar", "butter", "cinnamon"],
...     ["salt", "flour", "oil"],
... ]
>>> pack_itemsets(itemsets, min_length=2)
(['apple', 'sugar', 'flour', 'pear', 'butter', 'cinnamon', 'salt', 'oil'],
 array([0, 1, 2, 3, 1, 2, 4, 0, 3, 1, 4, 5, 6, 2, 7]),
 array([ 0,  3,  7, 12, 15]))
```

itembed.**prune_itemsets**(*indices*, *offsets*, ***, *mask=None*, *min_length=None*)

Filter packed indices.

Either an explicit mask or a length threshold must be defined.

Parameters

- **indices** (*int32, num_item*) – Packed index array.
- **offsets** (*int32, num_itemset + 1*) – Itemsets offsets in packed array.
- **mask** (*bool, num_itemset*) – Boolean mask.
- **min_length** (*int*) – Minimum length, inclusive.

Returns

- **indices** (*int32, num_item*) – Packed index array.
- **offsets** (*int32, num_itemset + 1*) – Itemsets offsets in packed array.

Example

```
>>> indices = np.array([0, 0, 1, 0, 1, 2, 0, 1, 2, 3])
>>> offsets = np.array([0, 1, 3, 6, 10])
>>> mask = np.array([True, True, False, True])
>>> prune_itemsets(indices, offsets, mask=mask, min_length=2)
(array([0, 1, 0, 1, 2, 3]), array([0, 2, 6]))
```

3.2 Tasks

Tasks are high-level building blocks used to define an optimization problem.

class itembed.**Task**(*learning_rate_scale*)

Abstract training task.

do_batch(*learning_rate*)

Apply training step.

class itembed.**UnsupervisedTask**(*items*, *offsets*, *syn0*, *syn1*, ***, *weights=None*, *num_negative=5*, *learning_rate_scale=1.0*, *batch_size=64*)

Unsupervised training task.

See also:

do_unsupervised_steps()

Parameters

- **items** (*int32, num_item*) – Itemsets, concatenated.
- **offsets** (*int32, num_itemset + 1*) – Boundaries in packed items.
- **indices** (*int32, num_step*) – Subset of offsets to consider.
- **syn0** (*float32, num_label x num_dimension*) – First set of embeddings.
- **syn1** (*float32, num_label x num_dimension*) – Second set of embeddings.
- **weights** (*float32, num_item, optional*) – Item weights, concatenated.

- **num_negative** (*int32, optional*) – Number of negative samples.
- **learning_rate_scale** (*float32, optional*) – Learning rate multiplier.
- **batch_size** (*int32, optional*) – Batch size.

do_batch(*learning_rate*)
Apply training step.

class itembed.**SupervisedTask**(*left_items*, *left_offsets*, *right_items*, *right_offsets*, *left_syn*, *right_syn*, ***, *left_weights=None*, *right_weights=None*, *num_negative=5*, *learning_rate_scale=1.0*, *batch_size=64*)
Supervised training task.

See also:

do_supervised_steps()

Parameters

- **left_items** (*int32, num_left_item*) – Itemsets, concatenated.
- **left_offsets** (*int32, num_itemset + 1*) – Boundaries in packed items.
- **right_items** (*int32, num_right_item*) – Itemsets, concatenated.
- **right_offsets** (*int32, num_itemset + 1*) – Boundaries in packed items.
- **left_syn** (*float32, num_left_label x num_dimension*) – Feature embeddings.
- **right_syn** (*float32, num_right_label x num_dimension*) – Label embeddings.
- **left_weights** (*float32, num_left_item, optional*) – Item weights, concatenated.
- **right_weights** (*float32, num_right_item, optional*) – Item weights, concatenated.
- **num_negative** (*int32, optional*) – Number of negative samples.
- **learning_rate_scale** (*float32, optional*) – Learning rate multiplier.
- **batch_size** (*int32, optional*) – Batch size.

do_batch(*learning_rate*)
Apply training step.

class itembed.**CompoundTask**(**tasks*, *learning_rate_scale=1.0*)
Group multiple sub-tasks together.

Parameters

- ***tasks** (*list of Task*) – Collection of tasks to train jointly.
- **learning_rate_scale** (*float32, optional*) – Learning rate multiplier.

do_batch(*learning_rate*)
Apply training step.

3.3 Training Tools

Embeddings initialization and training loop helpers:

`itembed.initialize_syn`(*num_label*, *num_dimension*, *method='uniform'*)

Allocate and initialize embedding set.

Parameters

- **num_label** (*int32*) – Number of labels.
- **num_dimension** (*int32*) – Size of embeddings.
- **method** (*{"uniform", "zero"}, optional*) – Initialization method.

Returns syn – Embedding set.

Return type float32, num_label x num_dimension

`itembed.train`(*task*, ***, *num_epoch=10*, *initial_learning_rate=0.025*, *final_learning_rate=0.0*)

Train loop.

Learning rate decreases linearly, down to zero.

Keyboard interruptions are silently captured, which interrupt the training process.

A progress bar is shown, using `tqdm`.

Parameters

- **task** (`Task`) – Top-level task to train.
- **num_epoch** (*int*) – Number of passes across the whole task.
- **initial_learning_rate** (*float*) – Maximum learning rate (inclusive).
- **final_learning_rate** (*float*) – Minimum learning rate (exclusive).

3.4 Postprocessing Tools

Once embeddings are trained, some methods are provided to normalize and use them.

`itembed.softmax`(*x*)

Compute softmax.

`itembed.norm`(*x*)

L_2 norm.

`itembed.normalize`(*x*)

L_2 normalization.

3.5 Low-Level Optimization Methods

At its core, itembed is a set of optimized methods.

`itembed.expit`(*x*)

Compute logistic activation.

`itembed.do_step`(*left*, *right*, *syn_left*, *syn_right*, *tmp_syn*, *num_negative*, *learning_rate*)

Apply a single training step.

Parameters

- **left** (*int32*) – Left-hand item.
- **right** (*int32*) – Right-hand item.
- **syn_left** (*float32, num_left x num_dimension*) – Left-hand embeddings.
- **syn_right** (*float32, num_right x num_dimension*) – Right-hand embeddings.
- **tmp_syn** (*float32, num_dimension*) – Internal buffer (allocated only once, for performance).
- **num_negative** (*int32*) – Number of negative samples.
- **learning_rate** (*float32*) – Learning rate.

`itembed.do_supervised_steps`(*left_itemset*, *right_itemset*, *left_weights*, *right_weights*, *left_syn*, *right_syn*, *tmp_syn*, *num_negative*, *learning_rate*)

Apply steps from two itemsets.

This is used in a supervised setting, where left-hand items are features and right-hand items are labels.

Parameters

- **left_itemset** (*int32, left_length*) – Feature items.
- **right_itemset** (*int32, right_length*) – Label items.
- **left_weights** (*float32, left_length*) – Feature item weights.
- **right_weights** (*float32, right_length*) – Label item weights.
- **left_syn** (*float32, num_left_label x num_dimension*) – Feature embeddings.
- **right_syn** (*float32, num_right_label x num_dimension*) – Label embeddings.
- **tmp_syn** (*float32, num_dimension*) – Internal buffer (allocated only once, for performance).
- **num_negative** (*int32*) – Number of negative samples.
- **learning_rate** (*float32*) – Learning rate.

`itembed.do_unsupervised_steps`(*itemset*, *weights*, *syn0*, *syn1*, *tmp_syn*, *num_negative*, *learning_rate*)

Apply steps from a single itemset.

This is used in an unsupervised setting, where co-occurrence is used as a knowledge source. It follows the skip-gram method, as introduced by Mikolov et al.

For each item, a single random neighbor is sampled to define a pair. This means that only a subset of possible pairs is considered. The reason is twofold: training stays in linear complexity w.r.t. itemset lengths and large itemsets do not dominate smaller ones.

Itemset must have at least 2 items. Length is not checked, for efficiency.

Parameters

- **itemset** (*int32, length*) – Items.
- **weights** (*float32, length*) – Item weights.
- **syn0** (*float32, num_label x num_dimension*) – First set of embeddings.
- **syn1** (*float32, num_label x num_dimension*) – Second set of embeddings.
- **tmp_syn** (*float32, num_dimension*) – Internal buffer (allocated only once, for performance).
- **num_negative** (*int32*) – Number of negative samples.
- **learning_rate** (*float32*) – Learning rate.

itembed.**do_supervised_batch**(*left_items, left_weights, left_offsets, left_indices, right_items, right_weights, right_offsets, right_indices, left_syn, right_syn, tmp_syn, num_negative, learning_rate*)

Apply supervised steps from multiple itemsets.

See also:

do_supervised_steps()

Parameters

- **left_items** (*int32, num_left_item*) – Itemsets, concatenated.
- **left_weights** (*float32, num_left_item*) – Item weights, concatenated.
- **left_offsets** (*int32, num_itemset + 1*) – Boundaries in packed items.
- **left_indices** (*int32, num_step*) – Subset of offsets to consider.
- **right_items** (*int32, num_right_item*) – Itemsets, concatenated.
- **right_weights** (*float32, num_right_item*) – Item weights, concatenated.
- **right_offsets** (*int32, num_itemset + 1*) – Boundaries in packed items.
- **right_indices** (*int32, num_step*) – Subset of offsets to consider.
- **left_syn** (*float32, num_left_label x num_dimension*) – Feature embeddings.
- **right_syn** (*float32, num_right_label x num_dimension*) – Label embeddings.
- **tmp_syn** (*float32, num_dimension*) – Internal buffer (allocated only once, for performance).
- **num_negative** (*int32*) – Number of negative samples.
- **learning_rate** (*float32*) – Learning rate.

itembed.**do_unsupervised_batch**(*items, weights, offsets, indices, syn0, syn1, tmp_syn, num_negative, learning_rate*)

Apply unsupervised steps from multiple itemsets.

See also:

do_unsupervised_steps()

Parameters

- **items** (*int32, num_item*) – Itemsets, concatenated.
- **weights** (*float32, num_item*) – Item weights, concatenated.

- **offsets** (*int32, num_itemset + 1*) – Boundaries in packed items.
- **indices** (*int32, num_step*) – Subset of offsets to consider.
- **syn0** (*float32, num_label x num_dimension*) – First set of embeddings.
- **syn1** (*float32, num_label x num_dimension*) – Second set of embeddings.
- **tmp_syn** (*float32, num_dimension*) – Internal buffer (allocated only once, for performance).
- **num_negative** (*int32*) – Number of negative samples.
- **learning_rate** (*float32*) – Learning rate.

BIBLIOGRAPHY

[AK15] Jacob Andreas and Dan Klein. When and why are log-linear models self-normalizing? In Rada Mihalcea, Joyce Yue Chai, and Anoop Sarkar, editors, *HLT-NAACL*, 244–249. The Association for Computational Linguistics, 2015. URL: http://dblp.uni-trier.de/db/conf/naacl/naacl2015.html#AndreasK15.

[BK16] Oren Barkan and Noam Koenigstein. Item2vec: neural item embedding for collaborative filtering. 2016. cite arxiv:1603.04259. URL: http://arxiv.org/abs/1603.04259.

[DZH+14] Jacob Devlin, Rabih Zbib, Zhongqiang Huang, Thomas Lamar, Richard Schwartz, and John Makhoul. Fast and robust neural network joint models for statistical machine translation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 1370–1380. Baltimore, Maryland, June 2014. Association for Computational Linguistics. URL: http://www.aclweb.org/anthology/P14-1129.

[GL14] Yoav Goldberg and Omer Levy. Word2vec explained: deriving mikolov et al.'s negative-sampling word-embedding method. 2014. cite arxiv:1402.3722. URL: http://arxiv.org/abs/1402.3722.

[GH10] Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: a new estimation principle for unnormalized statistical models. In Yee Whye Teh and D. Mike Titterington, editors, *AISTATS*, volume 9 of JMLR Proceedings, 297–304. JMLR.org, 2010. URL: http://dblp.uni-trier.de/db/journals/jmlr/jmlrp9.html#GutmannH10.

[MCCD13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *CoRR*, 2013. URL: http://dblp.uni-trier.de/db/journals/corr/corr1301.html#abs-1301-3781.

[MSC+13] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In C.J.C. Burges, L. Bottou, M. Welling, Z. Ghahramani, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 26*, pages 3111–3119. 2013. URL: http://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.

[MT12] Andriy Mnih and Yee Whye Teh. A fast and simple algorithm for training neural probabilistic language models. In *ICML*. icml.cc / Omnipress, 2012. URL: http://dblp.uni-trier.de/db/conf/icml/icml2012.html#MnihT12.

[VZFC13] Ashish Vaswani, Yinggong Zhao, Victoria Fossum, and David Chiang. Decoding with large-scale neural language models improves translation. In *EMNLP*, 1387–1392. ACL, 2013. URL: http://dblp.uni-trier.de/db/conf/emnlp/emnlp2013.html#VaswaniZFC13.

[WFC+17] Ledell Wu, Adam Fisch, Sumit Chopra, Keith Adams, Antoine Bordes, and Jason Weston. Starspace: embed all the things! 2017. cite arxiv:1709.03856. URL: http://arxiv.org/abs/1709.03856.

[ZVMK16] Barret Zoph, Ashish Vaswani, Jonathan May, and Kevin Knight. Simple, fast noise-contrastive estimation for large rnn vocabularies. In Kevin Knight, Ani Nenkova, and Owen Rambow, editors, *HLT-NAACL*, 1217–1222. The Association for Computational Linguistics, 2016. URL: http://dblp.uni-trier.de/db/conf/naacl/naacl2016.html#ZophVMK16.

PYTHON MODULE INDEX

i

INDEX