CMPUT 692 Paper Presentation:

A simple but tough-to-beat baseline for sentence embeddings

Sarah Davis, Nov 25th 2020

Presentation Outline

1. Introduction to paper
2. Background / Related work
3. Methods
4. Experimental Results
5. Discussion
6. Questions

Introduction to Paper

A simple but tough-to-beat baseline for sentence embeddings. Arora, Liang, and Ma (2017).

Problem

- The use of neural networks has exploded in the last few years for calculating complex embeddings of words
- This has motivated the use of neural networks for representing longer pieces of text (e.g., LSTM or GRU RNNs)
- However, it has been shown that these complex methods can be outperformed by more simple methods[1], especially in the context of transfer learning

[1] Weiting et al. 2016

Paper Contributions

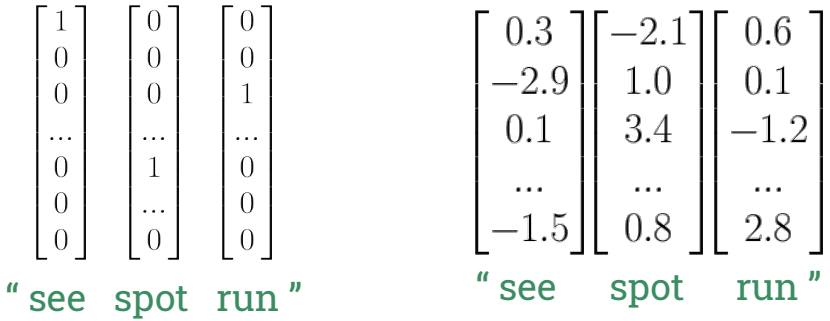
- *A simple but tough-to-beat baseline for sentence embeddings* seeks to improve upon simple sentence representation techniques
- The authors describe a method that is unsupervised and uses pre-trained word embeddings to represent sentences and beats sophisticated architectures in textual similarity tasks

Background Knowledge and Related Work

A simple but tough-to-beat baseline for sentence embeddings. Arora, Liang, and Ma (2017).

Word Embeddings

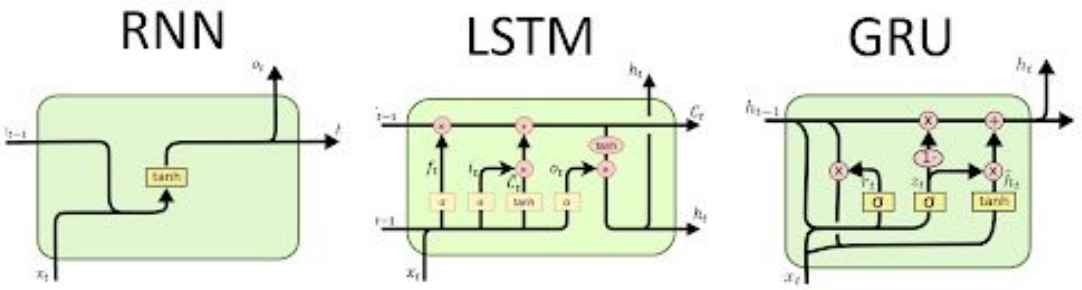
- Word embedding describes the representation of a token as a fixed length vector
- These vectors can be sparse or dense



Document Embeddings

- Document embedding is a term used to describe representing sequences of words as one vector (focus of this paper)
- Vary greatly in simplicity, normally rely on word vectors

$$\sum \prod$$



Inspiration for this paper

$$g_{\text{proj}}(x) = W_p \left(\frac{1}{n} \sum_{i}^{n} W_w^{x_i} \right) + b$$

- Previously[1], it was shown that one can learn paraphrastic sentence embeddings by starting with pre-trained word embeddings, then training a word-averaging model
- Used the Paraphrase Database[2] for training (contains pairs of sentences with similar/same meanings)
- A simple average did not work well, meaning supervision is mandatory for good results

[1]Wieting et al. 2016, [2]Ganitkevitch et al. 2013

Principal Component Analysis

- PCA allows the creation of “summary indices” (principal components) for large data collections, capture important patterns in the data
- Based on Singular Value Decomposition (SVD)
  - Decompose matrix $\mathbf{X}_{n\times p}$ into $\mathbf{U}_{n\times n}\mathbf{\Sigma}_{n\times p}\mathbf{V}^{\mathrm{T}}{}_{p\times p}$ where $\mathbf{U}^{\mathrm{T}}\mathbf{U}=\mathbf{I}_{n\times n}$ and $\mathbf{V}^{\mathrm{T}}\mathbf{V}=\mathbf{I}_{p\times p}$
- From here, the “principal components” can be determined by projecting the data $\mathbf{X}$ onto their “principal axes” $\mathbf{V}$, AKA $(\mathbf{XV})_{n\times p}$

Methods

A simple but tough-to-beat baseline for sentence embeddings. Arora, Liang, and Ma (2017).

Overview



- This paper hinges on the assumptions that :
  - More important words appear less frequently
  - The most common patterns may be semantically meaningless
- Therefore, the authors propose a new sentence embedding method, which involves two steps:
  - Weighting the words inversely on their commonness
  - Common component removal on the sentence vector

Computing Sentence Embeddings

$p(w)$ = estimated from corpus
α = experimentally determined
v_w = precomputed word vectors

Algorithm 1 Sentence Embedding

Input: Word embeddings $\{v_w : w \in \mathcal{V}\}$, a set of sentences $\mathcal{S}$, parameter a and estimated probabilities $\{p(w) : w \in \mathcal{V}\}$ of the words.
Output: Sentence embeddings $\{v_s : s \in \mathcal{S}\}$

1: **for all** sentence s in $\mathcal{S}$ **do**
2: $\quad v_s \leftarrow \frac{1}{|s|} \sum_{w \in s} \frac{a}{a+p(w)} v_w$
3: **end for**
4: Form a matrix X whose columns are $\{v_s : s \in \mathcal{S}\}$, and let u be its first singular vector
5: **for all** sentence s in $\mathcal{S}$ **do**
6: $\quad v_s \leftarrow v_s - uu^\top v_s$
7: **end for**

Computing Sentence Embeddings

$p(w)$ = estimated from corpus
α = experimentally determined
v_w = precomputed word vectors

Algorithm 1 Sentence Embedding

Input: Word embeddings $\{v_w : w \in \mathcal{V}\}$, a set of sentences $\mathcal{S}$, parameter a and estimated probabilities $\{p(w) : w \in \mathcal{V}\}$ of the words.
Output: Sentence embeddings $\{v_s : s \in \mathcal{S}\}$

1: **for all** sentence s in $\mathcal{S}$ **do**
2: $\quad v_s \leftarrow \frac{1}{|s|} \sum_{w \in s} \frac{a}{a+p(w)} v_w$
3: **end for**
4: Form a matrix X whose columns are $\{v_s : s \in \mathcal{S}\}$, and let u be its first singular vector
5: **for all** sentence s in $\mathcal{S}$ **do**
6: $\quad v_s \leftarrow v_s - uu^\top v_s$
7: **end for**

α / (α + $p(w)$), AKA "smooth inverse frequency"

Computing Sentence Embeddings

$p(w)$ = estimated from corpus
α = experimentally determined
v_w = precomputed word vectors

Algorithm 1 Sentence Embedding

Input: Word embeddings $\{v_w : w \in \mathcal{V}\}$, a set of sentences $\mathcal{S}$, parameter a and estimated probabilities $\{p(w) : w \in \mathcal{V}\}$ of the words.
Output: Sentence embeddings $\{v_s : s \in \mathcal{S}\}$

1: **for all** sentence s in $\mathcal{S}$ **do**
2: $\quad v_s \leftarrow \frac{1}{|s|} \sum_{w \in s} \frac{a}{a+p(w)} v_w$
3: **end for**
4: Form a matrix X whose columns are $\{v_s : s \in \mathcal{S}\}$, and let u be its first singular vector
5: **for all** sentence s in $\mathcal{S}$ **do**
6: $\quad v_s \leftarrow v_s - uu^\top v_s$
7: **end for**

projection of v_s onto first principal component

Theoretical Justification

- So we have this idea for how to make these sentences embeddings.
  - How do we know it is theoretically sound?
- The authors showed why doing this reflects how real-world sentences are generated by showing that the MLE sentence embeddings from a thoughtful generative text model matches their simple embeddings

Baseline Generative Model

$$\Pr[w \text{ emitted at time } t \mid c_t] \propto \exp\left(\langle c_t, v_w \rangle\right)$$

- Arora et al. (2017) drew from a previous paper[1] of theirs to define a baseline generative model for text
- Treat corpus generation as a dynamic process:
  - t^{th} word produced at step t
  - Driven by a random walk of a discourse vector, $c_t \in \Re^d$
  - Each word has a (latent) representation as well, $v_w \in \Re^d$
  - The inner product of these two vectors captures correlations between the discourse and the word

[1] Arora et al. 2016

Improved Generative Model

$$\Pr[w \text{ emitted at time } t \mid c_t] \propto \exp\left(\langle c_t, v_w \rangle\right)$$



- The authors sought to improve their previous random walk model to more realistically model sentences
- Added some simplifying assumptions to speed up calculation
  - Discourse vector for each sentence, $\mathbf{c}_s \in \Re^d$ (as opposed to each timestep/word, $\mathbf{c}_t$)
- Added two types of "smoothing term" — accounts for the fact:
  - Words can appear out of context
  - Words can appear because they are syntax-related

Smoothing Term α

$$\Pr[w \text{ emitted in sentence } s \mid c_s] = \alpha p(w) + (1-\alpha)\frac{\exp\left(\langle \tilde{c}_s, v_w \rangle\right)}{Z_{\tilde{c}_s}}$$

- α is a small weighting term (experimentally around 0.0001)
- $p(w)$ is the unigram probability of a word appearing in the entire corpus

Smoothing Term β

$$\Pr[w \text{ emitted in sentence } s \mid c_s] = \alpha p(w) + (1-\alpha)\frac{\exp\left(\langle \tilde{c}_s, v_w \rangle\right)}{Z_{\tilde{c}_s}}$$

$$\text{where } \tilde{c}_s = \beta c_0 + (1-\beta)c_s, \quad c_0 \perp c_s,$$

$$Z_{\tilde{c}_s} = \sum_{w \in \mathcal{V}} \exp\left(\langle \tilde{c}_s, v_w \rangle\right)$$

- c_0 is the “common discourse vector”, serves as a correction for most frequent discourse (normally related to syntax)
- Z_c serves as a normalizing constant

Maximum Likelihood Estimate

$$f_w(\tilde{c}_s) = \log\left[\alpha p(w) + (1-\alpha)\frac{\exp\left(\langle v_w, \tilde{c}_s\rangle\right)}{Z}\right]$$

Denote as log-likelihood of a sentence **s**



$$\arg\max \sum_{w\in s} f_w(\tilde{c}_s) \propto \sum_{w\in s} \frac{a}{p(w)+a} v_w, \text{ where } a = \frac{1-\alpha}{\alpha Z}$$

Experimental Results

A simple but tough-to-beat baseline for sentence embeddings. Arora, Liang, and Ma (2017).

Experimental Set-Up

- The authors tested their sentence embeddings using the Semantic Textual Similarity Tasks (2012-2015) (+ some others) against three types of sentence embedding methods:
  a. Unsupervised (skip-thought[1], avg-GloVe[2], TF-IDF-GloVe)
  b. Semi-Supervised (avg-PARAGRAMSL999[3])
  c. Supervised (PP[4], PP-Proj[4], DAN[5], RNN, iRNN, LSTM (o.g.)[6], LSTM(no)
- They then test the sentence embeddings as input to other supervised learning tasks

[1]Kiros et al. 2015, [2]Pennington et al. 2014, [3]Wieting et al. 2015, [4]Wieting et al. 2016, [5]Iyyer et al. 2015, [6]Gers et al. 2002

STS Tasks (2012 - 2015) + more

	Results collected from (Wieting et al., 2016) except tfidf-GloVe											Our approach	
Supervised or not	Su.							Un.			Se.	Un.	Se.
Tasks	PP	PP -proj.	DAN	RNN	iRNN	LSTM (no)	LSTM (o.g.)	ST	avg-GloVe	tfidf-GloVe	avg-PSL	GloVe +WR	PSL +WR
STS'12	58.7	**60.0**	56.0	48.1	58.4	51.0	46.4	30.8	52.5	58.7	52.8	56.2	59.5
STS'13	55.8	56.8	54.2	44.7	56.7	45.2	41.5	24.8	42.3	52.1	46.4	56.6	**61.8**
STS'14	70.9	71.3	69.5	57.7	70.9	59.8	51.5	31.4	54.2	63.8	59.5	68.5	**73.5**
STS'15	75.8	74.8	72.7	57.2	75.6	63.9	56.0	31.0	52.7	60.6	60.0	71.7	**76.3**
SICK'14	71.6	71.6	70.7	61.2	71.2	63.9	59.0	49.8	65.9	69.4	66.4	72.2	**72.9**
Twitter'15	52.9	52.8	**53.7**	45.1	52.9	47.6	36.1	24.7	30.3	33.8	36.3	48.0	49.0

Table 1: Experimental results (Pearson's $r \times 100$) on textual similarity tasks. The highest score in each row is in boldface. The methods can be supervised (denoted as Su.), semi-supervised (Se.), or unsupervised (Un.). "GloVe+WR" stands for the sentence embeddings obtained by applying our method to the GloVe word vectors; "PSL+WR" is for PSL word vectors. See the main text for the description of the methods.

Supervised Learning Tasks

	PP	DAN	RNN	LSTM (no)	LSTM (o.g.)	skip-thought	Ours
similarity (SICK)	84.9	85.96	73.13	85.45	83.41	85.8	**86.03**
entailment (SICK)	83.1	84.5	76.4	83.2	82.0	-	**84.6**
sentiment (SST)	79.4	83.4	86.5	86.6	**89.2**	-	82.2

Table 2: Results on similarity, entailment, and sentiment tasks. The sentence embeddings are computed unsupervisedly, and then used as features in downstream supervised tasks. The row for similarity (SICK) shows Pearson's $r \times 100$ and the last two rows show accuracy. The highest score in each row is in boldface. Results in Column 2 to 6 are collected from (Wieting et al., 2016), and those in Column 7 for skip-thought are from (Lei Ba et al., 2016).

Discussion

A simple but tough-to-beat baseline for sentence embeddings. Arora, Liang, and Ma (2017).

Interesting Points

- Sentiment classification didn't do as well as similarity and entailment… why is this?
  - "antonym problem" — "good" and "bad" are used in similar contexts but they mean the opposite of each other
  - Because of the weighted average scheme, words like "not" may be down-weighted (α / (α + p(w))

	PP	DAN	RNN	LSTM (no)	LSTM (o.g.)	skip-thought	Ours
similarity (SICK)	84.9	85.96	73.13	85.45	83.41	85.8	**86.03**
entailment (SICK)	83.1	84.5	76.4	83.2	82.0	-	**84.6**
sentiment (SST)	79.4	83.4	86.5	86.6	**89.2**	-	82.2

Interesting Points

- This method doesn't consider word order, but it outperformed algorithms that do (like RNN-based models)
- The authors found that word order does matter by training order-sensitive models with and without order shuffling

Dataset		RNN	LSTM (no)	LSTM (o.g.)
similarity (SICK)	original	73.13	85.45	83.41
	random	54.50	77.24	79.39
entailment (SICK)	original	76.4	83.2	82.0
	random	61.7	78.2	81.0
sentiment (SST)	original	86.5	86.6	89.2
	random	84.2	82.9	84.1

Drawbacks

- High reliance on pre-trained word embeddings
- Only reported impressive results when using semi-supervised word vectors (PSL)
  - If good pre-trained vectors you need are not available, this may not be a viable option
- If there was some way to take into account word order, the results may be even better

Our approach	
Un.	Se.
GloVe +WR	PSL +WR
56.2	59.5
56.6	**61.8**
68.5	**73.5**
71.7	**76.3**
72.2	**72.9**
48.0	49.0

How does this relate?

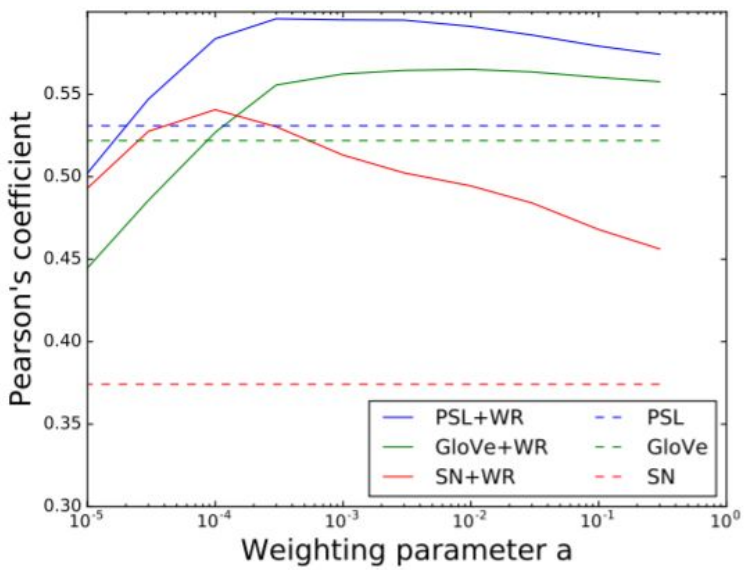
1. SIF vs. TF-IDF
2. Measures of textual similarity
3. MLE
4. Embeddings

Thank you!

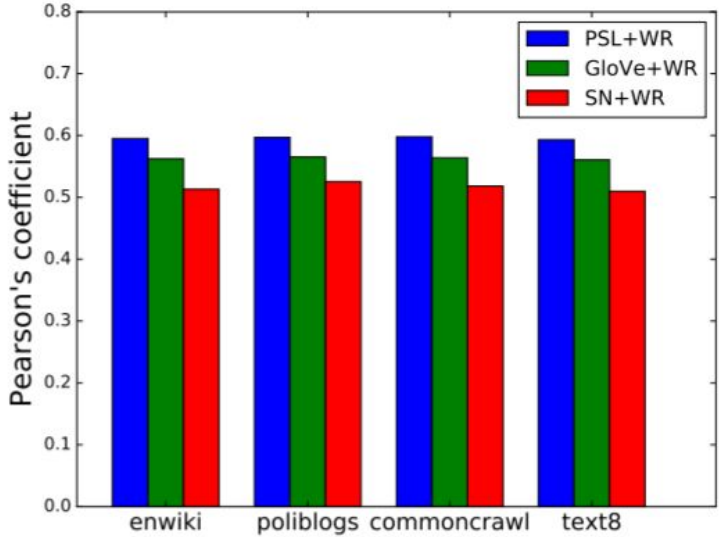
Questions?

Extras

We also note that the top singular vectors c_0 of the datasets seem to roughly correspond to the syntactic information or common words. For example, closest words (by cosine similarity) to c_0 in the SICK dataset are "just", "when", "even", "one", "up", "little", "way", "there", "while", and "but."



(a)



(b)

Figure 2: Effect of weighting scheme in our method on the average performance on STS 2012 tasks. Best viewed in color. (a) Performance v.s. weighting parameter a. Three types of word vectors (PSL, GloVe, SN) are tested using $p(w)$ estimated on the enwiki dataset. The best performance is usually achieved at $a = 10^{-3}$ to $a = 10^{-4}$. (b) Performance v.s. datasets used for estimating $p(w)$. Four datasets (enwiki, poliblogs, commoncrawl, text8) are used to estimate $p(w)$ which is then used in our method. The parameter a is fixed to be 10^{-3}. The performance is almost the same for different settings.

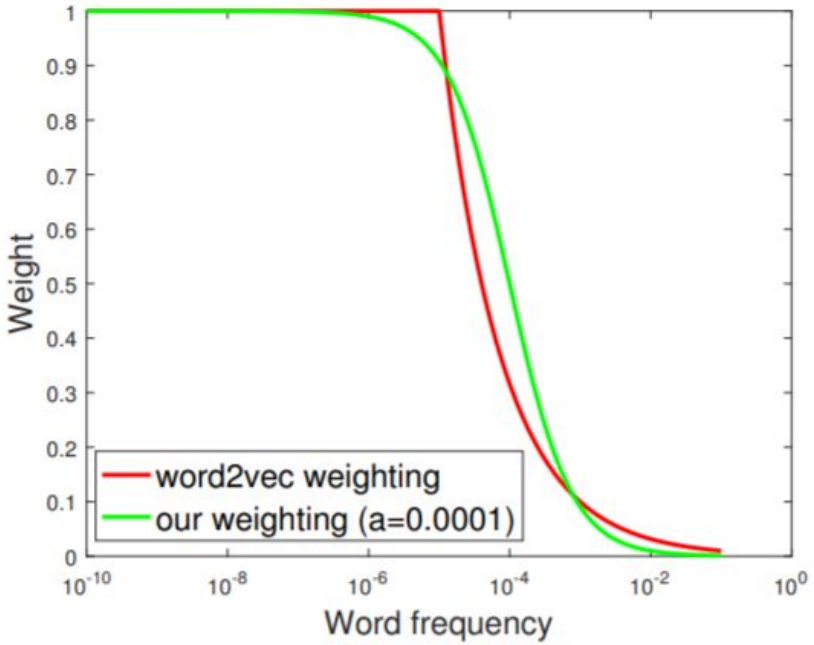


Figure 1: The subsampling probabilities in word2vec are similar to our weighting scheme.

Word2Vec

- Word2Vec[1] is one example of a way to calculate word embeddings, vectors for similar words should appear similar
- Main idea: for each word, from the training corpus, calculate the probability that each other word appears in its context

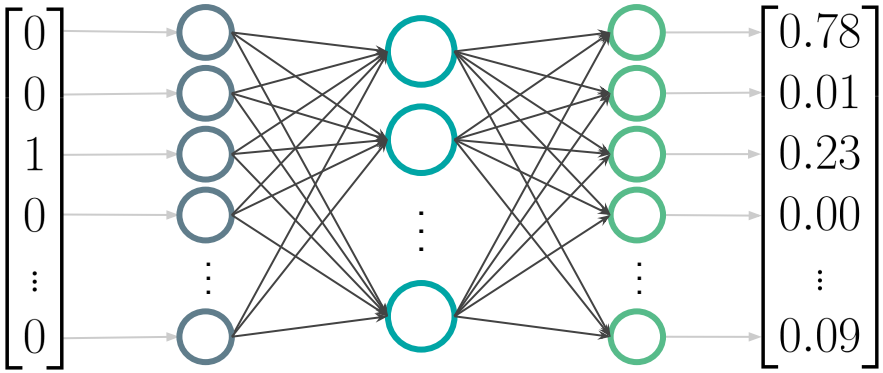
$$word_i = \begin{bmatrix} p(w_{1,i}) & p(w_{2,i}) & & p(w_{i,i}) & & p(w_{n,i}) \\ 0.21 & 0.98 & \dots & 0.06 & \dots & 0.10 \end{bmatrix}$$

[1]Mikolov et al. 2013

Word2Vec

- Train a single layer neural network to predict these probability vectors from the one hot encoded vectors:

$$word_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$



$$\begin{bmatrix} 0.78 \\ 0.01 \\ 0.23 \\ 0.00 \\ \vdots \\ 0.09 \end{bmatrix} \begin{matrix} p(w_{1,3}) \\ p(w_{2,3}) \\ p(w_{3,3}) \\ p(w_{4,3}) \\ \\ p(w_{|V|,3}) \end{matrix}$$

Word2Vec

- Train a single layer neural network to predict these probability vectors from the one hot encoded vectors:

$$word_3 = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

