Natural Language Processing

Skip Gram with Negative Sampling

BEN-GOUMI, Meryem
`meryem.ben-goumi@student-ecp.fr`
BOURIAL, Sarah
`sarah.bourial@student-cs.fr`
RAPPAPORT, Gabrielle
`gabrielle.rappaport@student-ecp.fr`
KAYO KOUOKAM, Josias
`josias.kayo-kouokam@student-ecp.fr`

1 INTRODUCTION

The goal of this exercise is to implement skip-gram with negative-sampling (SGNS) from scratch. To this end, we mainly relied on four references (see ref. 1, 2, 3, 4 in the bibliography). Following Adrien Guille's (2017) algorithm, we implemented an SGNS based on optimising the log-similarity of our training set using a stochastic gradient descent. The data used for this experiment is the Billion Word Corpus (link). This paper details our proceeding. Section 2 highlights the optimisation problem, Section 3 lays out the pre-processing of our data, Section 4 looks at our skip-gram algorithm step-by-step, Section 5 discusses various experiments undertaken using different hyperparameters and Section 6 presents our conclusions.

2 NEGATIVE SAMPLING IN `WORD2VEC`

Skip-Gram with Negative Sampling is a renowned method to learning vectorial representations of words. Let C be the corpus representing our language, and n the number of words in the vocabulary. The approach is based on *(target word, context word)* pairs generated from the corpus C. With regards to these pairs, $U \in \mathbf{R}^{nxm}$ (resp. $V \in \mathbf{R}^{nxm}$) is the matrix of embeddings of target words (resp. context words). As such, each target word (resp. context word) has a vectorial representation of size m that correspond to a row in matrix U (resp. V).
Therefore, the method aims at maximizing the log-likelihood of our set of pairs (w_i, w_j), assuming a probability:

$$p(w_i|w_j, U, V) = \sigma(u_i^T v_j) = \frac{1}{1 + e^{-u_i^T v_j}}$$

Hence, the objective is to solve the optimisation problem (P):

$$(\text{P}) : \underset{U,V}{\text{argmax}} = \sum_{(w_i, w_j)} log(\sigma(u_i^T v_j))$$

However, the problem (P) has a trivial solution: setting all values of matrices U as V equal to 1 leads to a dot product equal to m for each pair words, and therefore a high probability. In order to avoid this trivial solution, the approach suggests to a introduce a **Negative Sampling** element to our problem (P), by adding randomly chosen negative pairs of words. The ratio of positive pairs' number to negative pairs' number is equal to k, a hyper-parameter of our model. The resulting set of states is therefore a set of triplets $(w_i, w_j, \gamma_{i,j})$, with $\gamma_{i,j} = 1 or - 1$ to specify the nature of pair of words. With negative sampling, the optimisation problem (P) becomes:

$$(\text{P'}) : \underset{U,V}{\text{argmax}} = \sum_{(w_i, w_j)} log(\sigma(\gamma_{i,j} u_i^T v_j))$$

In the following sections, we describe the steps followed to implement this approach and the results obtained.

3 PRE-PROCESSING THE DATA

3.1 Tokenisation

Our tokenisation process starts with concatenating all the text in a file located at file path. This is to ensure homogeneity of treatment across the dataset. We split the sentences at each punctuation that ends a sentence i.e. full stop <.>, exclamation <!> and interrogation marks <?> and subsequently deleted that punctuation. Using the Regex library, create sentences tokens thereon used to create our positive and negative pairs.

3.2 Undersampling

The subsampling method we used was based on Goldberg and Levy (2014). The process was made to avoid using the the NLTK library to remove stopwords from the corpus, instead using a weighted function to subsample the corpus and dilute very frequent words i.e. akin to removing stop-words We randomly remove words that are more frequent than some threshold t with a probability of p where f marks the word's corpus frequency (the frequency of each word in the corpus):

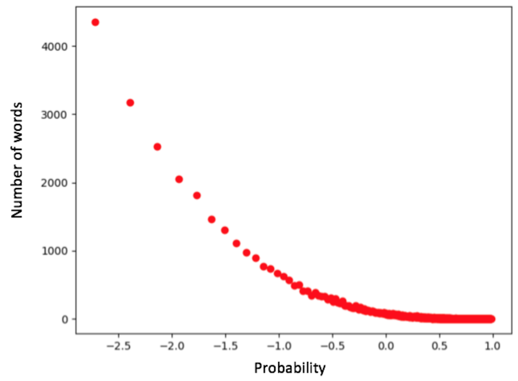
$$p = 1 - \sqrt{\frac{t}{f}}$$

NB: `word2vec`*'s initial code relies on a slightly different formula than the one mentioned in the paper we used:*

$$p = \frac{f-t}{t} - \sqrt{\frac{t}{f}}$$

The recommended value for t is $t = 10^{-5}$.
An important implementation detail of subsampling in `word2vec` that we made sure to apply in our code is that the random removal of tokens is performed before the processing of word-context pairs (so before running our skip-gram). This basically enlarges the context windowâĂŹs size for many tokens, because they can now reach words that were not in their original l-sized windows.
e.g. the sentence "the smart blonde woman jumped over the tight rope"

might be down to "smart blonde woman jumped tight rope". Here "rope" is now in the two-word context window of "jumped", even if it was originally 4 tokens away from "jumped".
There appears to be no clear-cut rule for when subsampling helps improve performances, with therefore ran our code with and without it. See Section 5 for a discussion of the results. We set our threshold according to the probability distribution of words in our given sample:



On the graph above, we can see that there is a small number of words with a probability higher than 0.9. We thus looked for the optimal probability threshold, comparing the lists of words to undersample. For a probability threshold of 0.93, we obtained the following list of 47 words, which seems relevant and consistent with a classic list of stop words.

```
words_to_be_undersampled: ['the', 's', 'for', 'and', 'to'
    , 'of', 'they', 'had', 'about', 'on', 'in', 'it' , '
    a', 'that', 'with', 'is',' its', 'have', 'will', '
    said', 'be', 'an', 'at', 'his',' been ',' their ','
    were ',' by ',' was', 'one', 'more', 'would', 'not',
     'or', 'but', 'as',' who ' , 'after', 'has', 'from',
     'we', 'are', 'this', 'he', 'i', 'new', 'which']
```

4 STEP-BY-STEP SKIGRAM ALGORITHM

4.1 Step 1: Initialisation

The objective here is to generate a list of $(w_i, w_j, \gamma_{i,j})$ triplets. In each triplet, w_i is the target word, w_j the context word, and $\gamma_{i,j}$ is equal to $+1$ or -1, depending on whether the target-context pair (w_i, w_j) is positive or negative.
Before generating such triplets, we start by removing from our dataset of sentences all rare words, i.e. words that occur less than *minCount* parameter. The function `pre_initisation` generates this new set of sentences and computes dictionaries keeping track of the number of occurences and the unique indexes of our set of words.
With this new set of cleaned sentences, we can start generating the positive pairs of words: $(w_i, w_j, +1)$. To do so, the function `targetw_contextw_positive_pairs` considers each word in each sentence as a target word, and associates it with context words surrounding it in the sentence. The context words are found after going through words located in the sentence at the left (at most *winSize*) or at the right (at most *winSize*) of our target word. Hence, the parameter *winSize* sets the size of the window sliding through the corpus.
For negative pairs, our function `targetw_contextw_negative_pairs` follows the approach explained in reference [4]. The idea is to build an **adaptative unigram table**, i.e. a table of size n_{table} in which each word w_i is repeated a number of times n_i proportional to $f(w_i)^{3/4}$, with $f(w_i)$ corresponding to the occurrences of w_i in the corpus C:

$$n_i = n_{table} \frac{f(w_(i)^{3/4}}{\sum\limits_{w_j} f(w_(j)^{3/4}}$$

However, we can note that given the formula above, n_i is not an integer. To solve this issue, we replace it with $\lceil n_i \rceil$ with a probability $n_i - \lfloor n_i \rfloor$, and with $\lfloor n_i \rfloor$ the rest of the time. This way, the expected number of times w_i is added to the table is equal to n_i.
Now that the unigram table is built, we can, for each target word in our positive pairs, randomly pick context words from the unigram table.

4.2 Step 2: Training

4.2.1 The approach. The training process closely followed the method given by Adrien Guille and based on Mikolov et. al. (2013). Indeed, we start by randomly initializing the embedding matrices U and V. Then, at each epoch, we randomly shuffle the set of pairs, split it into different batches, and update U and V using a Stochastic Gradient Descent with a constant step size η. It requires computing the gradient of the log-likelihood l:

$$l = \sigma(\gamma_{i,j} u_i^T v_j)$$

Hence, at each epoch, for each pair of words $(w_i, w_j, \gamma_{i,j})$ in our training batch, we update the i^{th} row of U and the j^{th} row of V according to the formulas below:

$$\begin{cases} u_i \leftarrow u_i + \eta \nabla_{u_i} l \\ v_j \leftarrow v_j + \eta \nabla_{v_j} l \end{cases}$$

4.2.2 Computing Gradients. The gradient of l in u_i and in v_j is a vector of $\mathbb{R}^m$:

$$\begin{cases} \nabla_{u_i} l = (\frac{\partial l}{\partial u_{i,1}}, \frac{\partial l}{\partial u_{i,2}}, \dots, \frac{\partial l}{\partial u_{i,m}}) \\ \nabla_{v_j} l = (\frac{\partial l}{\partial v_{j,1}}, \frac{\partial l}{\partial v_{j,2}}, \dots, \frac{\partial l}{\partial v_{j,m}}) \end{cases}$$

As we know the expression of l, we can easily compute the partial derivatives and deduce from it the gradient. Indeed, for $\lambda \in [\![1, m]\!]$:

$$\begin{cases} \frac{\partial l}{\partial u_{i,\lambda}} = \gamma_{ij} v_{j,\lambda} \sigma(-\gamma_{i,j} u_i^T v_j) \\ \frac{\partial l}{\partial v_{j,\lambda}} = \gamma_{ij} u_{i,\lambda} \sigma(-\gamma_{i,j} u_i^T v_j) \end{cases}$$

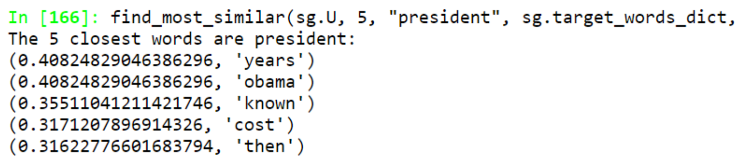
5 EXPERIMENTS

We trialled our model on 1000 sentences from the first data file of the training folder in the Billion Words Corpus. We performed multiple experiments, essentially focusing on the variation of the following parameters (here with our default settings: `negativerate = 5`, `winsize = 5` (window size), `epochs = 5`, `batchsize = 1024` and `stepsize = 0.025`. Our evaluation was two-fold: quantitative - in terms of words similarity scores and algorithm running time; and qualitative - in terms of the quality of similar words found.

5.1 Skip-gram hyperparameters

5.1.1 Window size. Just like subsampling, the window level affects the size of our context: put shortly, it describes the number of words selected on each side of our target word, respectively (Goldberg Levy, 2014). Intuitively, one could think a larger window size gives more context to a word and thus improve the accuracy of similarities. We challenged this idea, as a larger window can also lead to different contexts mixed for a single target word, and thus trained our model on a smaller window to confirm its impact.

5.1.2 Negative rate. The negative rate is the number of negative words generated from a pair when training our modele. Chablani (2017) states that 5-20 words works well for smaller datasets, while large ones can get away with only 2-5 words. We thus performed our skip-gram for three values of negative rate (k): $k = 2$, $k = 5$, $k = 10$. It appears that $k = 10$ was indeed the optimal value in that setting, probably because of our relatively small training sample (1000 sentences). The qualitative results are as follows



```
In [166]: find_most_similar(sg.U, 5, "president", sg.target_words_dict,
The 5 closest words are president:
(0.40824829046386296, 'years')
(0.40824829046386296, 'obama')
(0.35511041211421746, 'known')
(0.3171207896914326, 'cost')
(0.31622776601683794, 'then')
```

5.2 Subsampling

According to Goldberg ad Levy (2014), subsampling can indeed improve performance for some tasks, while decreasing performance for others. We thus performed undersampling on two values of k = 5 and k = 10. Our results with undersampling were better on k = 10, however still underperformed our output with k = 10 when not using undersampling at all. It turned out that using undersampling or increasing the negative rate (k) is a trade-off: indeed, by removing overfrequent words, undersampling effectively increases the window size. Thereby, increasing the negative rate on top of using undersampling might lead to too wide a window, hence resulting in incoherent similarities. We thus established an experimental rule: either increase `negativerate` and/or `winsize`; or use undersampling alone.

5.3 Training hyperparameters

5.3.1 Epochs. Increasing the number of epochs on which our algorthim runs (number of iterations) should improve the learning process and thereby make our embedding better. We therefore made two attempts changing the number of epochs, once with 10 and once with 15 epochs.

5.3.2 Batch size. The batch size is the number of examples used by the algorithm in estimating the error gradient. (Brownlee, 2019) It controls the accuracy of the estimate of the error gradient in the training sample, so that a smaller batch brings in more noise, thereby offering a regularization effect. Hence there is a tension between batch size and the speed and stability of the learning process. By improving the performance of our gradient optimization, it directly affects the performance of our algorithm. We thus tried using a smaller batch (half our default value).

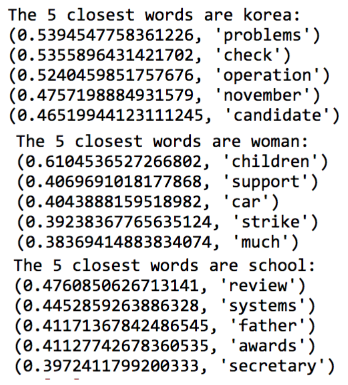
5.3.3 Step size. The step size represents the magnitude of modifications in our gradient descent, the η in our equation above. As we are aiming at finding the optimal parameters for our stochastic gradient, attempted at increasing its precision by reducing this parameter i.e. the steps at which the gradient updates

6 RESULTS

The results of our trials with hyperparameters variations - one at a time, every hyperparameter value was set ceteris paribus of default settings - are given below, taking the word korea as an example:

Parameters	Running time (in minutes)	List of similar words
		korea
Default settings	16’	’announced’, ’school’, ’u’, ’big’, ’such’
Winsize = 2	9	’u’, ’s’, ’school’, ’of’, ’won’
Epoch = 10	52’	’you’, ’high’, ’international’, ’but’,’still’
Epoch =15	65’	’coach’, ’royal’, ’point’, ’still’,’don’
Batch size = 512	19’	’m’, ’school’, ’help’, ’reuters’,’2009’
Step size = 0.01	14’	’the’, ’s’, ’for’, ’and’, ’to’

The window size appears to indeed negatively impact our results when reduced. Reducing the step size led to the same conclusion. However, increasing the number of epochs and reducing the batch size did lead to more accurate words and better probabilities (around 0.6), and less running time, respectively. We thus decided to set these as our hyperparameters in the final code. Keeping in mind the time and computational constraint of running the algorithm on a larger sample of 100 000 sentences, we set back our negative rate to k = 5 in the final version of our code, to meet expectations of it running in less than 48h (we expect it to run in 16h). Our final settings are thus: `winsize = 5`, `negativerate = 5`,`epoch = 10`, `batchsize = 512`, `stepsize = 0.025`, discarding subsampling. Examples of word similarities are given below for the words ’korea’, ’woman’, and ’school’:



```
The 5 closest words are korea:
(0.5394547758361226, 'problems')
(0.5355896431421702, 'check')
(0.5240459851757676, 'operation')
(0.4757198884931579, 'november')
(0.46519944123111245, 'candidate')
 The 5 closest words are woman:
 (0.6104536527266802, 'children')
 (0.4069691018177868, 'support')
 (0.4043888159518982, 'car')
 (0.39238367765635124, 'strike')
 (0.38369414883834074, 'much')
 The 5 closest words are school:
 (0.4760850626713141, 'review')
 (0.4452859263886328, 'systems')
 (0.41171367842486545, 'father')
 (0.41127742678360535, 'awards')
 (0.3972411799200333, 'secretary')
```

7 CONCLUSIONS

In conclusion, this exercise has been extremely formative in terms of algorithmic methods, process implementation and gradient optimization. The task was also intuitively challenging in terms of words' and outcomes' evaluation. Overall, we have come to the conclusion that SGNS surely is an efficient variant to the `word2vec` approach. Although it is not within the remit of this exercise, a useful extension here would be diving into a skip-gram with neural network architecture, or using a different gradient method such as the Riemannian optimization framework i.e. view the optimization of SGNS objective as a problem of searching for a good matrix with the low-rank constraint.

8 BIBLIOGRAPHY

(1) A very well-detailed post: Guille, A. (2017). Skip-Gram with Negative Sampling. [Blog] Mediamining.univ-lyon2.fr. Available at: link [Accessed 8 Feb. 2019].

(2) A paper discussing the most commonly used process of skip-gram negative-sampling: Mikolov T., Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013, pages 3111-3119, 2013. Available at: link.

(3) A further insight into the Mikolov papers: "`word2vec` Explained: Goldberg, Y. and Levy, O. (2019). `word2vec` Explained: Deriving Mikolov et al.'s Negative-Sampling Word-Embedding Method. Cornell University. [online] Available at: link [Accessed 16 Feb. 2019].

(4) A paper explaining the negative sampling process through an adaptative unigram table: Kaji, N. and Kobayash, H. (2017). Incremental Skip-gram Model with Negative Sampling. Association for Computational Linguistics, [online] pp.363-371. Available at: link [Accessed 19 Feb. 2019].

(5) Levy, O. and Goldberg, Y. (2019). Dependency-Based Word Embeddings. Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, [online] 2, pp.302âĂŞ308. Available at: link [Accessed 19 Feb. 2019].

(6) Brownlee, J. (2019). How to Control the Speed and Stability of Training Neural Networks With Gradient Descent Batch Size. [Blog] Better Deep Learning. Available at: link [Accessed 19 Feb. 2019].

(7) Chablani, M. (2017). Word2Vec (skip-gram model): PART 1 - Intuition. [Blog] Towards Data Science. Available at: link [Accessed 21 Feb. 2019].