CentraleSupélec

SKIPGRAM PROJECT

Natural Language Processing-Homework 1

February the 22th 2019

ATTYASSE Flora | BENICHOU Vadim | MICCICHE Carmelo | VIAL Jennifer

CentraleSupélec

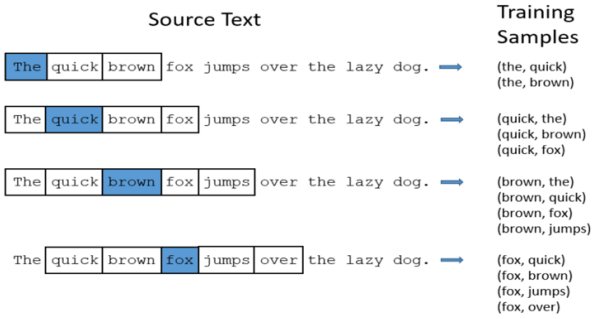
Presentation of the project

The Skipgram project aims at programming an algorithm in Python in order to determine for any given word its embedding and to assess its degree similarity with another word. At the end of the exercise, our code may return for a given word of a given corpus the N-closest ones in terms of cosine similarity. This kind of algorithm is a foundation of Natural Language Processing and has several uses. For instance, through this process, one can translate words, analyze tweets and so on. This report is a sum-up of the creation process of the algorithm, the choices made for the programming part, the obstacles faced and the final results.

Creation of our inputs

The goal of this step is to generate a Matrix of dimension (2XN) with pairs of words. The data imported are texts. To deal with it, one needs to isolate, store sentences and words in a list by proceeding to the tokenization step. To keep the most relevant elements, punctuations are to be removed. For this project, the numbers are also unconsidered. Furthermore, one has to manage the stopwords (a, the, on, of, and, so, on…) or the words that owe a too important frequency. Indeed, they can bias the analysis as they can be misassociated to other words; wrong similarity rates can be generated. At first glance, we tried to remove all the stop words, then we deleted elements that appeared more than 5,000 times in the corpus. We realized that the similarity when stopwords were removed was not as good as the one generated by removing words that appeared more than 5,000 (for instance 'state' was wrongly associated with 'within'). Nevertheless, imposing a fixed amount of words is not optimal neither to the extent that the study aims at building algorithm generalizable to every kind of corpus. For instance, not taking into account words that appear more than 10,000 times in a text of 300 elements is not relevant. Thus, it has been decreed that every word that represents more than 0.2 % (~5700 words in our case) of the total number of elements in the corpus is deleted. The idea of using the Skip-gram Model with Negative Sampling (by Nobuhiro Kaji and Hayato Kobayashi) rate of frequency was considered, nevertheless the results generated were not as good as the ones provided by the percentage method.

Once the tokenization is realized, the pairs are to be generated. Each word of the corpus is associated with its N neighbors to create "positive pairs". Here the window size chosen is N=2; in other word we consider the 2 previous and 2 next words of the target word. Tests with a window size of 4 and of 1 were also conducted, however, size N=2 gave more accurate results. Indeed, 1 was not important enough to update the embeddings correctly and 4 could lead to wrong associations. N negative samples are generated every "positive pair"; each time a target word is matched with one of its N neighbors, it will also be associated to 5 other random words chosen in the corpus. Negative rate of 4 and of 6 were also considered, it does not change the similarities significantly neither. Several options to generate them have been found: either the word associated is picked randomly in the corpus or it is chosen randomly with a rate computed with the Incremental Skip-gram Model with Negative Sampling study realized by Nobuhiro Kaji and Hayato Kobayashi. The second alternative has been chosen.



There are 3 configurations to generate the positive pairs according to the position of the target word:
- When the target word is the first or the last element of the sentence, 2 positive pairs are created (target word associated with the 2 next or previous words).
- When the target word is the second or the last element of the sentence, 3 positive pairs are created (target word associated with the 2 next or previous words and the previous or next word).
- For all the other positions of the target word, the classical process is implemented and 4 pairs are created.

Formula of the SGNS frequency rate: $\frac{f(w)^a}{\sum_{w' \in W} f(w')^a}$

The Optimization Algorithm

This step aims at determining the optimal embedding for each element of the corpus. A dictionary in which each word is matched with a vector of 100 values assigned randomly using the random uniform, is created. The embeddings initialized are multiplied by 1e-3 in order to avoid too heavy embedding values at the end of the training part. The optimization problem is translated though this function.

$$J(\theta) = -\log\left(\frac{1}{1+e^{(-Vc.Vw)}}\right) + \sum_{k=1}^{5} -\log\left(\frac{1}{1+e^{(Vk.Vw)}}\right)$$

The first part of the function corresponds to the optimization though our positive samples and the second to the optimization through the negatives ones. Those formulas model the optimization purpose of minimizing the value between 2 similar word and to maximizing the value between two distant word. By computing the partial derivatives of the functions above, one can deduce the gradients needed to update each vector:

$$V_w = V_w - 0.01 * \frac{\partial J(\theta)}{\partial V_w}$$
$$V_c = V_c - 0.01 * \frac{\partial J(\theta)}{\partial V_c}$$
$$V_k = V_k - 0.01 * \frac{\partial J(\theta)}{\partial V_k}$$

with:

$\frac{\partial J(\theta)}{\partial V_w} = \frac{-V_c.e^{-V_c.V_w}}{1+e^{-V_c.V_w}} + \sum_{k=1}^{5} \frac{V_k.e^{V_k.V_w}}{1+e^{V_k.V_w}}$ and $\frac{\partial J(\theta)}{\partial Vc} = \frac{V_w.e^{-V_c.V_w}}{1+e^{-V_c.V_w}}$ and $\frac{\partial J(\theta)}{\partial V_k} = \sum_{k=1}^{5} \frac{V_w.e^{V_k.V_w}}{1+e^{V_k.V_w}}$

Vc= the vector of the context word
Vw = the vector of our target word
Vk = the vector of our 5 distant word from our target word.

Based on this function, the sigmoid is defined: $Sigmoid(x, y) = \frac{1}{1+e^{-x.y}}$

Firstly, the gradient step chosen was 0.1, nevertheless, this step was too heavy and the output embeddings values were no satisfying (succession of nan), thus after having tested different lower values for the step, 0.01 provides correct outputs.

Different gradients are computed according to the case. If the word vector target is updated by considering a positive sample the gradient formula is the following:

$$Gradient_{positive\ target} = -y\ .Sigmoid(-y, x) = \frac{-y}{1+e^{y.x}}$$

If the word vector target is updated by considering a negative sample the gradient formula is the following:

$$Gradient_{negative\ target} = y\ .Sigmoid(x,y) = \frac{y}{1+e^{-x.y}}$$

If the word vector in negative association is updated the gradient formula is the following:

$$Gradient_{negative\ id} = x\ .Sigmoid(y.x) = \frac{x}{1+e^{-y.x}}$$

Based on it, there are several possibilities to manage the updates; either one decides to actualize only the target word vector at each iteration considering that each context word and negatively associated word will also be update by becoming a target word or one can chose for each iteration to update the target word vector, the context word vector and the negatively associated word vectors. Furthermore, one also has the choice to actualize the vector for each iteration or to update them every N iterations. For this project, the context, target and negative associated word vectors are updated for each iteration regarding the accuracy of the results. At the end of the step each word are assigned to their most significant embedding vectors (annex 4).

The Similarity Assessment

To assess the word embeddings generated above, the cosine similarity needs to be programmed. The formula we opted for is the following:

$$Similarity(W_1.W_2) = \frac{V_{w1}.V_{w2}}{\|V_{w1}\|.\|V_{w2}\|}$$

with V_{w1} and V_{w2} the embedding of word1 and word2

Tests with similar and non-similar words have been conducted. One could clearly notice that the similarity between 'war' and 'crime' is quite high meanwhile the one between 'war' and 'welcomed' is really low which is a good sign concerning the accuracy of our algorithm.

```
1 print('Similarity score:',score_similarity('war','crime',dico_word_vectors))
```

```
Similarity score: 0.9999802568933895
```

```
1 print('Similarity score:',score_similarity('war','welcomed',dico_word_vectors))
```

```
Similarity score: 5.003193620861657e-05
```

Based on the similarity score the most_similar function can be created. The aim is to find the 5 most similar words of a given word.

```
1 print('5 most similars words:',most_similar('conflict',dico_word_vectors,k=5))
```

```
5 most similars words: ['leadership', 'form', 'palestinian', 'authority', 'regional']
```

CentraleSupélec

Annexes:

(1)

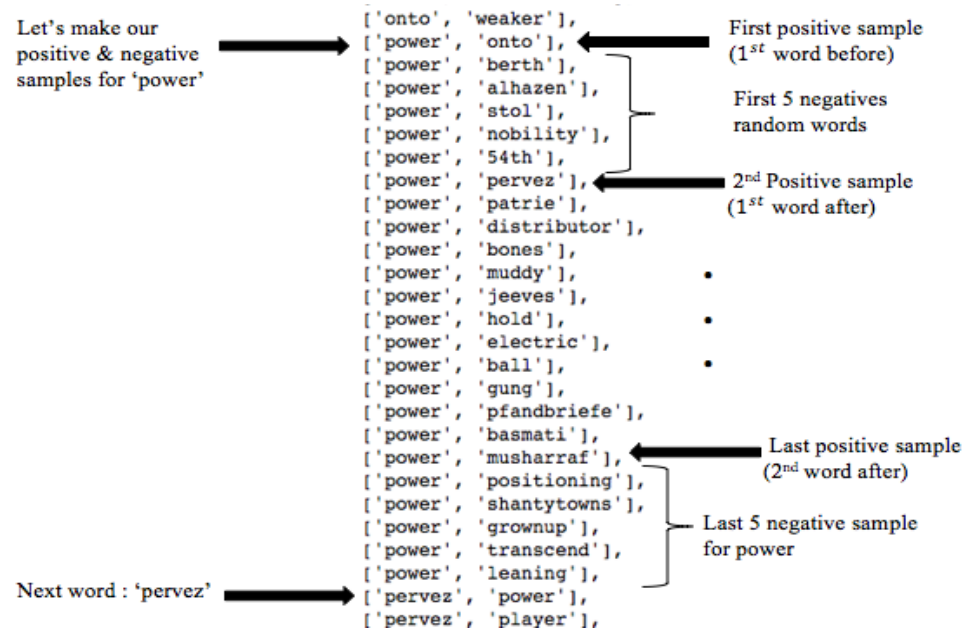
```
{'musharraf': 206,
 'last': 1853,
 'act': 548,
 'desperate': 88,
 'hold': 420,
 'onto': 90,
 'power': 3070,
 'pervez': 33,
 'discarded': 15,
 'pakistan': 760,
 'constitutional': 360,
 'framework': 291,
 'declared': 208,
 'state': 2520,
 'emergency': 159,
 'his': 4970,
 'goal': 394,
 'stifle': 19,
```

Dictionary of frequency screenshot

(2)

```
[0.00013314927505293886,
 0.0006915840290438583,
 0.00027734721535714087,
 7.035587927991866e-05,
 0.00022718275337432246,
 7.155175212417982e-05,
 0.0010099320915639856,
 3.371505293181542e-05,
 1.866409065446332e-05,
 0.0003544447803095227,
 0.00020237891653114606,
 0.0001725274219118419,
 0.00013411763672470823,
 0.0008709411220332296,
 0.0001096443441647621,
 0.0014494581564669048,
 0.00021655118136499902,
 2.2284535786890886e-05,
```

Rate of occurrence list screenshot



Example of list of pairs

(4)

```
{'musharraf': array([-0.00736422, -0.00702025, -0.00716213, -0.00792691, -0.00754438,
        -0.00738188, -0.00634726, -0.00805009, -0.00750925, -0.00645577,
        -0.00691728, -0.00682906, -0.00606316, -0.00646418, -0.00708437,
        -0.00717794, -0.00704516, -0.00798135, -0.00698777, -0.00674746,
        -0.00620511, -0.00760788, -0.00644923, -0.00649717, -0.00604575,
        -0.00726029, -0.00647824, -0.00688187, -0.00661261, -0.00713082,
        -0.0066623 , -0.00692527, -0.00642968, -0.0068394 , -0.00753033,
        -0.00726586, -0.00716559, -0.00693057, -0.00656054, -0.00709434,
        -0.0071374 , -0.00687194, -0.00695037, -0.00631858, -0.00697955,
        -0.00655596, -0.00672758, -0.00668533, -0.00654786, -0.00760563,
        -0.00702952, -0.00732958, -0.00793897, -0.00726895, -0.00679783,
        -0.0064694 , -0.00653671, -0.00620792, -0.00773015, -0.0068146 ,
        -0.00696247, -0.00751157, -0.00623145, -0.00726285, -0.0071455 ,
        -0.00658586, -0.00748089, -0.0067127 , -0.00618987, -0.00668259,
        -0.00750344, -0.00691316, -0.00703926, -0.00758366, -0.00727672,
        -0.00613049, -0.00659215, -0.00656468, -0.00590421, -0.00601999,
        -0.00694309, -0.00703506, -0.00649551, -0.00797484, -0.00750574,
        -0.00712883, -0.00717497, -0.00702586, -0.00603455, -0.00732231,
        -0.0070476 , -0.00730659, -0.00645533, -0.00631185, -0.00694036,
```

Dictionary of word embedding after having trained the algorithm