DL4NLP 2022 — Exercise 5

Jonas Belouadi, Steffen Eger
Natural Language Learning Group (NLLG),
University of Bielefeld,
Summer Semester 2022

To prepare for the tutorial, you can already install the dependencies listed in `requirements.txt` and optionally run `download_models.py`.

Task 1 (15min): BERT based Sentence Embeddings

Sentence embeddings is a general term for techniques which map whole sentences to real-valued vectors. In other words, they are similar to word embeddings but instead of operating on words they operate on sentences.

In the lecture, you learned about naïve approaches to generating simple sentence embeddings from word embeddings which we will explore in this task. We will evaluate these embeddings on the Semantic Textual Similarity (STS) problem. For STS, sentence embeddings are used to decide whether two sentences are similar in meaning or not. Usually, STS datasets consist of sentence pairs and human annotated scores which range from 0 (completely dissimilar) to 5 (completely equivalent). We will use Spearman's ρ to measure the correlation between these scores and the cosine similarities of sentence embeddings derived from BERT.

(a) BERT prepends a `[CLS]` token to each sentence for which it generates contextual word embeddings. `[CLS]` stands for *classification* and it's there to represent the whole encoded sequence for sentence-level classification. For each sentence extract the embedding corresponding to `[CLS]` and use it for sentence representation. Complete the code in `task1_simple_embed.py` marked with `YOUR CODE HERE (a)`.

(b) Another simple technique you know from the lecture is to simply compute the arithmetic mean of the word embeddings for each sentence to get sentence embeddings. Complete the code in `YOUR CODE HERE (b)` and return mean-pooled word embeddings as sentence representations. How do the results compare?

Hint: To be able to process sentences in batches the tokenizer has to pad all sentences to the same length (using the `[PAD]` token). When you average word embeddings you should exclude embeddings which correspond to the `[PAD]` token. For that you can use the attention mask which the tokenizer also returns. The attention mask is a matrix with the same shape as the input tokens which contains zeros for indices where the token is `[PAD]` and ones for all other tokens.

Task 2 (15min): Power Means & SBERT

The arithmetic mean is not the only way in which word embeddings can be reduced to a single vector. Power mean embeddings generalize the average to the so-called power mean:

$$\mathrm{M}_p(x_1, \ldots, x_n) = \frac{\left(\sum_i x_i^p\right)^{\frac{1}{p}}}{n} \tag{1}$$

For the final sentence representation different power means are concatenated together. Common values are $p = -\infty$ (min-pooling), $p = 1$ (arithmetic mean), and $p = \infty$ (max-pooling).

(a) Complete the code marked with `YOUR CODE HERE (a)` in `task2_power_sbert.py` and either return mean-pooled (equivalent to the solution of the previous task), max-pooled, or min-pooled embeddings, depending on the `mode` parameter.

Another idea is to add a pooling layer at the output of BERT (e.g. mean-pooling) and then fine-tune the in this way modified BERT model on various downstream tasks (e.g. Natural Language Inference and Semantic Textual Similarity). This is the approach employed by sBERT, which achieves state-of-the-art performance in many cases.

(b) Complete the code in `YOUR CODE HERE (b)` and return sBERT embeddings. Which method works better for you?

Hint Look at `https://github.com/UKPLab/sentence-transformers/tree/master/examples/applications/computing-embeddings` for usage examples of sBERT.

Hint The sBERT library uses PyTorch internally, another popular library for deep learning which is not covered by this tutorial. If you get a PyTorch tensor somewhere you can use `.numpy()` to convert it to a numpy array.

Task 3 (20min): Clustering

Clustering is a very prominent technique for exploratory data analysis. It concerns itself with grouping a set of objects into clusters so that objects in the same cluster are very similar while data points across clusters are very different to one another. A very common clustering algorithm is called k-means. Given a predefined number of clusters, k-means first randomly assigns each data point to a cluster and then performs the following two steps iteratively, until convergence:

Assignment step Assign each data point to the cluster with the nearest mean.

Update step Recalculate the means of each cluster.

In this exercise we will use k-means clustering with sentence embeddings to find out which sentences are categorically related. Complete the code in `task3_clustering.py`.

(a) Implement a function which returns all data points that belong to a cluster in a list. Complete the code in `YOUR CODE HERE (a)`.

(b) Implement a function that calculates for each cluster the mean of all its data points and returns the results in a list. Complete the code in `YOUR CODE HERE (b)`.

(c) Reassign each data point to the cluster with the closest mean in the section marked with `YOUR CODE HERE (c)`.

Hint You can use `numpy.linalg.norm` to compute the distance between two vectors.

Task 4 (10min): Multilingual Sentence Embeddings

There are multiple approaches to inducing multilingual sentence embeddings. A popular one is called multilingual knowledge distillation. The underlying principle of multilingual knowledge distillation is to train a model on cross-lingual but English-centric parallel data, with the goal of generating the same embedding as (English) sBERT for the same sentence in different languages. In this task you will investigate how well this approach works for assessing sentence similarity. Complete the code in `task4_multilingual.py`. The function marked with `YOUR CODE HERE` should return a matrix with pairwise cosine similarties between all sentences.