Lexical ressources: Multilingual Resources, Exercise 1

Multilingual Resources - Exercise 1

In this exercise you will handle embeddings for multiple languages.

You will need the python libraries NLTK wordnet (as well as the Open Multilingual Wordnet), gensim and numpy. Other libraries you might find useful include Matlotlib for visualisation and scipy or sklearn for data manipulation. You should have already installed most of these libraries in previous sessions. If that's not the case, note that nltk and gensim can be retrieved from pip; to install wordnet you will have to use the NLTK download functionality. Once NLTK is installed, simply run `python3 -c 'import nltk; nltk.download("wordnet"); nltk.download("omw")'` to install wordnet and OMW. The Open Multilingual Wordnet (or OMW for short) NLTK corpus contains the variants for wordnet in other languages than English. You can check it out in more details at this page.

This exercise also requires that you have access to two sets of embeddings for two different languages. You can retrieve such embeddings from the NLPL word embedding repository. **Pick embeddings with the same dimensionality, preferably computed with word2vec**, though other non-contextualized vectors, such as fasttext or Glove vectors, should work fine. As your work is going to to involvle fiddling with matrices, you might want to start with a fake, simplified dataset, eg. keeping only the first 10000 vectors.

You won't be provided with a canvas this time either. This exercise requires more thinking and understanding what you're doing, and less actual coding and hacking around with the lexical resources and models. Ask questions if anything is unclear!

1.1 Write a function to compute a set of word translations

There are multiple ways to do this.

- The most simple one, which has been shown to roughly work, is to simply take literal matches in the vocabularies of two sets of embeddings as valid word translations. The idea is that exact matches should correspond to loanwords or toponyms, which should have similar meanings across languages. For example, the word *Paris* should mean roughly the same thing in French or in English; likewise for *tacos*, *falafel*, or, to some extent, *parking*. Obviously, this only works if the two languages are written in the same alphabet.
- A more principled way of computing a set of word translations is to use an existing multilingual resource. The multilingual functionalities of wordnet can prove useful here. Do note, however, that words can be polysemous and ambiguous, and thus have multiple translations. Consider for instance that the French word *bois* can be matched to the English words *horn*, *wood*, *forest*, or *drink* (among other). Hence, you might want to restrict yourself to monosemous lemmas... or not, as that might severely limit the number of word translations you have.

The output of this function should be a list of pairs of words.

1.2 Write a function to load vector subspaces

In this second step, the aim is to restrict our attention only to the vectors for which we have a word-translation. Hence, you will have to write a function that takes an existing set of embeddings and a sequence of words, and returns a set of embeddings - as a matrix - corresponding to that list of words.

To do so, you will need to first load the set of embeddings (eg. with `gensim.models.KeyedVectors.load_word2vec_format()`), and then select the embeddings that you care about (which you can do in gensim using the bracketted syntax: `vector = model[word]`), before concatenating them all into a single matrix (see the function `numpy.concatenate()`).

1.3 Write a function to estimate the necessary rotations using SVD

While this is the more intellectually challenging part of this exercise, it doesn't require much computation.

1. Compute the matrix product of the two subset matrices $P = W_1^D \cdot W_2^D$
2. Apply the SVD algorithm on this matrix P (cf. the function `numpy.linalg.svd()`), which should return you the U, Σ and V^T matrices.
3. Discard the Σ matrix, and retrieve the V matrix with a transposition: $V = (V^T)^T$.
4. The estimated necessary rotations to return are U and V.

1.4 Put everything together

Write a function that ties everything together, from computing the set of word translations to estimating the rotations and applying them to the full set of embeddings via a simple matrix multiplication. Make sure that the subsets of embeddings you use to estimate the rotations are correctly described as matrices with matching rows.

A good practice to follow is to print out an evaluation of how good this alignment is. To do so, you can split randomly your set of word translations, using 90% of the word pairs to estimate the necessary rotations. The remaining 10% can be used to asses the model: for a given translation pair $\langle w_1^i, w_2^i \rangle$, the aligned vectors $\langle \vec{w}_1^i, \vec{w}_2^i \rangle$ should be exact matches. Hence how far from an exact match you are for that pair gives you a measure of how good the alignment is: this can be assessed for instance with cosine distance:

$$cosdist(\vec{w}_1^i, \vec{w}_2^i) = 1 - \cos(\vec{w}_1^i, \vec{w}_2^i)$$

Another way is to verify that the nearest neighbor of $\vec{w}_1^i$ is indeed $\vec{w}_2^i$, which can be interpreted as a **precision score** if you average it over the entire 10% test split.

1.5 Test and play

Once you've set up a basic alignment pipeline, the next thing is to experiment with it. Here are possible things to test and check:

- Write a function that, when given a word w_i from a language L retrieves its k nearest neighbors $n_1, n_2, \ldots, n_k$ in the other language L'. Display them on a graph, eg. using Matplotlib.
- Do you get similar results across languages? Are some language pairs easier to align than others, ie. yield better precision scores? Does the precision score correlate with the number of word translations pairs you have?
- When computing translation pairs, does using literal matches bring worse results than using wordnet? Does using monosemous lemmas from wordnet impact your results? What about using both Wordnet *and* literal matches?