Computational Semantics and Pragmatics 2013

Raquel Fernández – ILLC, University of Amsterdam

Guide to `dsm_basic.py` for those with very little programming experience.

- Preliminaries: to run the program, you will need NLTK. Install NLTK http://www.nltk.org/install.html and NLTK Data http://www.nltk.org/data.html. This should be problem-free.
- Once NLTK is installed and accessible, you should be able to run `dsm_basic.py`. It will take a couple of minutes to finish.
- Look at the code while reading the remaining points in this guide.
- In this basic version of the program, the corpus used to create the DSM is the Brown corpus. All the words in the corpus are read in and stored in the list `brown_words`.
- The vector space is created with the NLTK data type `ConditionalFreqList` (see NLTK's chapter 2, section 2.2 for details http://nltk.org/book/ch02.html). This will create a vector space where the "conditions" of the conditional frequency distribution (the rows in the matrix) correspond to the target words and the "samples" (the columns) to the context words.
- Initially, the vector space is empty (`space = nltk.ConditionalFreqDist()`). To populate it with frequency counts, the script takes each of the words in the list `brown_words` and goes through the following steps
  - `current` refers to the current target word being considered; `index` refers to the position in the list `brown_words` of the current target word;
  - To look for context words, the script makes use of the `context_size` parameter, which can be varied but is initially set to 10 (`context_size = 10` towards the beginning of the file). It takes as context words each of the words found within a range of 10 positions to the left and to the right of the current target word. Note that instead of simply looking within `range(index - context_size)` and `range(index + context_size)`, the code is a bit more complex because it takes care of cases when there are fewer words than those specified by the context size because the target word is at, or to close to, the beginning or the end of the corpus.
  - `cxword` refers to each context word found in the `context_size` range; `cxword_index` refers to the position of that context word in the list `brown_words`.
  - Each time a context word is found within the context size of the current word, the frequency count for the pair `(current,cxword)` is incremented (`space[ current ].inc(cxword)`).
- Going through the loop which creates a vector for each word in the corpus as specified above will take a few minutes (you will see the message `computing space...` while this is going on). Once the distributional model has been constructed, the script prints a few examples. It takes two example target words ("election" and "water") and for each of them prints the 50 context words with the highest co-occurrence frequency counts. This allows you to see what kind of context words are being considered and to guess what kind of vectors are associated with these target words. (As you will see, there are plenty of things that are less than ideal here...).
- Printing the overall matrix (`space`) would not be practical because it is too big, but you can print parts of the matrix by selecting specific target words (conditions) and specific dimensions or context words (samples). The basic script prints an example sub-matrix that shows the values of the dimensions 'vote' and 'water' for the example target words "election" and "water".
- Finally, the script defines a `cosine` function to measure the semantic similarity of two vectors and prints a few examples. As you can see, `cosine` takes three arguments (a vector space and two words) and returns a similarity score between 0 and 1.