Computational Semantics and Pragmatics 2012

Raquel Fernández – ILLC, University of Amsterdam

Homework #4 Due: 21/12/2012, 5:00PM

Exercise 1. The main aim of this exercise is to make you reflect about aspects of meaning that only become apparent when we examine the distributional behaviour of some words.

Have a look at this online tool: http://www.scottishcorpus.ac.uk/corpus/bnc/compair.php

(a) What do you think this tool does? You are welcome to search for descriptions of the tool on the web, but explain it in your own words.

(b) The English adverbs *'utterly'* and *'absolutely'* are typically considered synonymous by most thesauri (see e.g. http://thesaurus.com/browse/utterly). Compare them using this tool. What do you observe? What can you conclude about their meaning and usage?

(c) Compare at least one more pair of words and comment on the results.

Exercise 2. This exercise asks you to implement a simple Distributional Semantic Model, to incrementally improve it, and to evaluate its performance informally with respect to how well it captures semantic similarity.

As a starting point, you can use the basic Python implementation of a DSM in the companion file `dsm_basic.py`, which uses some of the capabilities of the Natural Language Toolkit (http://nltk.org). This basic implementation is due to Katrin Erk (see the heading in `dsm_basic.py`). On the last page of this exercise sheet you will find a guide to the code for those of you with little programming experience.

Note that you are not required to use the implementation in `dsm_basic.py`. You are welcome to instead create your own implementation from scratch in you favourite programming language (always include clear documentation and sample runs or demos). You may also use an off-the-shelf package for vector space models (you will find a list of resources at http://www.wordspace.collocations.de/doku.php/software:start), Most of these packages offer many sophisticated capabilities, however bear in mind that it may be rather time consuming to get them set up and running and to understand all their parameters. They would be more appropriate for a project than for a homework exercise.

The following items assume that you take as a starting point the implementation in `dsm_basic.py`. If you decide to start from scratch, try to go through similar steps (as much as it seems reasonable). Most of the question items below are rather open-ended. They give you pointers on things you may want to try, but you are welcome to make your own decisions depending on your programming skills and your own interests.

You may answer each of the points below in turn or choose to address them in a general report.

(a) Start by briefly describing the characteristics of the basic DSM created with `dsm_basic.py` in terms of the DSM parameters we have seen in class (you can find a summary of parameters on slide 3 of the short recap lecture on 5 December).

(b) As you will quickly realise, the resulting model has several problems. Take, for instance, the similarity scores printed at the end for a few pairs of words. Why are they all so high?

(c) Try to improve the model by pre-processing the corpus further before building the vector space. Some possibilities: The NLTK includes a corpus of "stopwords", a list of high frequency words that have little lexical content (see chapter 2, section 2.4 of the NLTK book). You could use this list to filter

out stopwords from the corpus before constructing the vector space. The NLTK also includes several lemmatizers (see chapter 3, section 3.6) that you could apply to the corpus. Explain how your changes affected the model and give some examples.

(d) Experiment with some other parameters such as the context size or with different corpora. You may also consider more sophisticated steps such as implementing a function to weight the frequency counts, or anything else you think could improve the model. Again, explain how your changes affected the model.

(e) Find a way to evaluate your model in terms of how well it predicts semantic similarity. There is no need to do a thorough evaluation, simply consider a few representative examples. Some possibilities: compare the performance of your model with that of Web Infomap http://clic.cimec.unitn.it/infomap-query/ (if you didn't implement a way to extract the k nearest neighbours of a target word, you could simply compare the similarity scores given by the two models). Another possibility could be to compare the performance of your model to WordNet, for instance by checking whether pairs of words that are part of the same synset are assigned higher similarity scores by your model than words belonging to different synsets; or by checking whether there seems to be an inverse correlation between the similarity score assigned to pairs of words by your model and the path-distance between those words in the WordNet ontology. Feel free to come up with your own evaluation method. You may even decide to collect similarity judgements from a small group of subjects!

Guide for `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 is 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.googlecode.com/svn/trunk/doc/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.