Assignment – Distributional Semantics

89-680 Introductions to NLP

The goal of this assignment is to compute distributional similarities for words based on corpus data and to assess and evaluate those similarities. The first parts of the description below explain how to compute word similarity using vectors in the original feature space, which you will build using corpus statistics. Then, you are also asked to compute similarities in an analogous manner using pre-trained word2vec word embeddings.

Corpus

Collect your statistics from a sample of Wikipedia, which is parsed by a dependency parser.

To first understand the corpus format, you can view in a text editor a smaller sample:
http://u.cs.biu.ac.il/~89-680/wikipedia.tinysample.trees.lemmatized
See explanation of the corpus format at: https://depparse.uvt.nl/DataFormat.html
If you like to view the parse tree of a sentence in a dependency tree format use this viewer:
http://www.cs.biu.ac.il/~89-680/DepTreeViewer_17_06_10.jar

Statistics should be collected at the level of the word lemma, which is included for each word in the corpus. Note the comments below on required filtering of uncommon words and contexts.

FYI, the lemmas were produced by python module nltk.stem.wordnet.WordNetLemmatizer (for info about the lemmatizer see: http://nltk.org/api/nltk.stem.html).

Similarity Computation

As explained in the next section, you are given certain target words and need to compute the similarity values between each target word and all other words in the corpus (which pass certain threshold criteria, as explained below).

General guidelines:

- Vector similarity measure: cosine
- Word-feature association measure: PMI, following the estimation method taught in class.
    - Optional: try the smoothed version of PMI, as taught in class, which was borrowed from word2vec. If you try both versions mention in the report which worked better.
- Implement the efficient algorithm for computing simultaneously the similarity between a target word and all other words, as we learned.

Word-feature co-occurrence types

You should experiment with three types of distributional vectors for representing a target word. These vector types are based on the following three types of co-occurrence between the target word and its features:

1. Content words co-occurring with the target word in the same sentence. The feature is the co-occurring word.
2. Content words within a window of two content words on each side of the target word (skipping function words). The feature is the co-occurring word.
3. Content words which are connected to the target word by a dependency edge. In addition to the connected word, the feature will include the label and direction of the dependency between the target word and the feature. In addition:
    a. If there is a dependency edge that connects a preposition to the target word then generate a feature in the following way:
        i. If the preposition modifies the target word (the target word is the parent) then create a feature with the head noun that modifies (is daughter of) the preposition.
        ii. If the preposition is modified by the target word (the target word is the daughter) then create a feature with the word that is modified by (is parent of) the preposition.
        iii. In both cases, the edge label should include the label of the edge which connects the preposition to its parent, concatenated with the lemma of the preposition itself.
        iv. The idea of this specification is to have the feature word being the content word that is related to the target word via the preposition ("skipping" the preposition). In addition, the preposition itself is collapsed into the dependency label, which makes sense since different prepositions usually reflect different semantic relationships. This is a common practice when using dependency relations for semantic purposes.

The required similarity computations should be performed three times, once for each co-occurrence type.

Thresholds and Filters

In the required evaluations you need to compute similarities only between words that occur at least 100 times in the corpus (in the lemma form). Thus, you need to build distributional vectors only for words of this or higher frequency.

In addition, to fit the computation into your computer resources you **have to apply** the following filters on features

- ignore words that have less than 75 occurrences (**this is mandatory**).
- limit to 100 most common contexts per word (**this is mandatory**).

Applying the filters will reduce the size of the word-feature co-occurrence matrix. Submit a file "counts_words.txt" with the counts for the 50 most common words, each in a separate line, in the following format:

w1 *<space>count
w2 <space> count

….

In addition, submit a file “counts_contexts_dep,txt”, the with counts of the 50 top dependency contexts in the dataset. The format is similar:

feature1 <space> count
feature2 <space> count
….

Required evaluation

Compute and print the top 20 most similar words, from the whole corpus, for each of the following **target words**:

car bus hospital hotel gun bomb horse fox table bowl guitar piano

In the submission, attach a file that presents for each of these target words three lists of the top 20 most similar words, one list for each of the three co-occurrence types. For each target word print the three lists in three columns (like in a table), so that it will be easy to view and compare them side by side. The file should be named top20.txt and its format is:

car

most_similar_word_window <space> most_similar_sentence_window <space> most_similar_dep

2nd similar_word_window <space> 2nd_similar_sentence_window <space> 2nd_similar_dep

****

bus

….

****

….

Examine the lists for these 12 target words and try to come up with some comparative characteristics of the types of similarity obtained for each co-occurrence type. Report your qualitative conclusions briefly in the evaluation report, along with examples.

In addition, take the first and last target words above (car, piano) and annotate their similarities manually. That is, for each candidate word in the similarity lists, make two judgments:

1. Whether you judge the candidate word to be topically related to the target word.
2. Whether you judge the candidate word to be in the same semantic class as the target word.

Notice that while for each word you have 3 similarity lists of 20 candidates, many words will appear in more than one list, so you need to perform these judgments for less than 60 candidates per target word.

Include your manual judgments in the submitted report and compute the average mean precision obtained for the similarities produced by each type of co-occurrence.

Word2vec evaluation

Upload the pre-trained word2vec vectors from the following website:

https://levyomer.wordpress.com/2014/04/25/dependency-based-word-embeddings/

Use two of the different versions of the embeddings: dependency-based and bag of words with window of k=5.

Repeat the evaluation above also for the two types of word2vec vectors. To compute similarities, consult the Python implementation slide from class, which relies on Numpy package, for which you can find a useful documentation here:

http://cs231n.github.io/python-numpy-tutorial/

What to submit

Code

Follow the same instructions regarding the code and its submission as in the previous assignments.

Report

Write a short report containing the following:

1. Summarize statistics regarding your experiments:
    a. The various thresholds you employed.
    b. Number of words considered for similarity: the number of words which passed the 100 frequency threshold (columns in the word-feature co-occurrence matrix).
    c. The number of features considered in your computation for each co-occurrence type (rows in the matrix).
2. Print the 20 most similar words (2^{nd}-order similarity) for each of the target words in the qualitative evaluation, for each co-occurrence type (in the table format instructed above). These lists should appear as an appendix at the end of the report. Write in the report your conclusions from examining these lists, as instructed above, along with examples.
3. For each target word, print in a convenient table format the 20 top context attributes for that word, for each of the 3 co-occurrence types (those attributes with highest PMI values in the target word's vector, considered as 1^{st}-order similarities). Write a short qualitative comparison between the lists in item 2 (2^{nd}-order) and the lists in item 3 (1^{st}-order).
4. Report your manual judgments for the two words (possibly in appendix), the MAP results for the 3 co-occurrence types (AP for each of the two words and their average as MAP), and provide some insight on these results.
   Notice: MAP should be calculated separately for each manually judged similarity type (topical and semantic).
5. Write a brief description of your implementation of:

a. The estimation of the PMI values
b. The efficient algorithm for computing all similarities for a target word.

In the description, point at the relevant parts in your code.

6. Report the results of the word2vec experiment, with the same types of content as in items 2, 3 and 4 above, for the two uploaded vector types of word2vec. For item 3, report the 10 context words for which the context vector has highest dot product similarity with the target word (see slides 73-77 in the word2vec presentation). Write a short comparison between the results of items 2-4 above and the corresponding results for the word2vec experiment.

Advice for the implementation of corpus statistics collection

In your implementation, you are advised to use the following data structure:
- a (main) hash-table (python dict or java HashMap) mapping words to (secondary) hash-tables.
- each of the secondary hash table will map a context to a number (which is the count or the score).

When going over the text, whenever you encounter a word-context pair you access the main hash-table and retrieve the hash-table for that word. You then update the secondary hash-table by increasing the context count.

In python, it will look something like this:

```
"""
from collections import defaultdict, Counter

counts = defaultdict(Counter)

for word, context in ....:
  context_counts_for_word = counts[word]
  context_counts_for_word[context] += 1
"""
```

For better memory efficiency, you may want to:
- store each word and context as a number and not a string (with additional hash-tables mapping strings to numbers).

Educational note: in a more realistic scenario, you may be required to work with much larger input files. In this case, you could not hold the entire counts for all the words in memory. The main problem will be the 1-count pairs -- if you could filter them (and maybe also the 2-count and 3-count pairs) you will likely be fine. But how do you know in advance that some word-context pair will have a low count? It is then advisable to use temporary disk storage. Your initial program will produce a file where each line is a single word-context pair, separated by space. Then, use a program such as unix's "sort" utility, that can sort the

lines in very large files by making use of temporary disk storage. Then, you could go over the large file in order, and read each word individually, throw away the one-counts for that word, and proceed to the next one.