LIN 350: Computational Semantics, Fall 2018, Erk

Homework 2: Distributional modeling

Due: Thursday October 11, 2018

This homework comes with a file, `lastname_firstname_hw2.py`. This is a (basically empty) file. Please record all the answers in the appropriate places of this file. Please make sure that you save the file in *plain text*, not something like the Word format.

For submission, please rename the file such that it reflects your last name. Please also remember to put your name in the 2nd line of the file, so that I know whose homework I'm looking at when I print out your answer.

For all problems, I need to see your code. I may also want to see answers to particular questions if the problem says so, but above all I need to see all the Python code that you used to solve a particular problem.

In this homework, you have a choice of **two completely separate options, option A and option B**. **Please choose one of the two options, and only complete the problems for that option**. Do not complete the problems in both option A and option B. Please mark the option that you are choosing in your solution file.

Whichever of the options you choose, please use Python to construct your solution.

If any of these instructions do not make sense to you, please get in touch with the instructor right away.

A perfect solution to this homework will be worth *100* points.

Option A: Using Gensim to build distributional models

For this option, you will need the following Python packages:

- NLTK, with NLTK data
- gensim, https://radimrehurek.com/gensim/
- pandas, https://pandas.pydata.org/

Background: gensim

The Python `gensim` package has functions for building and using different kinds of distributional models. Among other things, it has functions to build `word2vec` distributional models. There is a worksheet that demonstrates this at http://www.katrinerk.com/courses/python-worksheets/python-demo-gensim for an example. Basically, you can train a `word2vec` model (that is, compute vectors for all target words appearing in the text) from an NLTK corpus, like this:

```
from gensim.models import Word2Vec
from nltk.corpus import brown

brownmodel = Word2Vec(brown.sents(), iter=10, min_count=10, size=300)
```

where `brown.sents()` is an NLTK function that gives `gensim` access to the sentences in the Brown corpus. `iter=10` tells `gensim` how often it should go over the data during learning; the higher the number of iterations, the longer the training takes, and the more in-depth the model learns from the data. `min_count=10` says that words appearing less than 10 times should be discarded; if you have little data, don't set this number too high. `size=300` sets the number of dimensions in the distributional vectors. If you train on a small corpus, consider choosing a lower number of dimensions.

Here is how to use the trained model:

```
# compute pairwise similarity
brownmodel.wv.similarity("cat", "dog")

# get nearest neighbors, in this case 10
brownmodel.wv.most_similar("dog", topn = 10)
```

Background: pandas

Pandas is a Python package that is very convenient for managing and manipulating tables of data (as in Excel sheets). You can read in tables from files,

access columns by name and add columns, select rows by column values, sort by a column, and do a number of column-wise operations. Here are some examples:

```
# read in data from a file.
# 'csv' means comma-separated values.
# other column separators, such as whitespace, also works
filename = "/Users/kee252/Data/VS_Datasets/WordSim353/combined.tab"
wordsim353 = pandas.read_csv(filename, sep="\s+")

# how many rows are there?
print("number_of_rows:", len(wordsim353))

# accessing the column named "Human_mean"
print(wordsim353["Human_mean"])
print(wordsim353.Human_mean)

# accessing the first few rows of the table
print(wordsim353.head())

# adding a column.
# this one holds the index of each row.
# not terribly useful, just for demo purposes.
wordsim353["index"] = list(range(len(wordsim353)))

# mean, median, first quartile, third quartile
# over a whole column
print("mean", wordsim353.Human_mean.mean())
print("median", wordsim353.Human_mean.median())
print("1st_quartile", wordsim353.Human_mean.quantile(0.25))
print("3rd_quartile", wordsim353.Human_mean.quantile(0.75))

# sorting
# this produces a *new* sorted dataframe, which
# we capture in a new variable ws_sorted.
# the original wordsim353 remains unchanged.
# ascending = False means we want the largest values to come first.
ws_sorted = wordsim353.sort_values(by = "Human_mean", ascending = False)

# selecting rows based on column values.
# here: select pairs with high human ratings.
# this returns a new dataframe that only contains
# the rows in which Human_mean is above 7.
ws_high = wordsim353[wordsim353.Human_mean > 7]

# select pairs where either Word1 or Word2 is "tiger"
# don't omit the (), or you get very interesting error messages.
```

```
# again, this produces a new dataframe
wt = wordsim353[(wordsim353.Word1 == "tiger") | (wordsim353.Word2 == "tiger")]
```

Helpful code

You are welcome to use the following demo code as a basis for your solution:

http://www.katrinerk.com/courses/python-worksheets/python-demo-gensim

http://www.katrinerk.com/courses/python-worksheets/python-demo-predicting-word-similarity

Option A Problem 1: Finding nearest neighbors (25 pts.)

NLTK comes with a number of corpora. Among them are the following corpora, each comprising 1-2 million word tokens:

- `brown`: mixed genres, balanced corpus
- `gutenberg`: novels out of copyright
- `treebank`: Wall Street Journal
- `reuters`: short news reports

For this problem, you will compare distributional models computed from different sources and with different parameters. Choose **two** of the four corpora above. For each of the corpora, train **two** distributional model, for a total of 4 models. You can vary:

- number of dimensions (parameter `size`)
- number of times the corpus is read (parameter `iter`)
- cutoff for word frequency (parameter `min_count`)
- context window size (parameter `window`)
- algorithm to be used, skipgram or cbow (parameter `sg`, can be set to 0 or 1)
- ...

For a complete list of parameters, run

```
from gensim import Word2Vec
help(Word2Vec)
```

Now think of **three polysemous words** (that is, words with more than one sense). Ideally, choose words that you suspect would be used differently in the two corpora, for example you might choose the noun *bat* if you suspect that

one of your corpora talks about bat-the-animal more and the other corpus talks about bats-in-baseball more.

For each of those polysemous words, and each of the spaces (a.k.a. distributional models) that you have computed above, compute the **ten nearest neighbors** of that word in that space. Inspect the results, and comment on what you find. Do you observe differences in the results based on the corpora from which the spaces were computed? Do you observe differences based on the other parameters that you varied?

Option A Problem 2: Finding the odd one out (20 pts.)

"Odd one out" are sets of four words, one of which is the "odd one out": It differs form the other three words. For example, in "teacher actor doctor two", the word "two" is the odd one out. Here is how you can use distributional similarity to predict the "odd one out" for this example:

- Compute similarity of "teacher" to the other three words, and average over the three similarity values.
- In the same way, compute the average similarity of "actor" to the other three, of "doctor" to the other three, and of "two" to the other three.
- The one that has the lowest average similarity to the other three is the predicted "odd one out".

Here is a Python function that you can use to compute the average of a list of numbers:

```
def average(numberlist):
    return (sum(numberlist) / max(len(numberlist), 1))
```

Choose **one of the four distributional models you computed for problem 1**, and use it to detect the odd one out among the following four words:

tree mouse man paper

Option A Problem 3: Loading and probing the BLESS dataset (10 pts.)

The BLESS dataset is available on Canvas under Assignment 2. BLESS is a set of word pairs in particular semantic relations, as we discussed in class. You can use the slideset to remind yourself, or look at the original paper at https://www.aclweb.org/anthology/W11-2501.

Please download the dataset, and load it into Python using the `pandas` function `read_csv()`. The columns in this dataset are separated by whitespace, so please

use the same `sep` parameter setting as in the example above. Please store the BLESS data in a pandas dataframe called `bless`.

How many rows are there in the BLESS dataset? How many rows are there in the BLESS dataset where the "Relation" entry is "attri"?

Option A Problem 4: Computing similarities for BLESS (20 pts.)

Choose **two** of the spaces you computed for Problem 1 above, let's call them "space1" and "space2". For each of these two spaces, add a column to the `bless` dataframe from Problem 3 that contains cosine similarities of "Word1" and "Word2", as in the wordsim353 demo. You should add a column `space1` and a column `space2`. The column for `space1` should contain the cosine similarities, according to space1, for Word1 and Word2 in the first row, Word1 and Word2 in the second row, and so on. Likewise for space2.

Option A Problem 5: Summarizing results on BLESS (25 pts.)

You are now in a position to compare space1 and space2. We will do a simplified evaluation compared to the original BLESS (this is just in case you read the paper and are wondering why we are not doing what they say to do).

For space1, do the following:

- Select the sub-dataframe of `bless` where the Relation entry is "attri". Compute the **median**, **first quartile**, and **third quartile** of the similarities in column "space1" in that dataframe. The sample code at the beginning of the homework shows you how to do this.
- Do the same for the Relation entries "coord", "event", "hyper", "mero". (We'll skip the "random" entries.)

Now do the same for space2.

Comment on what you see. Are there differences between the two spaces that you can see from this analysis?

Option B: Constructing a distributional model from scratch

In this assignment, you will compute a distributional model from a corpus that is not as small as in the demo, but should be still manageable.

The corpus

The corpus is a collection of Gothic novels from Project Gutenberg, slightly expanded from the version we used in class last time. It comprises 2,743,346 words, including Project Gutenberg small print. It is available on Canvas under Assignment 2.

Code

For your solution to Option B, please use as a basis the code "building blocks of a distributional model" at `https://utexas.box.com/s/1dwtxh01droqjyrjzv334sudekk363v9` and modify it as needed.

Option B Problem 1: Preprocessing (25 pts.)

From each file in the corpus, remove the Project Gutenberg fine print in front of the actual content and after the actual content.

Then use Stanford CoreNLP to preprocess your data: Please do part-of-speech tagging and lemmatization.

Represent your data in the form `lemma-POS`, for example `go-VB` or `fire-NN`. Watch out: Please only use the first two letters of the part-of-speech tag. The rest is specific to the wordform, like `VBD` if the wordform is "went" rather than "go". But as you are using lemmas anyway, the specific part of the part-of-speech tag is useless. If you consistently only use the first two letters, you avoid the problem.

Afterwards, remove stopwords.

Option B problem 2: Computing the space (50 pts.)

Frequency cutoff

Choose a reasonable frequency cutoff for the target words and the context words. It is your choice. You do not have to use the same cutoff for both targets and context words. There are different ways you can do a cutoff: You can use some

minimum frequency for the context words in the corpus, or you can use only the N most frequent context words overall.

In your solution, please indicate clearly what kind of frequency cutoff you are using.

Context window

Please use a one-sentence context window.

The implementation that identifies context words of a target (the function `co_occurrences()` is inefficient in that it first creates a large list of target/context pairs (which contains each pair twice, as ⟨ word1, word2⟩ and ⟨ word2, word2⟩) and then does the counting.

Interleave context identification and counting, and do not create pairs twice. Provide comments on your code.

Storing your space

The demo script stores a space as a dictionary mapping from target words to a separate numpy array for each target. Change the representation such that it stores the whole space in one matrix, with one row per target word and one column per context dimension. Please use numpy arrays for this, not sparse matrices, as the latter are a hassle to use.

You will have to adapt the whole code, including PPMI computation and cosine similarity, to the use of a matrix.

Option B Problem 3: Using your model (25 pts.)

Determine the top 10 most similar words for each of the following target words, which all occur 50 times or more in the corpus.

> beauty-NN fire-NN heart-NN dare-VB throw-VB conceal-VB quiet-JJ evil-JJ apparently-RB slowly-RB

Determine these top 10 words for all 3 spaces, the raw counts space, the PPMI space, and the SVD-reduced space.

In your solution, include these lists of 10 most similar words. Also include a discussion of what you see:

- Do the lists of 10 most similar words look best for the raw space, the PPMI space, or the SVD space?
- Do you detect effects of the genre (gothic-ish stories) in these word lists?

- Inspecting only the *top* most similar word for each target, and only for the PPMI space, how often do you see synonyms of the target words? co-hyponyms? antonyms? You do not need to use a dictionary here, it will suffice if you label the words yourself. But you can use WordNet if you would like.