hw3

November 11, 2017

1 Homework 5: Lexicons and Distributional Semantics

This is due on **Friday, 11/10 (11pm)**

1.1 How to do this problem set

Most of these questions require writing Python code and computing results, and the rest of them have textual answers. Write all the textual answers in this document, show the output of your experiment in this document, and implement the functions in the `python` files.

Submit a PDF of thie .ipynb to Gradescope, and the .ipynb and all python files to Moodle.

The assignment has two parts: * In the first part, you will experiment with Turney's method to find word polarities in a twitter dataset, given some positive and negative seed words. * In the second part, you will experiment with distributional and vector semantics.

Your Name: Tapojit Debnath Tapu

List collaborators, and how you collaborated, here: (see our grading and policies page for details on our collaboration policy).

- *name 1*

1.2 Part 1: Lexicon semantics

Recall that PMI of a pair of words, is defined as:

$$PMI(x,y) = log\frac{P(x,y)}{P(x)P(y)}$$

The Turney mehod defines a word's polarity as:

$$Polarity(word) = PMI(word, positive\_word) - PMI(word, negative\_word)$$

where the joint probability $P(w,v)$ or, more specifically, $P(w\ NEAR\ v)$ is the probability of both being "near" each other. We'll work with tweets, so it means: if you choose a tweet at random, what's the chance it contains both `w` and `v`?

(If you look at the Turney method as explained in the SLP3 chapter, the "hits" function is a count of web pages that contain at least one instance of the two words occurring near each other.)

The positive_word and negative_word terms are initially constructed by hand. For example: we might start with single positive word ('excellent') and a single negative word ('bad'). We can also have list of positive words ('excellent', 'perfect', 'love', ...) and list of negative words ('bad', 'hate', 'filthy',....)

If we're using a seed list of multiple terms, just combine them into a single symbol, e.g. all the positive seed words get rewritten to POS_WORD (and similarly for NEG_WORD). This $P(w, POS\_WORD)$ effectively means the co-ocurrence of w with any of the terms in the list.

For this assignment, we will use twitter dataset which has 349378 tweets. These tweets are in the file named `tweets.txt`. These are the tweets of one day and filtered such that each tweet contains at least one of the seed words we've selected.

1.3 Question 1 (15 points)

The file `tweets.txt` contains around 349,378 tweets with one tweet per line. It is a random sample of public tweets, which we tokenized with twokenize.py's tokenizeRawTweetText()). The text you see has a space between each token so you can just use `.split()` if you want. We also filtered tweets to ones that included at least one term from one of these seed lists: * Positive seed list: ["good", "nice", "love", "excellent", "fortunate", "correct", "superior"] * Negative seed list: ["bad", "nasty", "poor", "hate", "unfortunate", "wrong", "inferior"]

Each tweet contains at least one positive or negative seed word. Take a look at the file (e.g. `less' and`grep'). Implement the Turney's method to calculate polarity scores of all words.

Some things to keep in mind: * Ignore the seed words (i.e. don't calculate the polarity of the seed words). * You may want to ignore the polarity of words beignning with `@` or `#`.

We recommend that you write code in a python file, but it's up to you.

QUESTION: You'll have to do something to handle zeros in the PMI equation. Please explain your and justify your decision about this.

textual answer here

If P(x,y) in PMI equation is zero, it is assumed that PMI is zero. For instance, for a given word, w, if P(w, POS_WORD) is zero, then PMI for it is zero. PMI for NEG_WORD is asserted to be negative. IF P(w, NEG_WORD) is zero, PMI for it is zero. PMI for POS_WORD is asserted to be positive. If P(w, POS_WORD) & P(w, NEG_WORD) are zero, then polarity is zero. Zero polarity means word is neutral.

1.4 Question 2 (5 points)

Print the top 50 most-positive words (i.e. inferred positive words) and the 50 most-negative words.

Many of the words won't make sense. Comment on at least two that do make sense, and at least two that don't. Why do you think these are happening with this dataset and method?

```
In [1]: # Write code to print words here
        import operator
        from collections import defaultdict
        from polarity import polarity_calc
        polarity=polarity_calc()
        max_50=dict(sorted(polarity.iteritems(), key=operator.itemgetter(1), reverse=True)[:50])
        min_50=dict(sorted(polarity.iteritems(), key=operator.itemgetter(1), reverse=True)[-50:])
        print "50 most positive tokens: ", "\n", max_50, "\n"
        print "50 most negative tokens: ", "\n", min_50

50 most positive tokens:
['friend\xf0\x9f\xa4\x9e\xf0\x9f\x8f\xbd', 'https://t.co/unteosecly', 'cassievers\xe2\x98\xba',
```

```
50 most negative tokens:
['us-they', 'sudmalis', 'bitchness', 'deomocrats', 'https://t.co/tx\xe2\x80\xa6', 'https://t.co/
```

1.4.1 Textual answer here.

Two that make sense: 'looooooovveeee' for positive tokens, since it is short for `love` and 'looser' from negative tokens, which means `loser`. Two that do not make sense: 'octavia' for positive tokens and 'condiment' from negative tokens. Both are supposed to be neutral words. They do not make sense since word in a tweet is considered to be positive or negative if there is a positive or negative seed word in said tweet respectively.

1.5 Question 3 (5 points)

Now filter out all the words which have total count < 500, and then print top 50 polarity words and bottom 50 polarity words.

Choose some of the words from both the sets of 50 words you got above which accoording to you make sense. Again please note, you will find many words which don't make sense. Do you think these results are better than the results you got in Question-1? Explain why.

```
In [2]: # Write code to print words here
        import operator
        from collections import defaultdict
        from polarity import polarity_calc
        polarity=polarity_calc(filter_500=True)
        max_50=dict(sorted(polarity.iteritems(), key=operator.itemgetter(1), reverse=True)[:50])
        min_50=dict(sorted(polarity.iteritems(), key=operator.itemgetter(1), reverse=True)[-50:]
        print "50 most positive tokens: ", "\n", max_50, "\n"
        print "50 most negative tokens: ", "\n", min_50
```

```
50 most positive tokens:
['friend\xf0\x9f\xa4\x9e\xf0\x9f\x8f\xbd', 'https://t.co/unteosecly', 'cassievers\xe2\x98\xba',

50 most negative tokens:
['us-they', 'sudmalis', 'bitchness', 'deomocrats', 'https://t.co/tx\xe2\x80\xa6', 'https://t.co/
```

1.5.1 Textual answer here.

Words which make sense:
'friend\xf0\x9f\xa4\x9e\xf0\x9f\x8f\xbd', 'luck\xf0\x9f\x92\x9e\xf0\x9f\x92\x9c' from positive tokens(Only their beginning parts like 'friend' and 'luck' make sense).<br.> words which do not make sense: **'wowowowowowowowow', 'winnn'** from negative tokens, since they are supposed to be in positive token group. Results have not improved at all; corresponding token sets from question 2 and question 3 are still strikingly similar. This may be because most tokens have frequency below 500. Owing to their low frequency, they are most likely to be associated with either positive or negative group, even if high frequency words are filtered out.

1.6 Question 4 (5 points)

Even after filtering out words with count < 500, many top-most and bottom-most polarity don't make sense. Identify what kind of words these are and what can be done to filter them out. You can read some tweets in the file to see what's happening.

1.6.1 Textual answer here.

These "tokens" mostly consist of emojis, like `friend\xf0\x9f\xa4\x9e\xf0\x9f\x8f\xbd`. The 'friend' part is distinguishable, but characters after it represent emojis. Moreover, there are a lot of html links too, like `https://t.co/jm\xe2\x80\xa6`, which should be removed. HTML links can be identified by checking whether the first 4 characters of a token are equivalent to the string 'http'. If fourth character from end of token string is a backslash \, then it means this particular token consist mostly of emojis and can be removed.

2 Part-2: Distributional Semantics

2.1 Cosine Similarity

Recall that, where i indexes over the context types, cosine similarity is defined as follows. x and y are both vectors of context counts (each for a different word), where x_i is the count of context i.

$$cossim(x, y) = \frac{\sum_i x_i y_i}{\sqrt{\sum_i x_i^2}\sqrt{\sum_i y_i^2}}$$

The nice thing about cosine similarity is that it is normalized: no matter what the input vectors are, the output is between 0 and 1. One way to think of this is that cosine similarity is just, um, the cosine function, which has this property (for non-negative x and y). Another way to think of it is, to work through the situations of maximum and minimum similarity between two context vectors, starting from the definition above.

Note: a good way to understand the cosine similarity function is that the numerator cares about whether the x and y vectors are correlated. If x and y tend to have high values for the same contexts, the numerator tends to be big. The denominator can be thought of as a normalization factor: if all the values of x are really large, for example, dividing by the square root of their sum-of-squares prevents the whole thing from getting arbitrarily large. In fact, dividing by both these things (aka their norms) means the whole thing can't go higher than 1.

In this problem we'll work with vectors of raw context counts. (As you know, this is not necessarily the best representation.)

2.2 Question 5 (5 points)

See the file `nytcounts.university_cat_dog`, which contains context count vectors for three words: "dog", "cat", and "university". These are immediate left and right contexts from a New York Times corpus. You can open the file in a text editor since it's quite small.

Write a function which takes context count dictionaries of two words and calculates cosine similarity between these two words. The function should return a number beween 0 and 1. Briefly comment on whether the relative simlarities make sense.

```
In [1]: import distsim; reload(distsim)

        word_to_ccdict = distsim.load_contexts("nytcounts.university_cat_dog")
        cat_dog_sim=distsim.cos_sim(word_to_ccdict,'cat','dog')
        cat_uni_sim=distsim.cos_sim(word_to_ccdict,'cat','university')
        dog_uni_sim=distsim.cos_sim(word_to_ccdict,'university','dog')

        # write code here to show output (i.e. cosine similarity between these words.)
        # We encourage you to write other functions in distsim.py itself.
        print "\ncosine similarity between cat and dog: ", cat_dog_sim, "\n"
        print "cosine similarity between cat and university: ", cat_uni_sim, "\n"
        print "cosine similarity between university and dog: ", dog_uni_sim

file nytcounts.university_cat_dog has contexts for 3 words

cosine similarity between cat and dog:  0.966891672715

cosine similarity between cat and university:  0.660442421144

cosine similarity between university and dog:  0.659230248969
```

Write your response here:

They do make sense, since cosine similarity between `cat` & `dog` is highest(both are animals) in comparison with that between `cat` & `university` and `dog` & `university`.

2.3 Question 6 (20 points)

Explore similarities in `nytcounts.4k`, which contains context counts for about 4000 words in a sample of New York Times articles. The news data was lowercased and URLs were removed. The context counts are for the 2000 most common words in twitter, as well as the most common 2000 words in the New York Times. (But all context counts are from New York Times.) The context counts only contain contexts that appeared for more than one word. The file has three tab-separate fields: the word, its count, and a JSON-encoded dictionary of its context counts. You'll see it's just counts of the immediate left/right neighbors.

Choose **six** words. For each, show the output of 20 nearest words (use cosine similarity as distance metric). Comment on whether the output makes sense. Comment on whether this approach to distributional similarity makes more or less sense for certain terms. Four of your words should be:

- a name (for example: person, organization, or location)
- a common noun
- an adjective
- a verb

You may also want to try exploring further words that are returned from a most-similar list from one of these. You can think of this as traversing the similarity graph among words.

Implementation note: On my laptop it takes several hundred MB of memory to load it into memory from the `load_contexts()` function. If you don't have enough memory available, your

computer will get very slow because the OS will start swapping. If you have to use a machine without that much memory available, you can instead implement in a streaming approach by using the `stream_contexts()` generator function to access the data; this lets you iterate through the data from disk, one vector at a time, without putting everything into memory. You can see its use in the loading function. (You could also alternatively use a key-value or other type of database, but that's too much work for this assignment.)

```
In [1]: import distsim; reload(distsim)
        import operator
        word_list=["stock", "limited", "inning", "yellow", "saved", "paris"]

        for word in word_list:
            dict_cos_sim=distsim.stream_cos_sim_calc(word)
            near_20=dict(sorted(dict_cos_sim.iteritems(), key=operator.itemgetter(1), reverse=Tr

            print "\n20 words nearest to ", word, ":\n ", near_20, "\n"

        ###Provide your answer below; perhaps in another cell so you don't have to reload the da
```

```
20 words nearest to  stock :
  ['state', 'bathroom', 'ball', 'mood', 'government', 'sun', 'gym', 'title', 'overall', 'primary


20 words nearest to  limited :
  ['desire', 'closer', 'required', 'wise', 'continuing', 'hard', 'crucial', 'loyal', 'related',


20 words nearest to  inning :
  ['worldwide', 'partners', 'lane', 'session', 'floor', 'grade', 'cemetery', 'term', 'garage', '


20 words nearest to  yellow :
  ['pink', 'hip', 'fashion', 'sweet', 'green', 'gorgeous', 'cheap', 'quiet', 'fat', 'dry', 'brig


20 words nearest to  saved :
  ['raised', 'made', 'kept', 'missed', 'lost', 'left', 'given', 'changed', 'joined', 'rejected',


20 words nearest to  paris :
  ['europe', 'afghanistan', '1995', '1994', '1997', '1996', '1999', '1998', 'washington', 'baghd
```

Textual response For words in the word list like *yellow*, *paris*, respective 20 nearest words obtained for each of them make sense. For instance, for *yellow*, many words closest to it are also colors. For *paris*, many words closest to it are names of places and cities. But for *stock*, *limited*,

saved and *inning*, results are more sparse. But still, cosine similarity algorithm picked up some words very close to their respective concepts. In case of *stock*, words like *budget, economy, brain, government* were closest to it. For terms with very few words in same category as themselves (like *stock*), the cosine similarity algorithm does not perform so well. But in case of adjectives and nouns which fall within categories like color, places, the algorithm performs very well.

2.4 Question 7 (10 points)

In the next several questions, you'll examine similarities in trained word embeddings, instead of raw context counts.

See the file `nyt_word2vec.university_cat_dog`, which contains word embedding vectors pre-trained by word2vec [1] for three words: "dog", "cat", and "university", from the same corpus. You can open the file in a text editor since it's quite small.

Write a function which takes word embedding vectors of two words and calculates cossine similarity between these 2 words. The function should return a number beween -1 and 1. Briefly comment on whether the relative simlarities make sense.

Implementation note: Notice that the inputs of this function are numpy arrays (v1 and v2). If you are not very familiar with the basic operation in numpy, you can find some examples in the basic operation section here: https://docs.scipy.org/doc/numpy-dev/user/quickstart.html

If you know how to use Matlab but haven't tried numpy before, the following link should be helpful: https://docs.scipy.org/doc/numpy-dev/user/numpy-for-matlab-users.html

[1] Mikolov, Tomas, et al. "Distributed representations of words and phrases and their compositionality." NIPS 2013.

```
In [1]: import distsim; reload(distsim)

        word_to_vec_dict = distsim.load_word2vec("nyt_word2vec.university_cat_dog")


        cat_dog_sim=distsim.cos_sim_word2vec(word_to_vec_dict,'cat','dog')
        cat_uni_sim=distsim.cos_sim_word2vec(word_to_vec_dict,'cat','university')
        dog_uni_sim=distsim.cos_sim_word2vec(word_to_vec_dict,'university','dog')

        print "\ncosine similarity between cat and dog: ", cat_dog_sim, "\n"
        print "cosine similarity between cat and university: ", cat_uni_sim, "\n"
        print "cosine similarity between university and dog: ", dog_uni_sim
        # write code here to show output (i.e. cosine similarity between these words.)
        # We encourage you to write other functions in distsim.py itself.
```

```
cosine similarity between cat and dog:  0.827517295965

cosine similarity between cat and university:  -0.205394745036

cosine similarity between university and dog:  -0.190753135501
```

Write your response here: They do make sense. Cosine similarity between cat and dog is high and positive, whereas that between cat and university & dog and university is negative.

2.5 Question 8 (20 points)

Repeat the process you did in the question 6, but now use dense vector from word2vec. Comment on whether the outputs makes sense. Compare the outputs of using nearest words on word2vec and the outputs on sparse context vector (so we suggest you to use the same words in question 6). Which method works better on the query words you choose. Please brief explain why one method works better than other in each case.

Not: we used the default parameters of word2vec in gensim to get word2vec word embeddings.

```
In [1]: import distsim
        import operator
        word_list=["stock", "limited", "inning", "yellow", "saved", "paris"]

        for word in word_list:
            dict_cos_sim=distsim.stream_cos_sim_calc_word2vec(word)
            near_20=dict(sorted(dict_cos_sim.iteritems(), key=operator.itemgetter(1), reverse=Tr

            print "\n20 words nearest to ", word, ":\n ", near_20, "\n"
        ###Provide your answer below
```

```
20 words nearest to  stock :
  ['shareholders', 'debt', 'investors', 'exchange', 'profit', 'price', 'dollar', 'sale', 'shares


20 words nearest to  limited :
  ['available', 'strong', 'necessary', 'increased', 'providing', 'certain', 'required', 'crucial


20 words nearest to  inning :
  ['scored', 'mets', 'ball', 'innings', 'n.b.a', 'sixth', 'leg', 'pitcher', 'ninth', 'eighth', '


20 words nearest to  yellow :
  ['pink', 'blue', 'tree', 'gray', 'leather', 'metal', 'shorts', 'jeans', 'tiny', 'lights', 'bri


20 words nearest to  saved :
  ['loved', 'raised', 'save', 'missed', 'lost', 'noticed', 'gotten', 'changed', 'liked', 'paid',


20 words nearest to  paris :
  ['el', 'del', 'australia', 'italy', 'la', 'madrid', 'hotel', 'royal', 'france', '1960', 'de',
```

For dense vectors, the outputs make way more sense, hence this method works the best. In case of *stock*, words closest to it are more relatable in this case compared to sparse vectors, such

as *shareholders*, *debt*, *investors*, etc. It is the same case for *inning*, since words closest to it are *scored, mets, ball*, etc. Unlike dense vectors, sparse vectors contain mostly zeros. Hence, many words which are actually contextually similar to the given word are left out when sparse vectors are used. For dense vectors, very few values are zero, hence contextually similar words do not get left out.

2.6 Question 9 (15 points)

An interesting thing to try with word embeddings is analogical reasoning tasks. In the following example, it's intended to solve the analogy question "king is to man as what is to woman?", or in SAT-style notation,

king : man :: ____ : woman

Some research has proposed to use additive operations on word embeddings to solve the analogy: take the vector $(v_{king} - v_{man} + v_{woman})$ and find the most-similar word to it. One way to explain this idea: if you take "king", get rid of its attributes/contexts it shares with "man", and add in the attributes/contexts of "woman", hopefully you'll get to a point in the space that has king-like attributes but the "man" ones replaced with "woman" ones.

Show the output for 20 closest words you get by trying to solve that analogy with this method. Did it work?

Please come up with another analogical reasoning task (another triple of words), and output the answer using the same method. Comment on whether the output makes sense. If the output makes sense, explain why we can capture such relation between words using an unsupervised algorithm. Where does the information come from? On the other hand, if the output does not make sense, propose an explanation why the algorithm fails on this case.

Note that the word2vec is trained in an unsupervised manner just with distributional statistics; it is interesting that it can apparently do any reasoning at all. For a critical view, see Linzen 2016.

```
In [5]: # Write code to show output here.
        import distsim
        import operator

        d=distsim.king_queen_analogy('king', 'man', 'woman')

        near_20=dict(sorted(d.iteritems(), key=operator.itemgetter(1), reverse=True)[:20]).keys(

        print "20 closest words for the analogy \"king : man :: ____ : woman\": ", "\n", near_20


        d2=distsim.king_queen_analogy('france', 'paris', 'madrid')

        near_20_d2=dict(sorted(d2.iteritems(), key=operator.itemgetter(1), reverse=True)[:20]).k

        print "20 closest words for the analogy \"france : paris :: __ : madrid\": ", "\n", near
```

```
20 closest words for the analogy "king : man :: ____ : woman":
['king', 'singer', 'woman', 'daughter', 'elizabeth', 'clark', 'mother', 'queen', 'royal', 'siste
```

```
20 closest words for the analogy "france : paris :: __ : madrid":
['ukraine', 'brazil', 'europe', 'korea', 'china', 'afghanistan', 'africa', 'india', 'france', 'a
```

2.6.1 Textual answer here.

Yes, the analogy does work. Some words from list of 20 closest are: **elizabeth, queen, royal, mary, princess** All of the words above allude to a woman of royalty and/or proper name of actual queens, such as *elizabeth, mary*. Analogy for my analysis is: **france is to paris as what is to madrid**? Therefore, in SAT-style notation: france : paris :: __ : madrid For my analogy, the method does work. The target answer is **spain**, a country. The closest words returned are almost all countries, which is what was desired. Besides **spain** being one of the words in the list of 20 closest words, some of the other countries present are *brazil, ukraine, korea*. There are also names of continents in the list, like *europe, africa*. The unsupervised algorithm here determined clusters of words which are related to each other. In a given tweet, people will talk about a particular topic only, which leads to usage of contextually related words in said tweet. For instance, in a tweet about countries to visit, many words used will be associated with countries or will be names of countries. If the tweet is about France, there is a good chance Paris will be mentioned too. So, the unsupervised algorithm obtains clusters alluding to relationships between words, such as that between names of countries, a country and its city, etc.