Homework 5: Lexicons and Distributional Semantics

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

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: Sanjay Reddy Satti (31315112)

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

- *name 1*

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.

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

As it wasn't explicitly mentioned. Few implementation assumptions:

1) If a tweet has both +ve and -ve word(s) the count for (word,+) and (word,-) is both increased by one

2) A tweet is considered as a set. Thus, if a tweet has two occurences of the same word, count(word) for that tweet is still taken as 1

3) log is taken with base 'e'

4) In PPMI, whenever the log value becomes negative, the term is equated to zero. As mentioned in JM textbook this cleanly solves the zero count problem (Note that log(0) is nothing but -ve infinity). But, we might lose valuable information because in PPMI 'ALL' -ve values are put to zero (This affects the polarity calculations. If PMI(word,NEG) is a -ve value this pushes the word towards having a higher polarity value). Another variation is to, calculate and use direct PMI values as usual. But whenever count becomes zero, simply put that PMI term to zero (In this variant, I'm considering -ve PMI values also... as long as it is not -ve infinity. When it is -ve infinity, simply put PMI to 0)

5) To deal with zero counts, count(word,context) values have been added with '1' when it is zero. But for calculating denominator, we need to find a way. One obvious way is to simply NOT add any additional term (The denominator simply remains as the number of tweets. To use this variant, simply put the 'assume' parameter as 0 in the function top_polarities. Laplace). Another way is to first

put the 'assume' parameter as 0 in the function top_polarities_Laplace). Another way is to first calculate the average number of words per tweet and also the number of words which have count(word,context) as zero. Then add the appropriate number of sentences. Using an example:

```
Number of words with count(word,POS_TERM)=0 is 20,857. (Example of s
uch a word is 'passively')

Number of words with count(word,NEG_TERM)=0 is 69,987. (Example of s
uch a word is 'Craziness')

Let smoothing factor be 1. That is, count(word,POS_TERM) is taken as
1 from now on instead of 0.
(There is one tweet in the corpus, which has the word and a POS_TERM
in it).

Average length of words per tweet is 15.487.

This can be interpreted as: Let us add 5866 (=(20857+69987)/15.487))
sentences to the corpus. These
will contain only those words which earlier had count as zero.

Thus,
Prob('passively',POS_TERM)=count('passively',POS_TERM)/(Total_tweets-
66)
Prob('passively',POS_TERM)=1.0/(174689+5866)
```

6) The submitted polar.py file has all the implementations of the variants mentioned above. I've used a variant of PPMI for the following questions. This calculates PMI as usual. Except that whenever, count is zero, the PMI is equated to zero. This version seemed to return good results (qualitatively). If laplacian or other variants are needed, please uncomment the lines and run the cells in this .ipynb file.

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 polar; reload(polar)

pos_words,neg_words=polar.top_polarities_PPMI(filename="tweets.txt",flag=1,
assume=0)
print "With NO thresholding. Using PPMI Variant"
print "Positive Terms: "+str([i[0] for i in pos_words])
print ""
print "Negative Terms: "+str([i[0] for i in neg_words])


#pos_words,neg_words=polar.top_polarities_PPMI(filename="tweets.txt",flag=1
```

```
sume=1)
#print "With NO thresholding. Using Direct PPMI"
#print "Positive Terms: "+str([i[0] for i in pos_words])
#print ""
#print "Negative Terms: "+str([i[0] for i in neg_words])

#pos_words,neg_words=polar.top_polarities_Laplace(filename="tweets.txt",fla
,assume=1)
#print "With NO thresholding. Using Laplace"
#print "Positive Terms: "+str([i[0] for i in pos_words])
#print ""
#print "Negative Terms: "+str([i[0] for i in neg_words])

#pos_words,neg_words=polar.top_polarities_Laplace(filename="tweets.txt",fla
,assume=0)
#print "With NO thresholding. Using Laplace variant"
#print "Positive Terms: "+str([i[0] for i in pos_words])
#print ""
#print "Negative Terms: "+str([i[0] for i in neg_words])
```

```
No. of tweets: 174689.0
With NO thresholding. Using PPMI Variant
Positive Terms: ['Decay', '23', 'Via', 'ol', 'Taylor', 'wishin', 'Stories',
'hosting', 'beneficial', '".....', 'fought', 'BEST', 'hood', 'pregnancy', '
format', '\xf0\x9f\x92\x98', 'Birthday', 'Shri', 'her\\', 'rebuilding', 'SC
ARY', 'Happy14thAnniversary', 'minimum', '\\n\\nMe', 'RTing', 'jungcook', '
visiting', '\xf0\x9f\xa4\x94\xf0\x9f\x98\x82', 'embraces', 'imagined', 'Ate
', 'Happy', 'admission', 'BIRTHDAY', 'Falling', '2009', 'lobbying', 'flippi
ng', 'Bowman', 'Stan', 'unfollowers', 'Chrissy', 'taehyung', 'xx', 'headass
', 'PROUD', 'Terry', 'described', 'Award', 'attract']

Negative Terms: ['FBI', 'hunger', '3am', 'anti-Semitism', 'moods', 'Kansas'
, 'Recent', 'doe\xe2\x80\xa6', 'marijuana', 'un\xe2\x80\xa6', 'unleashed',
'Victims', 'fee', 'Blame', 'introducing', 'Crime', 'then\xe2\x80\xa6', 'per
sist', 'DAMN', 'investigating', 'Snark', 'insomnia', 'understandable', 'lit
erate', 'tim\xe2\x80\xa6', 'disability', 'Sec', 'carrier', 'grieve', 'conde
mn', 'who\xe2\x80\xa6', 'condemns', 'crimes', 'JCC', 'LAUGH', 'IDK', 'Conde
mning', 'united', 'invites', 'deporting', 'American\xe2\x80\xa6', 'condemni
ng', 'Decide', 'swaddle', 'closure', 'decreased', 'engineer',
'Obama\xe2\x80\x99s', 'cigarettes', 'CHILD']
```

Textual answer here.

Decay has very high positive polarity. One reason might be because there is one tweet which has around 800 occurrences (Used grep). That tweet has the word 'good' in it.

Words which are commonly used to wish people, like 'wishin' or 'Happy' or 'Birthday' also have good positive score, which is to be expected (The sentences have many positive seeds). Words like 'Crime', 'condemn' or 'cigarettes' are usually embeded in sentences with -ve sense.

Words like 'Scary' and 'jungcook' are not actually positive and 'engineer' aren't exactly -ve polarity. One reason, could be the datasource itself. The timing these tweets are collected is important because Tweets are very dependent on the cultural events happeing during that time. As an example if 'Taylor' Swift has released a new popular catchy song she gets to be in the +ve terms. There will be many tweets referencing her thus increasing the PMI. 'Laugh' in negative doesn't make sense, but one reason for it might be sarcasm. This model simply looks at co-occurrences and uses statistics (counts of words in this case) and suffers from some of the same drawbacks as bag-of-words.

(counts of words in this case) and suffers from some of the same drawbacks as bag of words.

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 polar; reload(polar)

pos_words,neg_words=polar.top_polarities_PPMI(filename="tweets.txt",flag=0,
assume=0)
print "With thresholding. Using PPMI Variant"
print "Positive Terms: "+str([i[0] for i in pos_words])
print ""
print "Negative Terms: "+str([i[0] for i in neg_words])

#pos_words,neg_words=polar.top_polarities_PPMI(filename="tweets.txt",flag=0
sume=1)
#print "With thresholding. Using PPMI"
#print "Positive Terms: "+str([i[0] for i in pos_words])
#print ""
#print "Negative Terms: "+str([i[0] for i in neg_words])

#pos_words,neg_words=polar.top_polarities_Laplace(filename="tweets.txt",fla
,assume=1)
#print "With thresholding. Using Laplace"
#print "Positive Terms: "+str([i[0] for i in pos_words])
#print ""
#print "Negative Terms: "+str([i[0] for i in neg_words])

#pos_words,neg_words=polar.top_polarities_Laplace(filename="tweets.txt",fla
,assume=0)
#print "With thresholding. Using Laplace Variant"
#print "Positive Terms: "+str([i[0] for i in pos_words])
#print ""
#print "Negative Terms: "+str([i[0] for i in neg_words])
```

```
No. of tweets: 174689.0
With thresholding. Using PPMI Variant
Positive Terms: ['Decay', 'ol', 'Taylor', 'hood', 'pregnancy', 'Happy', 'bi
rthday', '\xe2\x9d\xa4\xef\xb8\x8f', '\xe2\x9d\xa4', 'D', 'hilarious', '\xf
0\x9f\x98\x8a', 'morning', 'Thank', 'Be', 'fall', 'thank', 'thanks', 'falli
ng', 'follow', 'beautiful', 'New', 'Thanks', 'amazing', 'Always', 'Nadine',
'soon', '\xf0\x9f\x98\x8d', 'you\\', 'proud', 'forever', ':)', 'happy', 'fe
els', 'support', '!!', 'deserve', 'sounds', 'together', 'movie', 'Have', 'G
od', 'vibes', 'ya', 'friend', 'smile', 'become', '!!!', '|', 'night']

Negative Terms: ['school', 'going', 'What', 'once', 'believe', 'something',
'use', 'woman', 'send', 'must', 'myself', 'says', 'ask', 'every', 'whole',
'country', 'Trump', 'women', 'negative', 'men', 'ppl', 'its', 'American', '
parents', 'went', 'Saf\xe2\x80\xa6', 'stand', 'w', 'after', 'Dems', 'agains
```

```
t', 'Why', 'Too', 'pick', 'ugly', 'evil', 'crime', 'store', 'explain', 'ME'
, 'forms', 'stands', 'beer', 'I\xe2\x80\x99ll', 'marijuana', 'crimes', 'uni
ted', 'condemning', 'cigarettes', 'CHILD']
```

Textual answer here.

As mentioned in JM textbook, PMI has the problem of being biased toward infrequent events; very rare words tend to have very high PMI values. This is clearly evident here. By removing rare words, the quality of the words as definitely increased.

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.

Textual answer here.

Emoticons and stop words and punctuation can be removed to get better accuracy (qualitatively). One can use regular expressions and nltk library to remove these.

Another important factor is to remove repetitions in tweets. This should increase the accuracy because MANY tweets are simply duplicated to create this dataset. Instead an actual dataset might provide a better insight. TO remove duplicates one might use a set() in Python.

As mentioned earlier, cultural aspect is also present when dealing with twitter data. As mentioned in the class, names like 'Bieber' are pretty common in tweets. One can probably run NER (Name Entity recognition) to maybe remove names like 'Taylor' (It is present in the top 50 polarity words).

Part-2: Distributional Semantics

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

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.)

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 [3]:

```
import distsim; reload(distsim)

word_to_ccdict = distsim.load_contexts("nytcounts.university_cat_dog")

# write code here to show output (i.e. cosine similarity between these word
s.)
# We encourage you to write other functions in distsim.py itself.
print "University and cat: "+str(distsim.cosine(word_to_ccdict['university'
],word_to_ccdict['cat']))
print "University and dog: "+str(distsim.cosine(word_to_ccdict['university'
],word_to_ccdict['dog']))
print "dog and cat: "+str(distsim.cosine(word_to_ccdict['dog'],word_to_ccdi
ct['cat']))
```

```
file nytcounts.university_cat_dog has contexts for 3 words
University and cat: 0.660442421144
University and dog: 0.659230248969
dog and cat: 0.966891672715
```

Write your response here:

Since both dog and cat are animals, the words occur in very similar sentences. Hence their contexts will be similar and hence have very high cosine similarity. University and animal are not synonyms in any way and hence have relatively lower cosine similarity. Another interesting fact is that, University and both animals have a similar cosine similarity.

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.)

This does make sense. Sparse vector representation of a word can be thought of as giving words frequently occuring in its context. Beautiful commonly occurs besdie 'quiet' or 'healthy'. This method is exceptionally good when it comes to proper nouns (like names).

In [4]:

```
import distsim; reload(distsim)
word_to_ccdict = distsim.load_contexts("nytcounts.4k")
###Provide your answer below; perhaps in another cell so you don't have to
reload the data each time
print "beautiful: "+str(distsim.NN('beautiful',word_to_ccdict,distsim.cosin
e))
print ""
print "edward: "+str(distsim.NN("edward",word_to_ccdict,distsim.cosine))
print ""
print "saved: "+str(distsim.NN('saved',word_to_ccdict,distsim.cosine))
print ""
print "red: "+str(distsim.NN('red',word_to_ccdict,distsim.cosine))
print ""
print "improve: "+str(distsim.NN('improve',word_to_ccdict,distsim.cosine))
print ""
print "Sleep: "+str(distsim.NN("sleep",word_to_ccdict,distsim.cosine))
```

```
file nytcounts.4k has contexts for 3648 words
beautiful: ['quiet', 'healthy', 'brilliant', 'simple', 'strange', 'handsome
', 'lovely', 'gorgeous', 'lonely', 'wonderful', 'strong', 'horrible', 'fant
astic', 'successful', 'terrible', 'weak', 'tight', 'massive', 'nasty', 'sha
rp']

edward: ['richard', 'robert', 'andrew', 'joseph', 'william', 'stephen', 'jo
hn', 'david', 'daniel', 'peter', 'james', 'charles', 'anthony', 'steven', '
christopher', 'susan', 'jonathan', 'kevin', 'alan', 'michael']

saved: ['accepted', 'rejected', 'reached', 'kept', 'made', 'raised', 'watch
```

```
ed', 'blocked', 'pushed', 'lost', 'ordered', 'won', 'missed', 'left', 'chan
ged', 'visited', 'joined', 'supported', 'stopped', 'given']

red: ['japanese', 'french', 'russian', 'british', 'german', 'chinese', 'u.s
.', 'senate', 'final', 'second', 'ball', 'national', 'pentagon', 'current',
'state', 'sun', 'traditional', 'movie', 'brain', 'past']

improve: ['expand', 'protect', 'keep', 'prevent', 'discuss', 'celebrate', '
avoid', 'win', 'sell', 'raise', 'save', 'leave', 'join', 'maintain', 'treat
', 'reduce', 'meet', 'attend', 'stop', 'describe']

Sleep: ['die', 'breathe', 'eat', 'play', 'work', 'shine', 'cheat', 'change'
, 'shout', 'grow', 'burn', 'drive', 'marry', 'cheer', 'pass', 'sing', 'hide
', 'shoot', 'stop', 'ride']
```

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 pretrained 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 [5]:

```
import distsim; reload(distsim)

#word_to_vec_dict = distsim.load_word2vec("nyt_word2vec.4k")
word_to_vec_dict = distsim.load_word2vec("nyt_word2vec.university_cat_dog")
# write code here to show output (i.e. cosine similarity between these word
s.)
# We encourage you to write other functions in distsim.py itself.
print "University and cat: "+str(distsim.cosine_np(word_to_vec_dict['univer
sity'],word_to_vec_dict['cat']))
print "University and dog: "+str(distsim.cosine_np(word_to_vec_dict['univer
sity'],word_to_vec_dict['dog']))
print "dog and cat: "+str(distsim.cosine_np(word_to_vec_dict['dog'],word_to
_vec_dict['cat']))
```

```
University and cat: -0.205394745036
University and dog: -0.190753135501
dog and cat: 0.827517295965
```

Write your response here:

As mentioned above, in many sentences cat and dog, both can be used and hence have similarity. It makes sense that University and animal have -ve cosine similarity, because they are essentially words from completely different domains.

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.

As mentioned, word2vec are very dense vectors. Compare this with normal counts where the vectors are VERY sparse. Considering an example from JM, a sparse vector cannot give the intuition that in the sentences (... word car ...) and (... word automobile ...) the 'word' is probably the same. But in dense vectors, because 'car' and 'automobile' are NOT specific columns, this intuition can be gained. All the similar words in the earlier case give us a vague idea about the nearby words, but in dense vectors case, we are getting much better synonyms. Sparse vectors give us context, but dense vectors try to give words with which the current word can be substituted with.

Consider 'beautiful':

In Sparse: 'quiet','healthy': The scenery is very beautiful and quiet (The words are used frequently together).
In Dense: 'lovely','wonderful': 'Lovely' is a good synonym of 'beautiful'. It is a step above than simply looking at context

Consider 'red':

In Sparse, it is simply returning where 'red' is used (All the countries mentioned like 'Japan', 'France' have red in their flags).
In Dense however, it recognized 'red' is a color and is returning similar words.

Consider 'edward':

In this case, both are more or less same. Infact I would say sparse is better giving ALL similar words as proper nouns (In Dense, it is returning incorrect tokens like 't.' and 'c.')

Consider 'improve', 'saved', 'sleep':

In this case both are equally bad. Sparse has antonyms like 'stop' (expected because it is looking just into context and we do use 'stop increasing' a lot) and Dense has antonyms like 'limit'. The same with 'saved' Dense managed to get irrelevant words like 'hurt' and 'gotten'. In sparse it is simply returning common verbs used along with 'saved'. Same is the case with 'sleep'. Both are returning more or less the same words.

In [6]:

```
import distsim
```

```
word_to_vec_dict = distsim.load_word2vec("nyt_word2vec.4k")
###Provide your answer below
print "beautiful: "+str(distsim.NN('beautiful',word_to_vec_dict,distsim.cos
ine_np))
print ""
print "edward: "+str(distsim.NN("edward",word_to_vec_dict,distsim.cosine_np
))
print ""
print "saved: "+str(distsim.NN('saved',word_to_vec_dict,distsim.cosine_np))
print ""
print "red: "+str(distsim.NN('red',word_to_vec_dict,distsim.cosine_np))
print ""
print "improve: "+str(distsim.NN('improve',word_to_vec_dict,distsim.cosine_
np))
print ""
print "Sleep: "+str(distsim.NN("sleep",word_to_vec_dict,distsim.cosine_np))
```

```
beautiful: ['lovely', 'wonderful', 'gorgeous', 'strange', 'weird', 'sexy',
'fantastic', 'bright', 'cute', 'beauty', 'amazing', 'lonely', 'scary', 'hil
arious', 'handsome', 'cool', 'silly', 'nice', 'boring', 'literally']

edward: ['william', 'charles', 'robert', 'andrew', 'stephen', 'joseph', 'ri
chard', 'alan', 'steven', 'daniel', 'lawrence', 'peter', 'anthony', 'jonath
an', 'susan', 't.', 'c.', 'thomas', 'john', 'henry']

saved: ['lost', 'collected', 'loved', 'hurt', 'dropped', 'missed', 'paid',
'changed', 'removed', 'bought', 'learned', 'noticed', 'save', 'gotten', 'di
scovered', 'brought', 'survived', 'raised', 'liked', 'returned']

red: ['blue', 'yellow', 'pink', 'wings', 'sox', 'green', 'ice', 'shirt', 's
ky', 'jeans', 'thick', 'black', 'leather', 'gold', 'white', 'bowl', 'bar',
'shirts', 'gray', 'cross']

improve: ['maintain', 'reduce', 'expand', 'develop', 'fix', 'handle', 'rais
e', 'provide', 'continue', 'extend', 'apply', 'keep', 'limit', 'respond', '
help', 'protect', 'ignore', 'avoid', 'gain', 'allow']

Sleep: ['eat', 'die', 'sleeping', 'breathe', 'sit', 'eating', 'bed', 'stick
', 'stay', 'get', 'awake', 'drink', 'kiss', 'leave', 'stomach', 'crying', '
wear', 'go', 'shoot', 'lie']
```

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 [7]:

```
# Write code to show output here.
data = distsim.load_word2vec("nyt_word2vec.4k")
print "Supposed Queen: "+str(distsim.NN(None,data,distsim.cosine_np,dict2=d
istsim.linear_expression(data['king'],data['man'],data['woman'])))
print ""
print "Supposed sister: "+str(distsim.NN(None,data,distsim.cosine_np,dict2=
distsim.linear_expression(data['brother'],data['man'],data['woman'])))
print ""
print "Supposed better: "+str(distsim.NN(None,data,distsim.cosine_np,dict2=
distsim.linear_expression(data['bad'],data['worse'],data['good'])))
```

```
Supposed Queen: ['king', 'queen', 'princess', 'prince', 'lord', 'royal', 'w
oman', 'mary', 'mama', 'daughter', 'singer', 'kim', 'elizabeth', 'girl', 'g
randma', 'sister', 'mother', 'clark', 'wedding', 'husband']

Supposed sister: ['sister', 'daughter', 'brother', 'husband',
'grandmother', 'son', 'mother', 'wife', 'grandfather', 'boyfriend', 'cousin
', 'uncle', 'girlfriend', 'father', 'sisters', 'sons', 'grandchildren', 'gr
andma', 'dad', 'friend']

Supposed better: ['good', 'bad', 'tough', 'nice', 'great', 'big', 'dumb', '
stupid', 'weird', 'smart', 'lucky', 'terrible', 'happy', 'hard', 'strong',
'positive', 'hungry', 'wonderful', 'easy', 'healthy']
```

Textual answer here.

Brother:man:: sister:woman works as expected (infact the result is better than King: Man:: Queen: Woman).

Bad:Worse::Good:Better gives expected results (although the expected 'better' word is missing)

I believe the answer captures the esssence. As mentioned on GloVe, the training here is done on global word-word co-occurence statistics of a corpus and the vectors can be thought of capturing the linear substructures of words. The difference between the vectors can be thought of capturing the meaning specified by the juxtaposition of the words. So, subtraction between King and man in a sense captures, how replacing the word 'King' with 'man' will work in a sentence.

"You shall know a word by the company it keeps!" indeed. By quantifying the context of the word, word2vec is proving it right.