Assignment 1: Exploring Word Vectors (23 Points)

Estimated Time: ~3 hours

The objective of this assignment is to warm-up you of some python coding, and also get you to familarize with some NLP concepts.

In [1]:

```
import sys
assert sys.version_info[0]==3
assert sys.version_info[1] >= 5

from gensim.models import KeyedVectors
from gensim.test.utils import datapath
import pprint
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [6, 5]
import nltk
nltk.download('reuters')
from nltk.corpus import reuters
import numpy as np
import random
import scipy as sp
from sklearn.decomposition import TruncatedSVD
from sklearn.decomposition import PCA

START_TOKEN = '<START>'
END_TOKEN = '<END>'

np.random.seed(0)
random.seed(0)
```

```
[nltk_data] Downloading package reuters to /root/nltk_data...
```

Word Vectors

Word Vectors are often used as a fundamental component for downstream NLP tasks, e.g. question answering, text generation, translation, etc., so it is important to build some intuitions as to their strengths and weaknesses. Here, you will explore two types of word vectors: those derived from *co-occurrence matrices*, and those derived via *GloVe*.

Note on Terminology: The terms "word vectors" and "word embeddings" are often used interchangeably. The term "embedding" refers to the fact that we are encoding aspects of a word's meaning in a lower dimensional space. As [Wikipedia (https://en.wikipedia.org/wiki/Word_embedding)](https://en.wikipedia.org/wiki/Word_embedding) states, "*conceptually it involves a mathematical embedding from a space with one dimension per word to a continuous vector space with a much lower dimension*".

Part 1: Count-Based Word Vectors (10 points)

Most word vector models start from the following idea:

You shall know a word by the company it keeps ([Firth, J. R. 1957:11 (https://en.wikipedia.org/wiki/John_Rupert_Firth)](https://en.wikipedia.org/wiki/John_Rupert_Firth))

Many word vector implementations are driven by the idea that similar words, i.e., (near) synonyms, will be used in similar contexts. As a result, similar words will often be spoken or written along with a shared subset of words, i.e., contexts. By examining these contexts, we can try to develop embeddings for our words. With this intuition in mind, many "old school" approaches to constructing word vectors relied on word counts. Here we elaborate upon one of those strategies, *co-occurrence matrices* (for more information, see [here (http://web.stanford.edu/class/cs124/lec/vectorsemantics.video.pdf)](http://web.stanford.edu/class/cs124/lec/vectorsemantics.video.pdf) or [here (https://medium.com/data-science-group-iitr/word-embedding-2d05d270b285)](https://medium.com/data-science-group-iitr/word-embedding-2d05d270b285)).

Co-Occurrence

A co-occurrence matrix counts how often things co-occur in some environment. Given some word w_i occurring in the document, we consider the *context window* surrounding w_i. Supposing our fixed window size is n, then this is the n preceding and n subsequent words in that document, i.e. words $w_{i-n} \dots w_{i-1}$ and $w_{i+1} \dots w_{i+n}$. We build a *co-occurrence matrix* M, which is a symmetric word-by-word matrix in which M_{ij} is the number of times w_j appears inside w_i's window among all documents.

Example: Co-Occurrence with Fixed Window of n=1:

Document 1: "all that glitters is not gold"

Document 2: "all is well that ends well"

*	`<START>`	all	that	glitters	is	not	gold	well	ends	`<END>`
`<START>`	0	2	0	0	0	0	0	0	0	0
all	2	0	1	0	1	0	0	0	0	0
that	0	1	0	1	0	0	0	1	1	0
glitters	0	0	1	0	1	0	0	0	0	0
is	0	1	0	1	0	1	0	1	0	0
not	0	0	0	0	1	0	1	0	0	0
gold	0	0	0	0	0	1	0	0	0	1
well	0	0	1	0	1	0	0	0	1	1
ends	0	0	1	0	0	0	0	1	0	0
`<END>`	0	0	0	0	0	0	1	1	0	0

Note: In NLP, we often add `<START>` and `<END>` tokens to represent the beginning and end of sentences, paragraphs or documents. In thise case we imagine `<START>` and `<END>` tokens encapsulating each document, e.g., " `<START>` All that glitters is not gold `<END>` ", and include these tokens in our co-occurrence counts.

The rows (or columns) of this matrix provide one type of word vectors (those based on word-word co-occurrence), but the vectors will be large in general (linear in the number of distinct words in a corpus). Thus, our next step is to run *dimensionality reduction*. In particular, we will run *SVD (Singular Value Decomposition)*, which is a kind of generalized *PCA (Principal Components Analysis)* to select the top k principal components. Here's a visualization of dimensionality reduction with SVD. In this picture our co-occurrence matrix is A with n rows corresponding to n words. We obtain a full matrix decomposition, with the singular values ordered in the diagonal S matrix, and our new, shorter length-k word vectors in U_k.

This reduced-dimensionality co-occurrence representation preserves semantic relationships between words, e.g. *doctor* and *hospital* will be closer than *doctor* and *dog*.

Notes: If you can barely remember what an eigenvalue is, here's a slow, friendly introduction to SVD (https://davetang.org/file/Singular_Value_Decomposition_Tutorial.pdf). Though, for the purpose of this class, you only need to know how to extract the k-dimensional embeddings by utilizing pre-programmed implementations of these algorithms from the numpy, scipy, or sklearn python packages. In practice, it is challenging to apply full SVD to large corpora because of the memory needed to perform PCA or SVD. However, if you only want the top k vector components for relatively small k — known as Truncated SVD (https://en.wikipedia.org/wiki/Singular_value_decomposition#Truncated_SVD) — then there are reasonably scalable techniques to compute those iteratively.

Plotting Co-Occurrence Word Embeddings

Here, we will be using the Reuters (business and financial news) corpus. If you haven't run the import cell at the top of this page, please run it now (click it and press SHIFT-RETURN). The corpus consists of 10,788 news documents totaling 1.3 million words. These documents span 90 categories and are split into train and test. For more details, please see https://www.nltk.org/book/ch02.html (https://www.nltk.org/book/ch02.html). We provide a `read_corpus` function below that pulls out only articles from the "crude" (i.e. news articles about oil, gas, etc.) category. The function also adds `<START>` and `<END>` tokens to each of the documents, and lowercases words. You do **not** have to perform any other kind of pre-processing.

In [2]:

```
!unzip /root/nltk_data/corpora/reuters.zip -d /root/nltk_data/corpora/.
```

```
Streaming output truncated to the last 5000 lines.
  inflating: /root/nltk_data/corpora/./reuters/training/2231
  inflating: /root/nltk_data/corpora/./reuters/training/2232
  inflating: /root/nltk_data/corpora/./reuters/training/2234
  inflating: /root/nltk_data/corpora/./reuters/training/2236
  inflating: /root/nltk_data/corpora/./reuters/training/2237
  inflating: /root/nltk_data/corpora/./reuters/training/2238
  inflating: /root/nltk_data/corpora/./reuters/training/2239
  inflating: /root/nltk_data/corpora/./reuters/training/2240
  inflating: /root/nltk_data/corpora/./reuters/training/2244
  inflating: /root/nltk_data/corpora/./reuters/training/2246
  inflating: /root/nltk_data/corpora/./reuters/training/2247
  inflating: /root/nltk_data/corpora/./reuters/training/2249
  inflating: /root/nltk_data/corpora/./reuters/training/225
  inflating: /root/nltk_data/corpora/./reuters/training/2251
  inflating: /root/nltk_data/corpora/./reuters/training/2252
  inflating: /root/nltk_data/corpora/./reuters/training/2253
  inflating: /root/nltk_data/corpora/./reuters/training/2257
  inflating: /root/nltk_data/corpora/./reuters/training/2259
```

In [3]:

```
def read_corpus(category="crude"):
    """ Read files from the specified Reuter's category.
        Params:
            category (string): category name
        Return:
            list of lists, with words from each of the processed files
    """
    files = reuters.fileids(category)
    return [[START_TOKEN] + [w.lower() for w in list(reuters.words(f))] + [END_TOKEN
```

Let's have a look what these documents are like….

In [4]:

```
reuters_corpus = read_corpus()
pprint.pprint(reuters_corpus[:3], compact=True, width=100)
```

```
  'the', 'rise', 'in', 'the', 'value', 'of', 'the', 'yen', 'and', 'a',
'decline', 'in', 'domestic',
  'electric', 'power', 'demand', '.', 'miti', 'is', 'planning', 'to',
'work', 'out', 'a', 'revised',
  'energy', 'supply', '/', 'demand', 'outlook', 'through', 'deliberati
ons', 'of', 'committee',
  'meetings', 'of', 'the', 'agency', 'of', 'natural', 'resources', 'an
d', 'energy', ',', 'the',
  'officials', 'said', '.', 'they', 'said', 'miti', 'will', 'also', 'r
eview', 'the', 'breakdown',
  'of', 'energy', 'supply', 'sources', ',', 'including', 'oil', ',',
'nuclear', ',', 'coal', 'and',
  'natural', 'gas', '.', 'nuclear', 'energy', 'provided', 'the', 'bul
k', 'of', 'japan', "'", 's',
  'electric', 'power', 'in', 'the', 'fiscal', 'year', 'ended', 'marc
h', '31', ',', 'supplying',
  'an', 'estimated', '27', 'pct', 'on', 'a', 'kilowatt', '/', 'hour',
'basis', ',', 'followed',
  'by', 'oil', '(', '23', 'pct', ')', 'and', 'liquefied', 'natural',
```

Question 1.1: Implement `distinct_words` [code] (2 points)

Write a method to work out the distinct words (word types) that occur in the corpus. You can do this with `for` loops, but it's more efficient to do it with Python list comprehensions. In particular, [this (https://coderwall.com/p/rcmaea/flatten-a-list-of-lists-in-one-line-in-python)](https://coderwall.com/p/rcmaea/flatten-a-list-of-lists-in-one-line-in-python) may be useful to flatten a list of lists. If you're not familiar with Python list comprehensions in general, here's [more information (https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Comprehensions.html)](https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Comprehensions.html).

Your returned `corpus_words` should be sorted. You can use python's `sorted` function for this.

You may find it useful to use [Python sets (https://www.w3schools.com/python/python_sets.asp)](https://www.w3schools.com/python/python_sets.asp) to remove duplicate words.

In [5]:

```
def distinct_words(corpus):
    """ Determine a list of distinct words for the corpus.
        Params:
            corpus (list of list of strings): corpus of documents
        Return:
            corpus_words (list of strings): sorted list of distinct words across the
            num_corpus_words (integer): number of distinct words across the corpus
    """
    corpus_words = []
    num_corpus_words = -1

    # ------------------
    # Write your implementation here.

    corpus_words = [word for line in corpus for word in line] # 1st loop --> for lin
    corpus_words = list(set(corpus_words))
    corpus_words.sort(reverse=False)
    num_corpus_words = len(corpus_words)

    # ------------------

    return corpus_words, num_corpus_words
```

In [6]:

```
# ---------------------
# Run this sanity check
# Note that this not an exhaustive check for correctness.
# ---------------------

# Define toy corpus
test_corpus = ["{} All that glitters isn't gold {}".format(START_TOKEN, END_TOKEN).s
test_corpus_words, num_corpus_words = distinct_words(test_corpus)

# Correct answers
ans_test_corpus_words = sorted([START_TOKEN, "All", "ends", "that", "gold", "All's",
ans_num_corpus_words = len(ans_test_corpus_words)

# Test correct number of words
assert(num_corpus_words == ans_num_corpus_words), "Incorrect number of distinct word

# Test correct words
assert (test_corpus_words == ans_test_corpus_words), "Incorrect corpus_words.\nCorre

# Print Success
print ("-" * 80)
print("Passed All Tests!")
print ("-" * 80)
```

```
----------------------------------------------------------------------
----------
Passed All Tests!
----------------------------------------------------------------------
----------
```

Question 1.2: Implement `compute_co_occurrence_matrix` [code] (3 points)

points)

Write a method that constructs a co-occurrence matrix for a certain window-size n (with a default of 4), considering words n before and n after the word in the center of the window. Here, we start to use `numpy (np)` to represent vectors, matrices, and tensors.

In [7]:

```
def compute_co_occurrence_matrix(corpus, window_size=4):
    """ Compute co-occurrence matrix for the given corpus and window_size (default c

        Note: Each word in a document should be at the center of a window. Words nea
              number of co-occurring words.

              For example, if we take the document "<START> All that glitters is not
              "All" will co-occur with "<START>", "that", "glitters", "is", and "not

        Params:
            corpus (list of list of strings): corpus of documents
            window_size (int): size of context window
        Return:
            M (a symmetric numpy matrix of shape (number of unique words in the corp
                Co-occurence matrix of word counts.
                The ordering of the words in the rows/columns should be the same as
            word2ind (dict): dictionary that maps word to index (i.e. row/column num
    """
    words, num_words = distinct_words(corpus)
    M = None
    word2ind = {}

    # ------------------
    # Write your implementation here.

    start = None
    end = None

    M = np.zeros((num_words, num_words))
    word2ind = {word:index for index,word in enumerate(words)}
    #print(word2ind)

    for sentense in corpus:
        total = len(sentense)
      #print(sentense)
      #print(total)

        for i in range(len(sentense)):
        current_word = sentense[i]

        if (i-window_size > 0):
            start = (i-window_size)
        else:
            start = 0
        if (i+window_size <= total):
            end = (i+window_size)
        else:
            start = total

        window_words = sentense[start:i] + sentense[i+1:end+1]

        for w in window_words:
            M[word2ind[current_word]][word2ind[w]] += 1

    # ------------------

    return M, word2ind
```

In [8]:

```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness.
# ---------------------

# Define toy corpus and get student's co-occurrence matrix
test_corpus = ["{} All that glitters isn't gold {}".format(START_TOKEN, END_TOKEN).s
M_test, word2ind_test = compute_co_occurrence_matrix(test_corpus, window_size=1)

# Correct M and word2ind
M_test_ans = np.array(
    [[0., 0., 0., 0., 0., 0., 1., 0., 0., 1.,],
     [0., 0., 1., 1., 0., 0., 0., 0., 0., 0.,],
     [0., 1., 0., 0., 0., 0., 0., 0., 1., 0.,],
     [0., 1., 0., 0., 0., 0., 0., 0., 0., 1.,],
     [0., 0., 0., 0., 0., 0., 0., 0., 1., 1.,],
     [0., 0., 0., 0., 0., 0., 0., 1., 1., 0.,],
     [1., 0., 0., 0., 0., 0., 0., 1., 0., 0.,],
     [0., 0., 0., 0., 0., 1., 1., 0., 0., 0.,],
     [0., 0., 1., 0., 1., 1., 0., 0., 0., 1.,],
     [1., 0., 0., 1., 1., 0., 0., 0., 1., 0.,]]
)
ans_test_corpus_words = sorted([START_TOKEN, "All", "ends", "that", "gold", "All's",
word2ind_ans = dict(zip(ans_test_corpus_words, range(len(ans_test_corpus_words))))

# Test correct word2ind
assert (word2ind_ans == word2ind_test), "Your word2ind is incorrect:\nCorrect: {}\nY

# Test correct M shape
assert (M_test.shape == M_test_ans.shape), "M matrix has incorrect shape.\nCorrect:

# Test correct M values
for w1 in word2ind_ans.keys():
    idx1 = word2ind_ans[w1]
    for w2 in word2ind_ans.keys():
        idx2 = word2ind_ans[w2]
        student = M_test[idx1, idx2]
        correct = M_test_ans[idx1, idx2]
        if student != correct:
            print("Correct M:")
            print(M_test_ans)
            print("Your M: ")
            print(M_test)
            raise AssertionError("Incorrect count at index ({}, {})=({}, {}) in matr

# Print Success
print ("-" * 80)
print("Passed All Tests!")
print ("-" * 80)
```

```
----------------------------------------------------------------------
----------
Passed All Tests!
----------------------------------------------------------------------
----------
```

Question 1.3: Implement `reduce_to_k_dim` [code] (1 point)

Construct a method that performs dimensionality reduction on the matrix to produce k-dimensional embeddings. Use SVD to take the top k components and produce a new matrix of k-dimensional embeddings.

Note: All of numpy, scipy, and scikit-learn (`sklearn`) provide *some* implementation of SVD, but only scipy and sklearn provide an implementation of Truncated SVD, and only sklearn provides an efficient randomized algorithm for calculating large-scale Truncated SVD. So please use [sklearn.decomposition.TruncatedSVD (https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html)](https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html).

In [9]:

```
def reduce_to_k_dim(M, k=2):
    """ Reduce a co-occurence count matrix of dimensionality (num_corpus_words, num_
        to a matrix of dimensionality (num_corpus_words, k) using the following SVD
            - http://scikit-learn.org/stable/modules/generated/sklearn.decompositior

        Params:
            M (numpy matrix of shape (number of unique words in the corpus , number
            k (int): embedding size of each word after dimension reduction
        Return:
            M_reduced (numpy matrix of shape (number of corpus words, k)): matrix of
                    In terms of the SVD from math class, this actually returns U * S
    """
    n_iters = 10     # Use this parameter in your call to `TruncatedSVD`
    M_reduced = None
    print("Running Truncated SVD over %i words..." % (M.shape[0]))

    # ------------------
    # Write your implementation here.

    svd = TruncatedSVD(n_components=k, n_iter=n_iters, random_state=42)
    M_reduced = svd.fit_transform(M)

    # ------------------

    print("Done.")
    return M_reduced
```

In [10]:

```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness
# In fact we only check that your M_reduced has the right dimensions.
# ---------------------

# Define toy corpus and run student code
test_corpus = ["{} All that glitters isn't gold {}".format(START_TOKEN, END_TOKEN).s
M_test, word2ind_test = compute_co_occurrence_matrix(test_corpus, window_size=1)
M_test_reduced = reduce_to_k_dim(M_test, k=2)

# Test proper dimensions
assert (M_test_reduced.shape[0] == 10), "M_reduced has {} rows; should have {}".form
assert (M_test_reduced.shape[1] == 2), "M_reduced has {} columns; should have {}".fc

# Print Success
print ("-" * 80)
print("Passed All Tests!")
print ("-" * 80)
```

```
Running Truncated SVD over 10 words...
Done.
----------------------------------------------------------------------
----------
Passed All Tests!
----------------------------------------------------------------------
----------
```

Question 1.4: Implement `plot_embeddings` [code] (1 point)

Here you will write a function to plot a set of 2D vectors in 2D space. For graphs, we will use Matplotlib (`plt`).

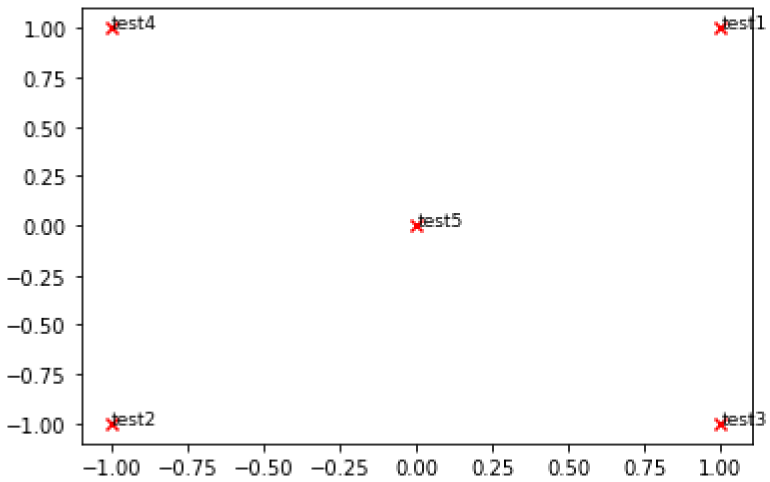
In [11]:

```
def plot_embeddings(M_reduced, word2ind, words):
    """ Plot in a scatterplot the embeddings of the words specified in the list "wor
        NOTE: do not plot all the words listed in M_reduced / word2ind.
        Include a label next to each point.

        Params:
            M_reduced (numpy matrix of shape (number of unique words in the corpus ,
            word2ind (dict): dictionary that maps word to indices for matrix M
            words (list of strings): words whose embeddings we want to visualize
    """

    # ------------------
    # Write your implementation here.

    for w in words:
        x = M_reduced[word2ind[w]][0]
        y = M_reduced[word2ind[w]][1]
        plt.scatter(x, y, marker='x', color='red')
        plt.text(x, y, w, fontsize=9)
    plt.show()

    # ------------------
```

In [12]:

```
# ----------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness.
# The plot produced should look like the "test solution plot" depicted below.
# ----------------------

print ("-" * 80)
print ("Outputted Plot:")

M_reduced_plot_test = np.array([[1, 1], [-1, -1], [1, -1], [-1, 1], [0, 0]])
word2ind_plot_test = {'test1': 0, 'test2': 1, 'test3': 2, 'test4': 3, 'test5': 4}
words = ['test1', 'test2', 'test3', 'test4', 'test5']
plot_embeddings(M_reduced_plot_test, word2ind_plot_test, words)

print ("-" * 80)
```

```
----------------------------------------------------------------------
----------
Outputted Plot:
```




```
----------------------------------------------------------------------
----------
```

Test Plot Solution

Question 1.5: Co-Occurrence Plot Analysis [written] (3 points)

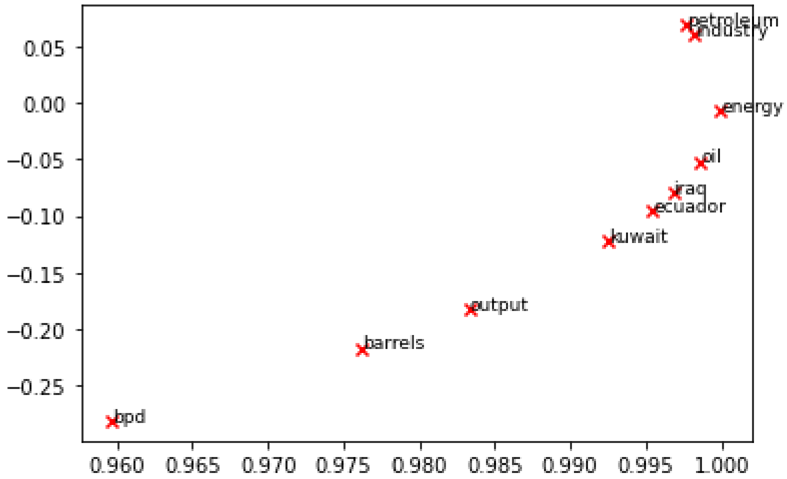
Now we will put together all the parts you have written! We will compute the co-occurrence matrix with fixed window of 4 (the default window size), over the Reuters "crude" (oil) corpus. Then we will use TruncatedSVD to compute 2-dimensional embeddings of each word. TruncatedSVD returns U*S, so we need to normalize the returned vectors, so that all the vectors will appear around the unit circle (therefore closeness is directional closeness). **Note**: The line of code below that does the normalizing uses the NumPy concept of *broadcasting*. If you don't know about broadcasting, check out [Computation on Arrays: Broadcasting by Jake VanderPlas (https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html)](https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html).

Run the below cell to produce the plot. It'll probably take a few seconds to run. What clusters together in 2-dimensional embedding space? What doesn't cluster together that you might think should have? **Note:** "bpd" stands for "barrels per day" and is a commonly used abbreviation in crude oil topic articles.

In [13]:

```
# ------------------------------
# Run This Cell to Produce Your Plot
# ------------------------------
reuters_corpus = read_corpus()
M_co_occurrence, word2ind_co_occurrence = compute_co_occurrence_matrix(reuters_corpu
M_reduced_co_occurrence = reduce_to_k_dim(M_co_occurrence, k=2)

# Rescale (normalize) the rows to make them each of unit-length
M_lengths = np.linalg.norm(M_reduced_co_occurrence, axis=1)
M_normalized = M_reduced_co_occurrence / M_lengths[:, np.newaxis] # broadcasting

words = ['barrels', 'bpd', 'ecuador', 'energy', 'industry', 'kuwait', 'oil', 'output

plot_embeddings(M_normalized, word2ind_co_occurrence, words)
```

```
Running Truncated SVD over 8185 words...
Done.

/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:10: Runti
meWarning: invalid value encountered in true_divide
  # Remove the CWD from sys.path while we load stuff.
```



Write your answer here.

cluster1 : petroleum and industry

cluster2 : energy and oil

cluster3 : ecuador, iraq and kuwait(these are country names)

cluster4 : barrels and outputs

cluster4 : bpd

In my opinion, bpd is far way from barrels and should be near with barrels and output because bpd stands for "barrels per day". Petrolum should be near with energy and oil more than industry since these 3 are types of energy.

Part 2: Prediction-Based Word Vectors (13 points)

As discussed in class, more recently prediction-based word vectors have demonstrated better performance, such as word2vec and GloVe (which also utilizes the benefit of counts). If you're feeling adventurous, challenge yourself and try reading [GloVe's original paper (https://nlp.stanford.edu/pubs/glove.pdf)](https://nlp.stanford.edu/pubs/glove.pdf).

Then run the following cells to load the GloVe vectors into memory. **Note**: If this is your first time to run these cells, i.e. download the embedding model, it will take a couple minutes to run. If you've run these cells before, rerunning them will load the model without redownloading it, which will take about 1 to 2 minutes.

In [18]:

```
def load_embedding_model():
    """ Load GloVe Vectors
        Return:
            word_vectors: All 400000 embeddings, each lengh 100
    """
    import gensim.downloader as api
    word_vectors = api.load("glove-wiki-gigaword-300")
    print("Loaded vocab size %i" % len(word_vectors.vocab.keys()))
    return word_vectors
```

In [19]:

```
# -----------------------------------
# Run Cell to Load Word Vectors
# Note: This will take a couple minutes
# -----------------------------------
word_vectors = load_embedding_model()
type(word_vectors)
```

```
Loaded vocab size 400000
```

Out[19]:

```
gensim.models.keyedvectors.Word2VecKeyedVectors
```

In [20]:

```
# try some function most_similar
word_vectors.most_similar('queen')
```

Out[20]:

```
[('elizabeth', 0.6771447658538818),
 ('princess', 0.635676383972168),
 ('king', 0.6336469650268555),
 ('monarch', 0.5814188122749329),
 ('royal', 0.543052613735199),
 ('majesty', 0.5350357294082642),
 ('victoria', 0.5239557027816772),
 ('throne', 0.5097099542617798),
 ('lady', 0.5045416355133057),
 ('crown', 0.49980056285858154)]
```

Note: If you are receiving a "reset by peer" error, rerun the cell to restart the download.

Cosine Similarity

Now that we have word vectors, we need a way to quantify the similarity between individual words, according to these vectors. One such metric is cosine-similarity. We will be using this to find words that are "close" and "far" from one another.

We can think of n-dimensional vectors as points in n-dimensional space. If we take this perspective [L1 (http://mathworld.wolfram.com/L1-Norm.html)](http://mathworld.wolfram.com/L1-Norm.html) and [L2 (http://mathworld.wolfram.com/L2-Norm.html)](http://mathworld.wolfram.com/L2-Norm.html) Distances help quantify the amount of space "we must travel" to get between these two points. Another approach is to examine the angle between two vectors. From trigonometry we know that:

Instead of computing the actual angle, we can leave the similarity in terms of $similarity = cos(\Theta)$. Formally the [Cosine Similarity (https://en.wikipedia.org/wiki/Cosine_similarity)](https://en.wikipedia.org/wiki/Cosine_similarity) s between two vectors p and q is defined as:

$$s = \frac{p \cdot q}{||p||||q||}, \text{ where } s \in [-1, 1]$$

Question 2.1: Words with Multiple Meanings (1.5 points) [code + written]

Polysemes and homonyms are words that have more than one meaning (see this [wiki page (https://en.wikipedia.org/wiki/Polysemy)](https://en.wikipedia.org/wiki/Polysemy) to learn more about the difference between polysemes and homonyms). Find a word with *at least two different meanings* such that the top-10 most similar words (according to cosine similarity) contain related words from *both* meanings. For example, "leaves" has both "go_away" and "a_structure_of_a_plant" meaning in the top 10, and "scoop" has both "handed_waffle_cone" and "lowdown". You will probably need to try several polysemous or homonymic words before you find one.

Please state the word you discover and the multiple meanings that occur in the top 10. Why do you think many of the polysemous or homonymic words you tried didn't work (i.e. the top-10 most similar words only contain **one** of the meanings of the words)?

Note: You should use the `word_vectors.most_similar(word)` function to get the top 10 similar words. This function ranks all other words in the vocabulary with respect to their cosine similarity to the given word. For further assistance, please check the **[GenSim documentation (https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.FastTextKey](https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.FastTextKey)**

In [24]:

```
# ------------------
# Write your implementation here.

pprint.pprint(word_vectors.most_similar('organ'))
print('----------------------------------')
pprint.pprint(word_vectors.most_similar('hatch'))


# ------------------
```

```
[('organs', 0.7157173156738281),
 ('transplants', 0.5657877922058105),
 ('piano', 0.4991111755371094),
 ('transplant', 0.4965413212776184),
 ('harpsichord', 0.49085456132888794),
 ('transplantation', 0.4844741225242615),
 ('instrument', 0.4700425863265991),
 ('donor', 0.4571865200996399),
 ('choir', 0.4560093283653259),
 ('wurlitzer', 0.4549233317375183)]
----------------------------------
[('orrin', 0.6428661346435547),
 ('hatches', 0.4678693115711212),
 ('sen.', 0.4528954029083252),
 ('keyes', 0.3940433263778686 5),
 ('senator', 0.38685500621795654),
 ('chicks', 0.37095874547958374),
 ('cornyn', 0.359639048576355),
 ('lugar', 0.3583526909351349),
 ('senate', 0.3581147789955139),
 ('airlock', 0.3567692041397095)]
```

In [25]:

```
pprint.pprint(word_vectors.most_similar('light'))
```

```
[('lights', 0.5719754695892334),
 ('bright', 0.5631998777389526),
 ('dark', 0.5300681591033936),
 ('sunlight', 0.5233156681060791),
 ('lighter', 0.5068577527999878),
 ('blue', 0.4855985641479492),
 ('heavy', 0.4685371518135071),
 ('ultraviolet', 0.4530498683452606),
 ('colored', 0.4443990290164947 5),
 ('shine', 0.44215908646583557)]
```

Write your answer here.

I discovered in the word "organ" which had "transplant" and "instrument".In the sencond one "hatch", it had "chicks" and "airlock".

Many of the polysemous or homonymic words I tried didn't work. As an example, "dark" and "heavy" appear in the top 10 of the word "light" because "light" is opposite word of "dark" and "heavy" and the embedding will be closer to these two words in the vector.

Question 2.3: Synonyms & Antonyms (2 points) [code + written]

When considering Cosine Similarity, it's often more convenient to think of Cosine Distance, which is simply 1 - Cosine Similarity.

Find three words (w_1, w_2, w_3) where w_1 and w_2 are synonyms and w_1 and w_3 are antonyms, but Cosine Distance $(w_1, w_3) <$ Cosine Distance (w_1, w_2).

As an example, w_1="happy" is closer to w_3="sad" than to w_2="cheerful". Please find a different example that satisfies the above. Once you have found your example, please give a possible explanation for why this counter-intuitive result may have happened.

You should use the the `word_vectors.distance(w1, w2)` function here in order to compute the cosine distance between two words. Please see the **GenSim documentation (https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.FastTextKey** for further assistance.

In [26]:

```
    # ------------------
    # Write your implementation here.

w1 = "thin"
w2 = "slim"
w3 = "thick"
w1_w2_distance = word_vectors.distance(w1, w2)
w1_w3_distance = word_vectors.distance(w1, w3)

print("Synonyms {}, {} have cosine distance: {}".format(w1, w2, w1_w2_distance))
print("Antonyms {}, {} have cosine distance: {}".format(w1, w3, w1_w3_distance))

    # ------------------
```

```
Synonyms thin, slim have cosine distance: 0.6212787330150604
Antonyms thin, thick have cosine distance: 0.372145414352417
```

Write your answer here.

The cosine distance between thin and slim is more than the cosine distance between thin and thick. The reason why being counter-intuitive is that the word "thin" and "thick" are often used in describing object size and "slim" may be used or described other things and not size.

Question 2.4: Analogies with Word Vectors [written] (1.5 points)

Word vectors have been shown to *sometimes* exhibit the ability to solve analogies.

As an example, for the analogy "man : king :: woman : x" (read: man is to king as woman is to x), what is x?

In the cell below, we show you how to use word vectors to find x using the `most_similar` function from the **GenSim documentation (https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.KeyedVector** The function finds words that are most similar to the words in the `positive` list and most dissimilar from the words in the `negative` list (while omitting the input words, which are often the most similar; see this paper (https://www.aclweb.org/anthology/N18-2039.pdf)). The answer to the analogy will have the highest cosine similarity (largest returned numerical value).

In [27]:

```
# Run this cell to answer the analogy -- man : king :: woman : x
pprint.pprint(word_vectors.most_similar(positive=['woman', 'king'],
                                        negative=['man']))
```

```
[('queen', 0.6713277101516724),
 ('princess', 0.5432624220848083),
 ('throne', 0.5386104583740234),
 ('monarch', 0.5347574949264526),
 ('daughter', 0.498025119304657),
 ('mother', 0.4956442713737488),
 ('elizabeth', 0.4832652509212494),
 ('kingdom', 0.47747087478637695),
 ('prince', 0.4668239951133728),
 ('wife', 0.4647327661514282)]
```

Question 2.5: Finding Analogies [code + written] (1.5 points)

Find an example of analogy that holds according to these vectors (i.e. the intended word is ranked top).

Note: You may have to try many analogies to find one that works!

In [30]:

```
    # ------------------
    # Write your implementation here.

    pprint.pprint(word_vectors.most_similar(positive=['father','woman'],
                                            negative=['man']))#man : woman :: father

    # ------------------
```

```
[('mother', 0.8266004920005798),
 ('daughter', 0.7897562980651855),
 ('husband', 0.7275589108467102),
 ('wife', 0.7243250608444214),
 ('grandmother', 0.6960293054580688),
 ('her', 0.6724176406860352),
 ('daughters', 0.6517106294631958),
 ('parents', 0.6363436579704285),
 ('son', 0.6361645460128784),
 ('sister', 0.6358030438423157)]
```

man : woman :: father : x ? father for man and mother for woman

Question 2.6: Incorrect Analogy [code + written] (1.5 points)

Find an example of analogy that does *not* hold according to these vectors.

In [33]:

```
    # ------------------
    # Write your implementation here.
    pprint.pprint(word_vectors.most_similar(positive=['apple','orange'],
                                            negative=['orange']))#orange : orange ::
    # ------------------
```

```
[('iphone', 0.5987043380737305),
 ('macintosh', 0.5836330652236938),
 ('ipod', 0.5761124491691589),
 ('microsoft', 0.5663832426071167),
 ('ipad', 0.5628098249435425),
 ('intel', 0.5457563400268555),
 ('ibm', 0.5286195278167725),
 ('google', 0.5282472372055054),
 ('imac', 0.5072519779205322),
 ('software', 0.4962984621524811)]
```

orange:orange :: apple:red

I expected orange-orange+apple = red However, the answer doesn't work perhaps because the two words "orange" are same. The results I have got are the products of apple company and any other tech organizations.

Question 2.7: Guided Analysis of Bias in Word Vectors [written] (1 point)

It's important to be cognizant of the biases (gender, race, sexual orientation etc.) implicit in our word embeddings. Bias can be dangerous because it can reinforce stereotypes through applications that employ these models.

Run the cell below, to examine (a) which terms are most similar to "woman" and "worker" and most dissimilar to "man", and (b) which terms are most similar to "man" and "worker" and most dissimilar to "woman". Point out the difference between the list of female-associated words and the list of male-associated words, and explain how it is reflecting gender bias.

In [31]:

```
# Run this cell
# Here `positive` indicates the list of words to be similar to and `negative` indica
# most dissimilar from.
pprint.pprint(word_vectors.most_similar(positive=['woman', 'worker'],
                                        negative=['man']))
pprint.pprint(word_vectors.most_similar(positive=['man', 'worker'],
                                        negative=['woman']))
```

```
[('employee', 0.5915157794952393),
 ('workers', 0.5560789108276367),
 ('nurse', 0.514857828617096),
 ('pregnant', 0.4897522032260895),
 ('mother', 0.48388367891311646),
 ('female', 0.46243947744369507),
 ('child', 0.4448588490486145),
 ('teacher', 0.44152435660362244),
 ('waitress', 0.44121503829956055),
 ('employer', 0.4378713071346283)]
[('workers', 0.5640615224838257),
 ('employee', 0.5365462303161621),
 ('laborer', 0.483084499835968),
 ('working', 0.474678635597229),
 ('factory', 0.4493158459663391),
 ('mechanic', 0.43802663683891296),
 ('work', 0.4276600480079651),
 ('unemployed', 0.42742660641670227),
 ('worked', 0.4222966134548187),
 ('job', 0.42074185609817505)]
```

Write your answer here.

(a)For the first, man: worker :: woman: x,

similar words... employee, workers, nurse,teacher, waitress, employer,

dissimilar words... child, mother, female, pregnant

(b)For the second, woman: worker :: man: x,

similar wrods... workers, employee, laborer, working, factory, machanic, work, worked, job

dissimmilar words... unemployed,

In female-associated words, there are nurse, pregnant and mother.

In male-associated words, there are factory and mechanic.

According to these two comparison, the corpus is gender biases.

Question 2.8: Independent Analysis of Bias in Word Vectors [code + written] (1 point)

Use the `most_similar` function to find another case where some bias is exhibited by the vectors. Please briefly explain the example of bias that you discover.

In [32]:

```
    # ------------------
    # Write your implementation here.

pprint.pprint(word_vectors.most_similar(positive=['woman', 'housework'],
                                        negative=['man'])) #man : housework :: womar
print("---------------------------------------------")
pprint.pprint(word_vectors.most_similar(positive=['man', 'housework'],
                                        negative=['woman'])) #woman : housework :: r


    # ------------------
```

```
[('chores', 0.4877675175666809),
 ('childbirth', 0.44542282819747925),
 ('childcare', 0.42295587062835693),
 ('bare-breasted', 0.41507065296173096),
 ('schoolwork', 0.41476333141326904),
 ('rearing', 0.411271870136261),
 ('motherhood', 0.4104798436164856),
 ('mothers', 0.40171176195144653),
 ('breastfeeding', 0.3893485963344574),
 ('mothering', 0.3790197372436523 4)]
---------------------------------------------
[('chores', 0.5121639966964722),
 ('schoolwork', 0.36822015047073364),
 ('chore', 0.3667598664760 5896),
 ('homework', 0.35656312108039856),
 ('shoveling', 0.3536524176597595),
 ('mowing', 0.3445923328399658),
 ('menial', 0.3429771065711975),
 ('doing', 0.3393510580062866),
 ('drudgery', 0.3328542113304138),
 ('ungodly', 0.33163124322891235)]
```

Write your answer here.

For the first item, man : housework :: woman : x? ,

"childcare" and "rearing" are appeared but any words realated with caring the "children" does not contain in man related housework. But "schoolwork" may be related but not totally. So, it is gender bias.

Question 2.9: Thinking About Bias [written] (2 points)

Give one explanation of how bias gets into the word vectors. Argue whether this can be lead to problems into the society. Last, how do we address this.

Write your answer here.

Give one explanation of how bias gets into the word vectors.

A bias direction measures the cosine distance from a word of interest (e.g., nurse) to a stereotyped group (e.g., female) and to the non-stereotyped group (e.g., male). If the distances are substantially different, we can assume bias in the embeddings.

Argue whether this can be lead to problems into the society.

Bias in the word embedding can hit into the society such as gender equality and ethnicity but also can lead to negative biases against middle and working-class socioeconomic status, male children, senior citizens, plain physical appearance and intellectual phenomena such as Islamic religious faith, non-religiosity and conservative political orientation.

Last, how do we address this.

Since word embeddings focuses on a number of techniques such as data augmentation , adjusted objective functions during training and post-training. For instance, in data augmentation approach, to prepare the training data, we replace gender identifying words with words of the opposite gender. These replacements are then combined with the original data and fed into the model for training. By doing this, we balance out the bias seen in the text with the opposite bias, thus making the model neutral towards both groups.

In [ ]:

```
```