CS224N Assignment 1: Exploring Word Vectors (25 Points)

Due 4:30pm, Tue Jan 17

Welcome to CS224N!

Before you start, make sure you read the README.txt in the same directory as this notebook for important setup information. A lot of code is provided in this notebook, and we highly encourage you to read and understand it as part of the learning :)

If you aren't super familiar with Python, Numpy, or Matplotlib, we recommend you check out the review session on Friday. The session will be recorded and the material will be made available on our website (http://web.stanford.edu/class/cs224n/index.html#schedule). The CS231N Python/Numpy tutorial (https://cs231n.github.io/python-numpy-tutorial/) is also a great resource.

Assignment Notes: Please make sure to save the notebook as you go along. Submission Instructions are located at the bottom of the notebook.

In [1]:
```
# All Import Statements Defined Here
# Note: Do not add to this list.
# ----------------

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

from platform import python_version
assert int(python_version().split(".")[1]) >= 5, "Please upgrade your Python 
    the README.txt file found in the same directory as this notebook. Your Py


from gensim.models import KeyedVectors
from gensim.test.utils import datapath
import pprint
import matplotlib.pyplot as plt
plt.rcParams['figure.figsize'] = [10, 5]

import nltk
nltk.download('reuters') #to specify download location, optionally add the ar
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) 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))

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 (https://web.stanford.edu/~jurafsky/slp3/6.pdf) or here (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:

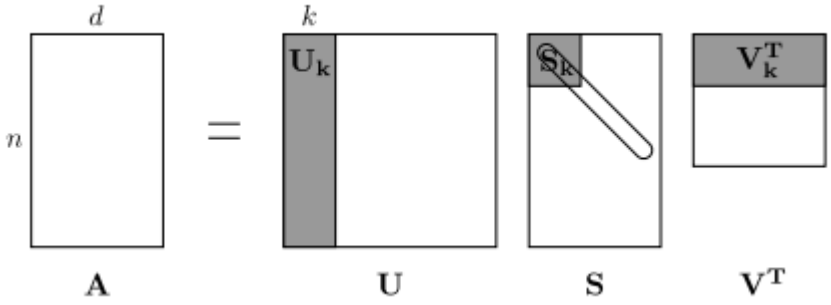
Document 1: "all that glitters is not gold"

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 this 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). If you want to learn more thoroughly about PCA or SVD, feel free to check out lectures 7 (https://web.stanford.edu/class/cs168/l/l7.pdf), 8 (http://theory.stanford.edu/~tim/s15/l/l8.pdf), and 9 (https://web.stanford.edu/class/cs168/l/l9.pdf) of CS168. These course notes provide a great high-level treatment of these general purpose algorithms. 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 "gold" (i.e. news articles

about gold, mining, 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]:
```
def read_corpus(category="gold"):
    """ 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))] + [EN
```

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

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

```
[['<START>', 'western', 'mining', 'to', 'open', 'new', 'gold', 'mine', 'in',
'australia', 'western',
  'mining', 'corp', 'holdings', 'ltd', '&', 'lt', ';', 'wmng', '.', 's',
'>', '(', 'wmc', ')',
  'said', 'it', 'will', 'establish', 'a', 'new', 'joint', 'venture', 'gold',
'mine', 'in', 'the',
  'northern', 'territory', 'at', 'a', 'cost', 'of', 'about', '21', 'mln', 'd
lrs', '.', 'the',
  'mine', ',', 'to', 'be', 'known', 'as', 'the', 'goodall', 'project', ',',
'will', 'be', 'owned',
  '60', 'pct', 'by', 'wmc', 'and', '40', 'pct', 'by', 'a', 'local', 'w',
'.', 'r', '.', 'grace',
  'and', 'co', '&', 'lt', ';', 'gra', '>', 'unit', '.', 'it', 'is', 'locate
d', '30', 'kms', 'east',
  'of', 'the', 'adelaide', 'river', 'at', 'mt', '.', 'bundey', ',', 'wmc',
'said', 'in', 'a',
  'statement', 'it', 'said', 'the', 'open', '-', 'pit', 'mine', ',', 'with',
'a', 'conventional',
  'leach', 'treatment', 'plant', ',', 'is', 'expected', 'to', 'produce', 'ab
out', '50', ',', '000',
  'ounces', 'of', 'gold', 'in', 'its', 'first', 'year', 'of', 'production',
'from', 'mid', '-',
  '1988', '.', 'annual', 'ore', 'capacity', 'will', 'be', 'about', '750',
',', '000', 'tonnes', '.',
  '<END>'],
 ['<START>', 'belgium', 'to', 'issue', 'gold', 'warrants', ',', 'sources',
'say', 'belgium',
  'plans', 'to', 'issue', 'swiss', 'franc', 'warrants', 'to', 'buy', 'gold',
',', 'with', 'credit',
  'suisse', 'as', 'lead', 'manager', ',', 'market', 'sources', 'said', '.',
'no', 'confirmation',
  'or', 'further', 'details', 'were', 'immediately', 'available', '.', '<END
>'],
 ['<START>', 'belgium', 'launches', 'bonds', 'with', 'gold', 'warrants', 'th
e', 'kingdom', 'of',
  'belgium', 'is', 'launching', '100', 'mln', 'swiss', 'francs', 'of', 'seve
n', 'year', 'notes',
  'with', 'warrants', 'attached', 'to', 'buy', 'gold', ',', 'lead', 'manange
r', 'credit', 'suisse',
  'said', '.', 'the', 'notes', 'themselves', 'have', 'a', '3', '-', '3',
'/', '8', 'pct', 'coupon',
  'and', 'are', 'priced', 'at', 'par', '.', 'payment', 'is', 'due', 'april',
'30', ',', '1987',
  'and', 'final', 'maturity', 'april', '30', ',', '1994', '.', 'each', '50',
',', '000', 'franc',
  'note', 'carries', '15', 'warrants', '.', 'two', 'warrants', 'are', 'requi
red', 'to', 'allow',
  'the', 'holder', 'to', 'buy', '100', 'grammes', 'of', 'gold', 'at', 'a',
'price', 'of', '2', ',',
  '450', 'francs', ',', 'during', 'the', 'entire', 'life', 'of', 'the', 'bon
d', '.', 'the',
  'latest', 'gold', 'price', 'in', 'zurich', 'was', '2', ',', '045', '/',
'2', ',', '070', 'francs',
  'per', '100', 'grammes', '.', '<END>']]
```

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) 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).

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) to remove duplicate words.

In [4]:
```
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 acr
            n_corpus_words (integer): number of distinct words across the cor
    """
    corpus_words = []
    n_corpus_words = -1

    ### SOLUTION BEGIN
    corpus_words = [*map(set, corpus)]
    corpus_words = sorted([*set().union(*corpus_words)])
    n_corpus_words = len(corpus_words)
    ### SOLUTION END

    return corpus_words, n_corpus_words
```

In [5]:
```
# ---------------------
# 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_T
test_corpus_words, num_corpus_words = distinct_words(test_corpus)

# Correct answers
ans_test_corpus_words = sorted([START_TOKEN, "All", "ends", "that", "gold", "
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 distin

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

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

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

Question 1.2: Implement `compute_co_occurrence_matrix` [code] (3 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. If you're not familiar with NumPy, there's a NumPy tutorial in the second half of this cs231n Python NumPy tutorial (http://cs231n.github.io/python-numpy-tutorial/).

In [1]:
```
def compute_co_occurrence_matrix(corpus, window_size=4):
    """ Compute co-occurrence matrix for the given corpus and window_size (de

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

              For example, if we take the document "<START> All that glitters
              "All" will co-occur with "<START>", "that", "glitters", "is", a

        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 t
                Co-occurence matrix of word counts.
                The ordering of the words in the rows/columns should be the s
            word2ind (dict): dictionary that maps word to index (i.e. row/col
    """
    words, n_words = distinct_words(corpus)
    M = None
    word2ind = {}

    ### SOLUTION BEGIN

    # map distinct words to an ordered index
    for i in range(n_words):
        word2ind[words[i]] = i

    M = np.zeros((n_words, n_words))
    for sentence in corpus:
        for w in range(len(sentence)):
            current_word = sentence[w]
            neighbors = sentence[max(0, w-window_size):min(len(sentence)
                                                    , w+window_size+1)]

            # add neighbors to matrix M
            for n in neighbors:
                M[word2ind[current_word]][word2ind[n]] += 1

            M[word2ind[current_word]][word2ind[current_word]] -= 1
    ### SOLUTION END

    return M, word2ind
```

In [7]:
```
# ---------------------
# 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_T
M_test, word2ind_test = compute_co_occurrence_matrix(test_corpus, window_size

# 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", "
word2ind_ans = dict(zip(ans_test_corpus_words, range(len(ans_test_corpus_word

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

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

# 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 ({}, {})=({}, {}) 

# 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 [8]:
```
def reduce_to_k_dim(M, k=2):
    """ Reduce a co-occurence count matrix of dimensionality (num_corpus_words
        to a matrix of dimensionality (num_corpus_words, k) using the followi
            - http://scikit-learn.org/stable/modules/generated/sklearn.decomp

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

    ### SOLUTION BEGIN
    truncated_svd = TruncatedSVD(n_components=k)

    M_reduced = truncated_svd.fit_transform(M)
    ### SOLUTION END

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

In [9]:
```
# ---------------------
# 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_T
M_test, word2ind_test = compute_co_occurrence_matrix(test_corpus, window_size
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 {
assert (M_test_reduced.shape[1] == 2), "M_reduced has {} columns; should have

# 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`).

For this example, you may find it useful to adapt this code (http://web.archive.org/web/20190924160434/https://www.pythonmembers.club/2018/05/08/matp scatter-plot-annotate-set-text-at-label-each-point/). In the future, a good way to make a plot is to look at the Matplotlib gallery (https://matplotlib.org/gallery/index.html), find a plot that looks somewhat like what you want, and adapt the code they give.

In [10]:
```
def plot_embeddings(M_reduced, word2ind, words):
    """ Plot in a scatterplot the embeddings of the words specified in the lis
        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 co
            word2ind (dict): dictionary that maps word to indices for matrix M
            words (list of strings): words whose embeddings we want to visuali
    """

    ### SOLUTION BEGIN
    for word in words:
        x = M_reduced[word2ind[word]][0]
        y = M_reduced[word2ind[word]][1]
        plt.scatter(x, y, marker='x', color='red')
        plt.text(x+0.0001, y+0.0001, word, fontsize=20)
    plt.show()
    ### SOLUTION END
```

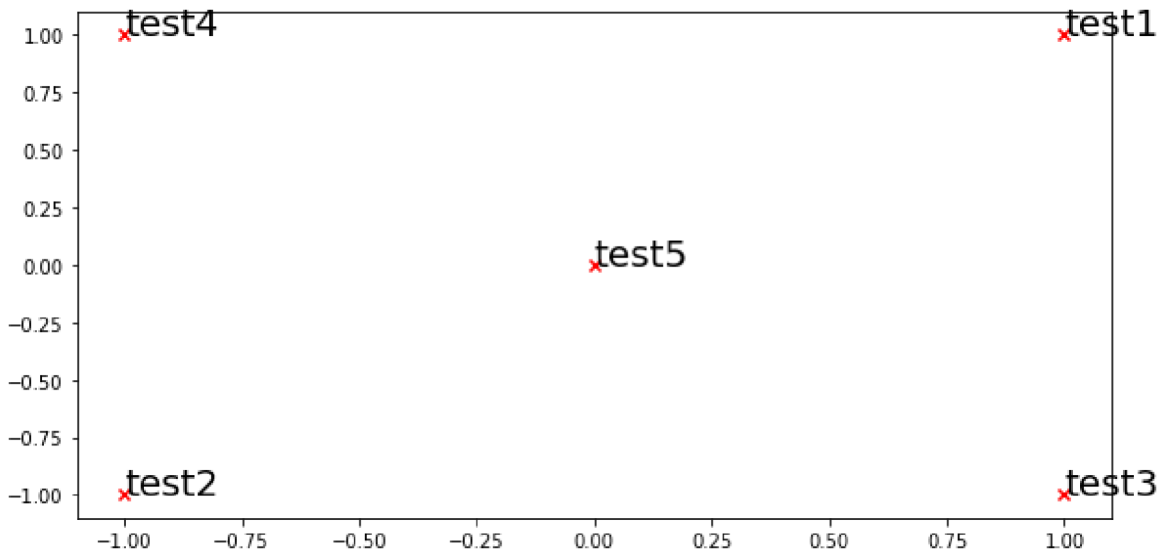
In [11]:
```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness.
# The plot produced should look like the "test solution plot" depicted below.
# ---------------------

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'
words = ['test1', 'test2', 'test3', 'test4', 'test5']
plot_embeddings(M_reduced_plot_test, word2ind_plot_test, words)

print ("-" * 80)
```

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




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

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 "gold" 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).

Run the below cell to produce the plot. It'll probably take a few seconds to run.

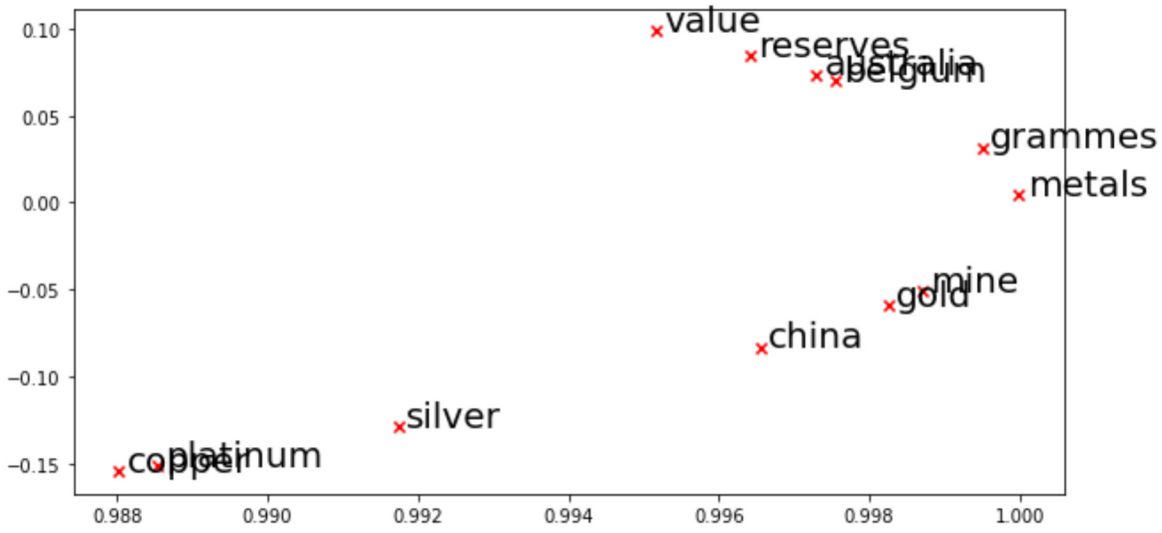
In [12]:
```
# ----------------------------
# Run This Cell to Produce Your Plot
# ----------------------------
reuters_corpus = read_corpus()
M_co_occurrence, word2ind_co_occurrence = compute_co_occurrence_matrix(reuter
M_reduced_co_occurrence = reduce_to_k_dim(M_co_occurrence, k=2)

# 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] # broadcast

words = ['value', 'gold', 'platinum', 'reserves', 'silver', 'metals', 'copper
         'belgium', 'australia', 'china', 'grammes', "mine"]

plot_embeddings(M_normalized, word2ind_co_occurrence, words)
```

```
Running Truncated SVD over 2830 words...
Done.
```



Verify that your figure matches "question_1.5.png" in the assignment zip. If not, use that figure to answer the next two questions.

a. Find at least two groups of words that cluster together in 2-dimensional embedding space. Give an explanation for each cluster you observe.

SOLUTION BEGIN

In this example, the co-occurrence matrix counts the number of times each word appears inside a fixed window of size 4 around a word of interest. The method cluster groups of words that tend to occur together in location or written very frequently along with a shared subset of words:

- Cluster 1: Gold and Mine
- Cluster 2: platinum and copper

The cluster 1 ‘gold -mine’ is a common phrase and have more than 46 common occurances inside the fixed window of size 4. In the corpus, they are written in a similar context and appear together with 240 shared words.

The cluster 2 'platinum and copper' are both metal. The words copper and platinum doesn't have common occurances inside the fixed window of size 4, but they are written very frequently along with common words in similar context.

SOLUTION END

b. What doesn't cluster together that you might think should have? Describe at least two examples.

SOLUTION BEGIN

However, the method fails to capture relationship between words, that doesn't appears in close proximity. Like for example, China is too far from the cluster Australia and Belgium. I also think that metals and copper should be cluster together since they are relatively close together in concept. The cause of this could be that these groups of words do not have common occurrences in the corpus and appear in different contexts, which makes it difficult for the model to calculate better groups.

SOLUTION END

Part 2: Prediction-Based Word Vectors (15 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). Here, we shall explore the embeddings produced by GloVe. Please revisit the class notes and lecture slides for more details on the word2vec and GloVe algorithms. If you're feeling adventurous, challenge yourself and try reading GloVe's original paper (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 [ ]:
```
def load_embedding_model():
    """ Load GloVe Vectors
        Return:
            wv_from_bin: All 400000 embeddings, each lengh 200
    """
    import gensim.downloader as api
    wv_from_bin = api.load("glove-wiki-gigaword-200")
    #print("Loaded vocab size %i" % len(list(wv_from_bin.index_to_key)))
    print("Loaded vocab size %i" % len(list(wv_from_bin.vocab.keys())))
    return wv_from_bin
```

In [ ]:
```
# ----------------------------------
# Run Cell to Load Word Vectors
# Note: This will take a couple minutes
# ----------------------------------
wv_from_bin = load_embedding_model()
```

```
[================================================] 100.0% 252.1/252.1MB do
wnloaded
Loaded vocab size 400000
```

Note: If you are receiving a "reset by peer" error, rerun the cell to restart the download. If you run into an "attribute" error, you may need to update to the most recent version of gensim and numpy. You can upgrade them inline by uncommenting and running the below cell:

In [ ]:
```
#!pip install gensim --upgrade
#!pip install numpy --upgrade
```

Reducing dimensionality of Word Embeddings

Let's directly compare the GloVe embeddings to those of the co-occurrence matrix. In order to avoid running out of memory, we will work with a sample of 10000 GloVe vectors instead. Run the following cells to:

1. Put 10000 Glove vectors into a matrix M
2. Run `reduce_to_k_dim` (your Truncated SVD function) to reduce the vectors from 200-dimensional to 2-dimensional.

In [ ]:
```
def get_matrix_of_vectors(wv_from_bin, required_words):
    """ Put the GloVe vectors into a matrix M.
        Param:
            wv_from_bin: KeyedVectors object; the 400000 GloVe vectors loaded
        Return:
            M: numpy matrix shape (num words, 200) containing the vectors
            word2ind: dictionary mapping each word to its row number in M
    """
    import random
    #words = list(wv_from_bin.index_to_key)
    words = list(wv_from_bin.vocab.keys())
    print("Shuffling words ...")
    random.seed(225)
    random.shuffle(words)
    words = words[:10000]
    print("Putting %i words into word2ind and matrix M..." % len(words))
    word2ind = {}
    M = []
    curInd = 0
    for w in words:
        try:
            M.append(wv_from_bin.get_vector(w))
            word2ind[w] = curInd
            curInd += 1
        except KeyError:
            continue
    for w in required_words:
        if w in words:
            continue
        try:
            M.append(wv_from_bin.get_vector(w))
            word2ind[w] = curInd
            curInd += 1
        except KeyError:
            continue
    M = np.stack(M)
    print("Done.")
    return M, word2ind
```

In [ ]:
```
# ----------------------------------------------------------------
# Run Cell to Reduce 200-Dimensional Word Embeddings to k Dimensions
# Note: This should be quick to run
# ----------------------------------------------------------------
M, word2ind = get_matrix_of_vectors(wv_from_bin, words)
M_reduced = reduce_to_k_dim(M, k=2)

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

```
Shuffling words ...
Putting 10000 words into word2ind and matrix M...
Done.
Running Truncated SVD over 10012 words...
Done.
```

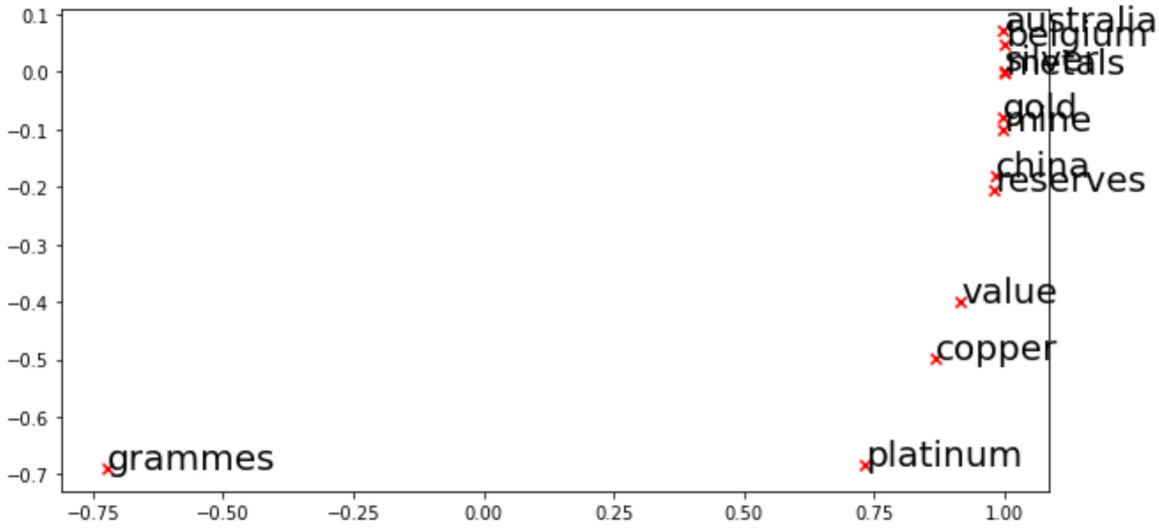
Note: If you are receiving out of memory issues on your local machine, try closing other applications to free more memory on your device. You may want to try restarting your machine so that you can free up extra memory. Then immediately run the jupyter notebook and see if you can load the word vectors properly. If you still have problems with loading the embeddings onto your local machine after this, please go to office hours or contact course staff.

Question 2.1: GloVe Plot Analysis [written] (3 points)

Run the cell below to plot the 2D GloVe embeddings for `['value', 'gold', 'platinum', 'reserves', 'silver', 'metals', 'copper', 'belgium', 'australia', 'china', 'grammes', "mine"]` .

In [ ]:
```
words = ['value', 'gold', 'platinum', 'reserves', 'silver', 'metals', 'copper',
         'belgium', 'australia', 'china', 'grammes', "mine"]

plot_embeddings(M_reduced_normalized, word2ind, words)
```



a. What is one way the plot is different from the one generated earlier from the co-occurrence matrix? What is one way it's similar?

SOLUTION BEGIN

The Glove plot is more clustered and the plot from the co-occurence matrix is more disperse. The GloVe plot cluster together the words Australia, Belgium, silver and metals. Surprisingly, copper and platinum are not clustered together as in the co-occurrence matrix plot. Some similarities between both plots are observed, like for example, gold and mine are clustered together, the same for the words Australia and Belgium.

SOLUTION END

b. What is a possible cause for the difference?

SOLUTION BEGIN

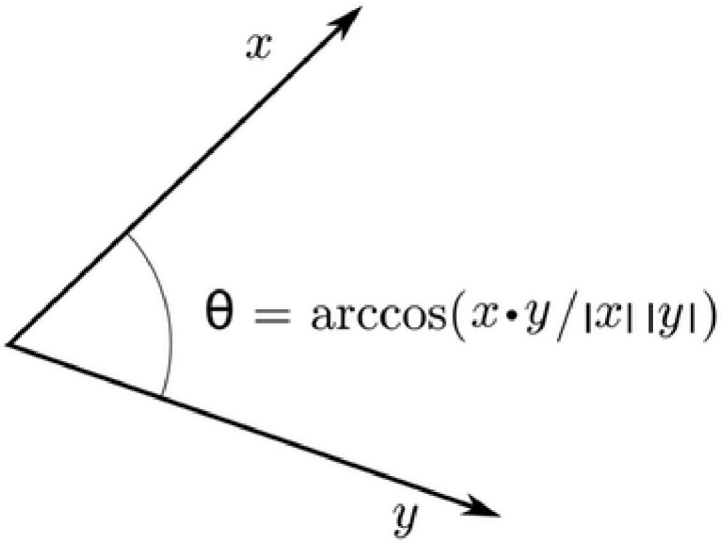
The possible cause for the difference is that the co-occurrence matrix only considered words that appears in close proximity in the corpus whereas GloVe captures the ratios of co-occurrence probabilities, which mitigate the effect large counts words, scaling the matrix of co-occurrence.

SOLUTION END

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) and L2 (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) s between two vectors p and q is defined as:

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

Question 2.2: 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) 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 `wv_from_bin.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.keyedvecto**

In [ ]:
```
### SOLUTION BEGIN
similars = wv_from_bin.most_similar("serve")
print(similars)
### SOLUTION END
```

```
[('serving', 0.7944269180297852), ('served', 0.7393940687179565), ('serves',
0.7277334928512573), ('make', 0.5822009444236755), ('to', 0.581701159477233
9), ('break', 0.5659143924713135), ('should', 0.5649185180664062), ('well',
0.5588014125823975), ('provide', 0.5582747459411621), ('give', 0.55733692646
02661)]
```

SOLUTION BEGIN

- Please state the word you discover and the multiple meanings that occur in the top 10.

The polysemous word I discover is 'serve'. The multiple meanings that occur in the top 10 includes make, provide and give.

- 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)?

A possible explanation could be that only one of the word meanings has been used most frequently in the corpus and the other meanings are very rare in the corpus.

SOLUTION END

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 `wv_from_bin.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.keyedvecto](https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvecto) for further assistance.

In [ ]:
```
### SOLUTION BEGIN

w1 = "dry"
w2 = "arid"
w3 = "wet"
w1_w2_dist = wv_from_bin.distance(w1, w2)
w1_w3_dist = wv_from_bin.distance(w1, w3)

print("Synonyms {}, {} have cosine distance: {}".format(w1, w2, w1_w2_dist))
print("Antonyms {}, {} have cosine distance: {}".format(w1, w3, w1_w3_dist))

### SOLUTION END
```

```
Synonyms dry, arid have cosine distance: 0.47727078199386597
Antonyms dry, wet have cosine distance: 0.26958924531936646
```

SOLUTION BEGIN

Dry and arid are synonyms and wet is an antonym. However, the cosine distance using the Reuters corpus indicates that dry is further away from arid than from wet. One reason could be that dry and wet appear frequently together in the same sentence, they are usually used in similar context and are exchangeable in a much larger variety of context. On the other hand, dry and arid are synonyms only in a specific context, like for example talking about landscapes.

SOLUTION END

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 : grandfather :: woman : x" (read: man is to grandfather 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.keyedvecto** 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 [ ]:
```
# Run this cell to answer the analogy -- man : grandfather :: woman : x
pprint.pprint(wv_from_bin.most_similar(positive=['woman', 'grandfather'], neg
```

```
[('grandmother', 0.7608444690704346),
 ('granddaughter', 0.7200808525085449),
 ('daughter', 0.7168302536010742),
 ('mother', 0.7151535749435425),
 ('niece', 0.7005682587623596),
 ('father', 0.6659887433052063),
 ('aunt', 0.6623408794403076),
 ('grandson', 0.6618767976760864),
 ('grandparents', 0.6446609497070312),
 ('wife', 0.6445354223251343)]
```

Let m, g, w, and x denote the word vectors for `man`, `grandfather`, `woman`, and the answer, respectively. Using **only** vectors m, g, w, and the vector arithmetic operators $+$ and $-$ in your answer, to what expression are we maximizing x's cosine similarity?

Hint: Recall that word vectors are simply multi-dimensional vectors that represent a word. It might help to draw out a 2D example using arbitrary locations of each vector. Where would `man` and `woman` lie in the coordinate plane relative to `grandfather` and the answer?

SOLUTION BEGIN

The expression is:

x = g-m+w

SOLUTION END

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

a. For the previous example, it's clear that "grandmother" completes the analogy. But give an intuitive explanation as to why the `most_similar` function gives us words like "granddaughter", "daughter", or "mother?

SOLUTION BEGIN

The most_similar function gives us words like "granddaughter", "daughter", or "mother because they are very similar to the positive words and least similar to the words in the negative list. Then we will have a high cosine similarity for these words.

SOLUTION END

b. Find an example of analogy that holds according to these vectors (i.e. the intended word is ranked top). In your solution please state the full analogy in the form x:y :: a:b. If you believe the analogy is complicated, explain why the analogy holds in one or two sentences.

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

In [ ]:
```
### SOLUTION BEGIN

x, y, a, b = 'prince', 'king', 'princess', 'queen'
assert wv_from_bin.most_similar(positive=[a, y], negative=[x])[0][0] == b

### SOLUTION END
```

SOLUTION BEGIN

'prince':'king' = 'princess':'queen'

SOLUTION END

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

a. Below, we expect to see the intended analogy "hand : glove :: foot : **sock**", but we see an unexpected result instead. Give a potential reason as to why this particular analogy turned out the way it did?

In [ ]:
```
pprint.pprint(wv_from_bin.most_similar(positive=['foot', 'glove'], negative=[
```

```
[('45,000-square', 0.4922032356262207),
 ('15,000-square', 0.4649604558944702),
 ('10,000-square', 0.45447561144828796),
 ('6,000-square', 0.44975775480270386),
 ('3,500-square', 0.4441334009170532),
 ('700-square', 0.44257503747940063),
 ('50,000-square', 0.4356396794319153),
 ('3,000-square', 0.43486517667770386),
 ('30,000-square', 0.4330596923828125),
 ('footed', 0.43236875534057617)]
```

SOLUTION BEGIN

It seems that the model may not know that 'foot' can have different meanings and it is not only an unit of measurement. Another reason could be that glove and sock may not have similar word vectors.

SOLUTION END

b. Find another example of analogy that does *not* hold according to these vectors. In your solution, state the intended analogy in the form x:y :: a:b, and state the **incorrect** value of b according to the word vectors (in the previous example, this would be **'45,000-square'**).

In [ ]:
```
### SOLUTION BEGIN

x, y, a, b = 'cat', 'tiger', 'dog', 'wolf'
pprint.pprint(wv_from_bin.most_similar(positive=[a, y], negative=[x]))
### SOLUTION END
```

```
[('woods', 0.4931395351886749),
 ('tigers', 0.4840834140777588),
 ('bear', 0.43776991963386536),
 ('lion', 0.4275178611278534),
 ('hunting', 0.4224510192871094),
 ('mickelson', 0.41707539558410645),
 ('cub', 0.41608965396881104),
 ('hunter', 0.4141656458377838),
 ('fighting', 0.4121222198009491),
 ('soldier', 0.4110243320465088)]
```

SOLUTION BEGIN

I expect to see the intended analogy "cat : tiger :: dog : wolf", but we see an unexpected result instead: woods. It looks like that dog and wolf may not similar in the word vector.

SOLUTION END

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 "profession" and most dissimilar to "man", and (b) which terms are most similar to "man" and "profession" 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 [ ]:
```
# Run this cell
# Here `positive` indicates the list of words to be similar to and `negative`
# most dissimilar from.

pprint.pprint(wv_from_bin.most_similar(positive=['man', 'profession'], negati
print()
pprint.pprint(wv_from_bin.most_similar(positive=['woman', 'profession'], nega
```

```
[('reputation', 0.5250176191329956),
 ('professions', 0.5178038477897644),
 ('skill', 0.49046966433525085),
 ('skills', 0.49005505442619324),
 ('ethic', 0.4897659420967102),
 ('business', 0.4875851273536682),
 ('respected', 0.4859202802181244),
 ('practice', 0.4821045994758606),
 ('regarded', 0.47785723209381104),
 ('life', 0.4760662317276001)]

[('professions', 0.5957458019256592),
 ('practitioner', 0.498841255903244),
 ('teaching', 0.48292142152786255),
 ('nursing', 0.4821180999279022),
 ('vocation', 0.4788966178894043),
 ('teacher', 0.47160351276397705),
 ('practicing', 0.46937811374664307),
 ('educator', 0.46524325013160706),
 ('physicians', 0.4628995358943939),
 ('professionals', 0.4601394236087799)]
```

SOLUTION BEGIN

"woman" is to "profession" as "man" is to: reputation, professions, skill, skills, ethic, business, respected, practice, regarded, life.

"man" is to "profession" as "woman" is to: professions, practitioner, teaching, nursing, vocation, teacher, practicing, educator, physicians, professionals.

The list of words associated with women are words related primarily to careers that help people, such as teaching, nursing, educator. The words related to man, is more about ethic and business. These results came from both the corpus and the language itself, which contains gender stereotypes, making the model repeat those stereotypes.

SOLUTION END

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

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

In [ ]:
```
### SOLUTION BEGIN

A = 'mother'
B = 'father'
word = 'doctor'
pprint.pprint(wv_from_bin.most_similar(positive=[A, word], negative=[B]))
print()
pprint.pprint(wv_from_bin.most_similar(positive=[B, word], negative=[A]))

### SOLUTION END
```

```
[('nurse', 0.7208659648895264),
 ('doctors', 0.6413154602050781),
 ('patient', 0.6289440393447876),
 ('woman', 0.6113752126693726),
 ('hospital', 0.6000144481658936),
 ('pregnant', 0.5975667238235474),
 ('nurses', 0.5725877285003662),
 ('physician', 0.5669364929199219),
 ('medical', 0.5617853403091431),
 ('patients', 0.5472391247749329)]

[('physician', 0.6719361543655396),
 ('surgeon', 0.6208167672157288),
 ('dr.', 0.5724584460258484),
 ('brother', 0.5710499882698059),
 ('son', 0.5303334593772888),
 ('he', 0.5294877290725708),
 ('medical', 0.5288361310958862),
 ('uncle', 0.5231920480728149),
 ('himself', 0.5133481621742249),
 ('pharmacist', 0.5111744999885559)]
```

SOLUTION BEGIN

The results shows an inherent bias that word mother is associated with terms like nurse, pregnant, and patient, and the word father is associated to physician and surgeon. The result exhibit female/male gender stereotypes.

SOLUTION END

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

a. Give one explanation of how bias gets into the word vectors. Briefly describe a real-world example that demonstrates this source of bias.

SOLUTION BEGIN

The biases are inherently present in the training data. Model is a statistical extraction of these latent variables and the vector space is a tool to expose the biases and make them clearly visible. Biases are not a property of the algorithm but of the world respectively training data collected from it. A real world example was discussed by Bolukbasi et al. (2016). The authors found that trained word embeddings in Google News articles exhibit female/male gender stereotypes. They found analogies such as "man is to programmer as woman is to homemaker", which exhibit implicit sexism in the text.

SOLUTION END

b. What is one method you can use to mitigate bias exhibited by word vectors? Briefly describe a real-world example that demonstrates this method.

SOLUTION BEGIN

One method we can use to mitigate bias exhibited by word vectors is to use "hard debiasing" technique. A very famous real-world example is the work by Bolukbasi et al. (2016). The authors proposed a method for debiasing word embeddings which involves modifying an embedding to remove gender stereotypes, while preserving the relationships between the words.

SOLUTION END

Submission Instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of all cells).
3. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
4. Once you've rerun everything, select File -> Download as -> PDF via LaTeX (If you have trouble using "PDF via LaTex", you can also save the webpage as pdf. Make sure all your solutions especially the coding parts are displayed in the pdf, it's okay if the provided codes get cut off because lines are not wrapped in code cells).
5. Look at the PDF file and make sure all your solutions are there, displayed correctly. The PDF is the only thing your graders will see!
6. Submit your PDF on Gradescope.