CS224N Assignment 1: Exploring Word Vectors (25 Points)

Due 4:30pm, Tue April 9th 2024

Welcome to CS224N!

Before you start, make sure you **read the README.md** in the same directory as this notebook for important setup information. You need to install some Python libraries before you can successfully do this assignment. 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. The CS231N 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.

```python
In [1]:  import sys
         assert sys.version_info[0] == 3
         assert sys.version_info[1] >= 8

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

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

         from datasets import load_dataset
         imdb_dataset = load_dataset("stanfordnlp/imdb")

         import re
         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>'
         NUM_SAMPLES = 150

         np.random.seed(0)
```

```
random.seed(0)
# ----------------
```

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 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)

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 or here).

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 $w_{i-n} \ldots w_{i-1}$ and $w_{i+1} \ldots w_{i+n}$. 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

In NLP, we commonly use `<START>` and `<END>` tokens to mark the beginning and end of sentences, paragraphs, or documents. These tokens are included in co-occurrence counts, encapsulating each document, for example: " `<START>` All that glitters is not gold `<END>` ".

The matrix rows (or columns) provide word vectors based on word-word co-occurrence, but they can be large. To reduce dimensionality, we employ Singular Value Decomposition (SVD), akin to PCA, selecting the top k principal components. The SVD process decomposes the co-occurrence matrix A into singular values in the diagonal S matrix and new, shorter word vectors in U_k.

This dimensionality reduction maintains semantic relationships; for instance, *doctor* and *hospital* will be closer than *doctor* and *dog*.

For those unfamiliar with eigenvalues and SVD, a beginner-friendly introduction to SVD is available here. Additional resources for in-depth understanding include lectures 7, 8, and 9 of CS168, providing high-level treatment of these algorithms. For practical implementation, utilizing pre-programmed functions from Python packages like numpy, scipy, or sklearn is recommended. While applying full SVD to large corpora can be memory-intensive, scalable techniques such as Truncated SVD exist for extracting the top k vector components efficiently.

Plotting Co-Occurrence Word Embeddings

Here, we will be using the Large Movie Review Dataset. This is a dataset for binary sentiment classification containing substantially more data than previous benchmark datasets. We provide a set of 25,000 highly polar movie reviews for training, and 25,000 for testing. There is additional unlabeled data for use as well. We provide a `read_corpus` function below that pulls out the text of a movie review from the dataset. 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.

```python
In [2]: def read_corpus():
            """ Read files from the Large Movie Review Dataset.
                Params:
                    category (string): category name
                Return:
                    list of lists, with words from each of the processed files
            """
            files = imdb_dataset["train"]["text"][:NUM_SAMPLES]
            return [[START_TOKEN] + [re.sub(r'[^\w]', '', w.lower()) for w in f.split(" ")]
```

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

```python
In [3]: imdb_corpus = read_corpus()
        pprint.pprint(imdb_corpus[:3], compact=True, width=100)
        print("corpus size: ", len(imdb_corpus[0]))
```

```
[['<START>', 'i', 'rented', 'i', 'am', 'curiousyellow', 'from', 'my', 'video', 'stor
e', 'because',
  'of', 'all', 'the', 'controversy', 'that', 'surrounded', 'it', 'when', 'it', 'wa
s', 'first',
  'released', 'in', '1967', 'i', 'also', 'heard', 'that', 'at', 'first', 'it', 'wa
s', 'seized',
  'by', 'us', 'customs', 'if', 'it', 'ever', 'tried', 'to', 'enter', 'this', 'countr
y', 'therefore',
  'being', 'a', 'fan', 'of', 'films', 'considered', 'controversial', 'i', 'really',
'had', 'to',
  'see', 'this', 'for', 'myselfbr', 'br', 'the', 'plot', 'is', 'centered', 'around',
'a', 'young',
  'swedish', 'drama', 'student', 'named', 'lena', 'who', 'wants', 'to', 'learn', 'ev
erything',
  'she', 'can', 'about', 'life', 'in', 'particular', 'she', 'wants', 'to', 'focus',
'her',
  'attentions', 'to', 'making', 'some', 'sort', 'of', 'documentary', 'on', 'what',
'the', 'average',
  'swede', 'thought', 'about', 'certain', 'political', 'issues', 'such', 'as', 'th
e', 'vietnam',
  'war', 'and', 'race', 'issues', 'in', 'the', 'united', 'states', 'in', 'between',
'asking',
  'politicians', 'and', 'ordinary', 'denizens', 'of', 'stockholm', 'about', 'their',
'opinions',
  'on', 'politics', 'she', 'has', 'sex', 'with', 'her', 'drama', 'teacher', 'classma
tes', 'and',
  'married', 'menbr', 'br', 'what', 'kills', 'me', 'about', 'i', 'am', 'curiousyello
w', 'is',
  'that', '40', 'years', 'ago', 'this', 'was', 'considered', 'pornographic', 'reall
y', 'the', 'sex',
  'and', 'nudity', 'scenes', 'are', 'few', 'and', 'far', 'between', 'even', 'then',
'its', 'not',
  'shot', 'like', 'some', 'cheaply', 'made', 'porno', 'while', 'my', 'countrymen',
'mind', 'find',
  'it', 'shocking', 'in', 'reality', 'sex', 'and', 'nudity', 'are', 'a', 'major', 's
taple', 'in',
  'swedish', 'cinema', 'even', 'ingmar', 'bergman', 'arguably', 'their', 'answer',
'to', 'good',
  'old', 'boy', 'john', 'ford', 'had', 'sex', 'scenes', 'in', 'his', 'filmsbr', 'b
r', 'i', 'do',
  'commend', 'the', 'filmmakers', 'for', 'the', 'fact', 'that', 'any', 'sex', 'show
n', 'in', 'the',
  'film', 'is', 'shown', 'for', 'artistic', 'purposes', 'rather', 'than', 'just', 't
o', 'shock',
  'people', 'and', 'make', 'money', 'to', 'be', 'shown', 'in', 'pornographic', 'thea
ters', 'in',
  'america', 'i', 'am', 'curiousyellow', 'is', 'a', 'good', 'film', 'for', 'anyone',
'wanting',
  'to', 'study', 'the', 'meat', 'and', 'potatoes', 'no', 'pun', 'intended', 'of', 's
wedish',
  'cinema', 'but', 'really', 'this', 'film', 'doesnt', 'have', 'much', 'of', 'a', 'p
lot', '<END>'],
 ['<START>', 'i', 'am', 'curious', 'yellow', 'is', 'a', 'risible', 'and', 'pretentio
us', 'steaming',
  'pile', 'it', 'doesnt', 'matter', 'what', 'ones', 'political', 'views', 'are', 'be
cause', 'this',
```

```
  'film', 'can', 'hardly', 'be', 'taken', 'seriously', 'on', 'any', 'level', 'as',
'for', 'the',
  'claim', 'that', 'frontal', 'male', 'nudity', 'is', 'an', 'automatic', 'nc17', 'th
at', 'isnt',
  'true', 'ive', 'seen', 'rrated', 'films', 'with', 'male', 'nudity', 'granted', 'th
ey', 'only',
  'offer', 'some', 'fleeting', 'views', 'but', 'where', 'are', 'the', 'rrated', 'fil
ms', 'with',
  'gaping', 'vulvas', 'and', 'flapping', 'labia', 'nowhere', 'because', 'they', 'don
t', 'exist',
  'the', 'same', 'goes', 'for', 'those', 'crappy', 'cable', 'shows', 'schlongs', 'sw
inging', 'in',
  'the', 'breeze', 'but', 'not', 'a', 'clitoris', 'in', 'sight', 'and', 'those', 'pr
etentious',
  'indie', 'movies', 'like', 'the', 'brown', 'bunny', 'in', 'which', 'were', 'treate
d', 'to', 'the',
  'site', 'of', 'vincent', 'gallos', 'throbbing', 'johnson', 'but', 'not', 'a', 'tra
ce', 'of',
  'pink', 'visible', 'on', 'chloe', 'sevigny', 'before', 'crying', 'or', 'implying',
  'doublestandard', 'in', 'matters', 'of', 'nudity', 'the', 'mentally', 'obtuse', 's
hould', 'take',
  'into', 'account', 'one', 'unavoidably', 'obvious', 'anatomical', 'difference', 'b
etween', 'men',
  'and', 'women', 'there', 'are', 'no', 'genitals', 'on', 'display', 'when', 'actres
ses', 'appears',
  'nude', 'and', 'the', 'same', 'cannot', 'be', 'said', 'for', 'a', 'man', 'in', 'fa
ct', 'you',
  'generally', 'wont', 'see', 'female', 'genitals', 'in', 'an', 'american', 'film',
'in',
  'anything', 'short', 'of', 'porn', 'or', 'explicit', 'erotica', 'this', 'alleged',
  'doublestandard', 'is', 'less', 'a', 'double', 'standard', 'than', 'an', 'admitted
ly',
  'depressing', 'ability', 'to', 'come', 'to', 'terms', 'culturally', 'with', 'the',
'insides',
  'of', 'womens', 'bodies', '<END>'],
 ['<START>', 'if', 'only', 'to', 'avoid', 'making', 'this', 'type', 'of', 'film', 'i
n', 'the',
  'future', 'this', 'film', 'is', 'interesting', 'as', 'an', 'experiment', 'but', 't
ells', 'no',
  'cogent', 'storybr', 'br', 'one', 'might', 'feel', 'virtuous', 'for', 'sitting',
'thru', 'it',
  'because', 'it', 'touches', 'on', 'so', 'many', 'important', 'issues', 'but', 'i
t', 'does', 'so',
  'without', 'any', 'discernable', 'motive', 'the', 'viewer', 'comes', 'away', 'wit
h', 'no', 'new',
  'perspectives', 'unless', 'one', 'comes', 'up', 'with', 'one', 'while', 'ones', 'm
ind', 'wanders',
  'as', 'it', 'will', 'invariably', 'do', 'during', 'this', 'pointless', 'filmbr',
'br', 'one',
  'might', 'better', 'spend', 'ones', 'time', 'staring', 'out', 'a', 'window', 'at',
'a', 'tree',
  'growingbr', 'br', '', '<END>']]
corpus size:  290
```

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 use `for` loops to process the input `corpus` (a list of list of strings), but try using Python list comprehensions (which are generally faster). In particular, this may be useful to flatten a list of lists. If you're not familiar with Python list comprehensions in general, here's more information.

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 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 across th
                     n_corpus_words (integer): number of distinct words across the corpus
             """
             seen = set()

             # ------------------
             # Write your implementation here.
             corpus_words = list(set([y for x in corpus for y in x if y not in seen and not
             corpus_words.sort()
             n_corpus_words = len(seen)
             # ------------------

             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_TOKEN).
         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 wor

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

         # 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.

In [6]:
```python
def compute_co_occurrence_matrix(corpus, window_size=4):
    """ Compute co-occurrence matrix for the given corpus and window_size (default

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

              For example, if we take the document "<START> All that glitters is no
              "All" will co-occur with "<START>", "that", "glitters", "is", and "no

        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 cor
                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 nu
    """
    words, n_words = distinct_words(corpus)
    M = np.zeros((n_words,n_words))
    word2ind = {word:i for i,word in enumerate(words)}

    # ------------------
    # Write your implementation here.
    for sentence in corpus:
        for word_index in range(len(sentence)):
            center_word_ind = word2ind[sentence[word_index]]
            start_index = max (0 , word_index - window_size )
            end_index = min (len(sentence), word_index + window_size)
            for index in range(start_index, end_index):
                if index != word_index:
                    context_word_index = word2ind[sentence[index]]
                    M[center_word_ind][context_word_index] += 1
                    M[context_word_index][center_word_ind] += 1
    # ------------------

    return M, word2ind
```

```python
# --------------------
# 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).
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: {}\n

# 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 mat

# 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.

```python
In [8]: 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.decompositio

        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 o
                In terms of the SVD from math class, this actually returns U *
    """
    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=2, n_iter=n_iters, random_state=42)
    M_reduced = svd.fit_transform(M)
    # ------------------

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

```python
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_TOKEN).
    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 {}".for
    assert (M_test_reduced.shape[1] == 2), "M_reduced has {} columns; should have {}".f

    # 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. In the future, a good way to make a plot is to look at the Matplotlib gallery, 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 list "wo
                  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 word in words:
                  word_ind = word2ind[word]
                  x = M_reduced[word_ind][0]
                  y = M_reduced[word_ind][1]
                  plt.scatter(x, y, marker='x', color='red')
                  plt.text(x+0.02, y+0.02, word, fontsize=9)
              plt.show()
              # ------------------
```

```
In [11]:  # ---------------------
          # Run this sanity check
          # Note that this is not an exhaustive check for correctness.
          # The plot produced should look like the included file question_1.4_test.png
          # ---------------------

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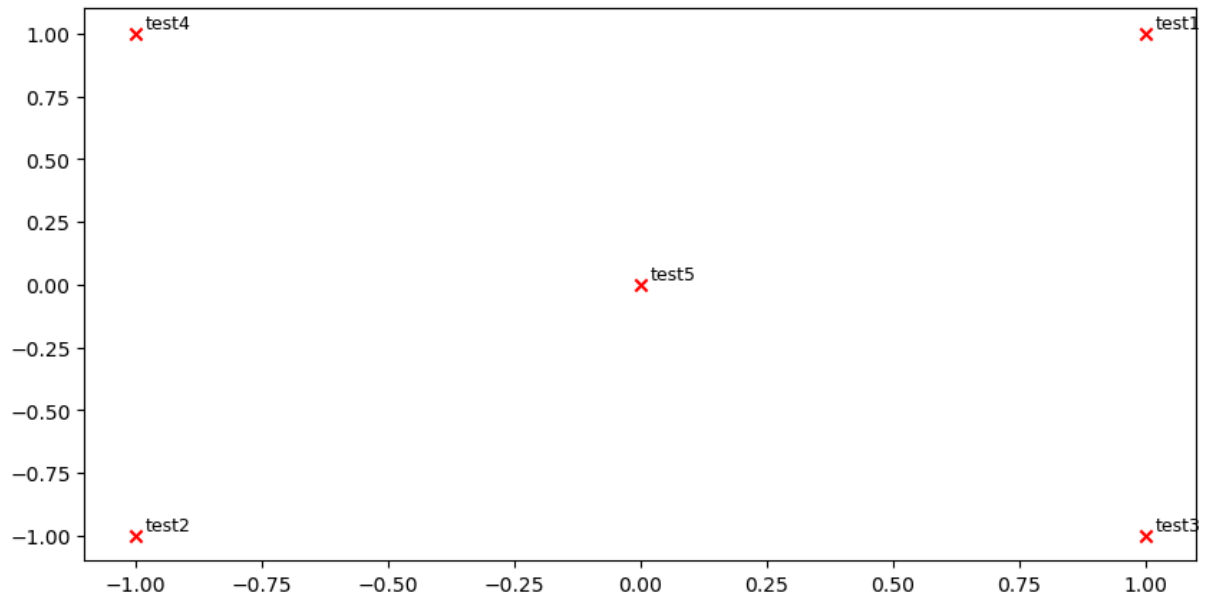
Question 1.5: Co-Occurrence Plot Analysis [written] (3 points)

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 Large Movie Review 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.

Run the below cell to produce the plot. It can take up to a few minutes to run.

In [12]:
```
# ------------------------------
# Run This Cell to Produce Your Plot
# ------------------------------
imdb_corpus = read_corpus()
M_co_occurrence, word2ind_co_occurrence = compute_co_occurrence_matrix(imdb_corpus)
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 = ['movie', 'book', 'mysterious', 'story', 'fascinating', 'good', 'interestin

plot_embeddings(M_normalized, word2ind_co_occurrence, words)
```

```
Running Truncated SVD over 5880 words...
Done.
```
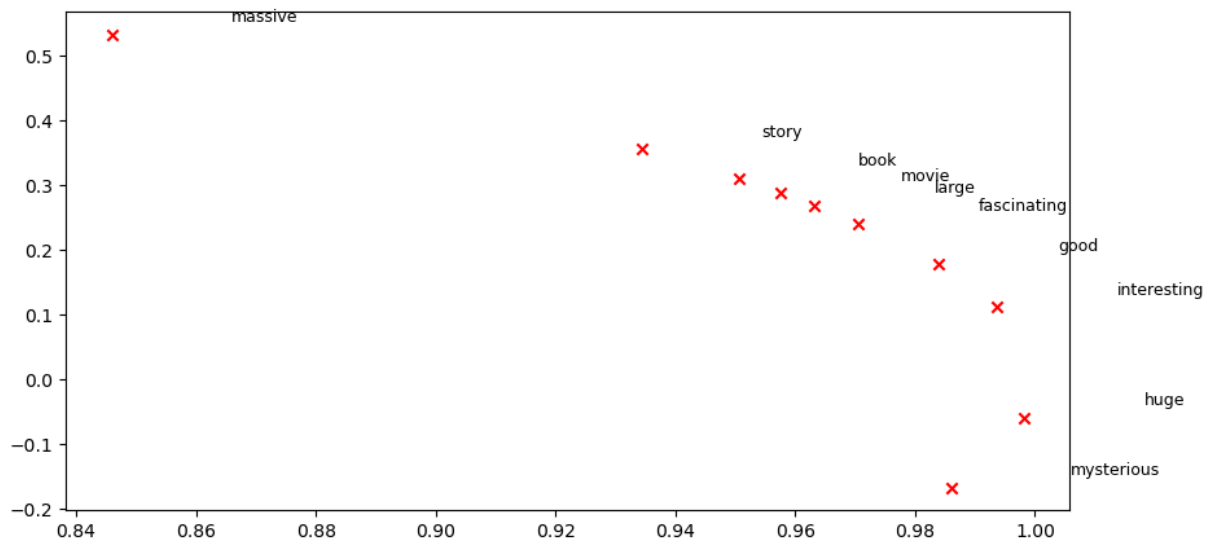

Verify that your figure matches "question_1.5.png" in the assignment zip. If not, use the figure in "question_1.5.png" to answer the next two questions.

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

Answer:

1. book and movie
2. good and fascinating

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

Answer

1. huge and massive
2. large and massive

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.

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 [13]: def load_embedding_model():
             """ Load GloVe Vectors
                 Return:
                     wv_from_bin: All 400000 embeddings, each length 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)))
             return wv_from_bin
         wv_from_bin = load_embedding_model()
```

```
Loaded vocab size 400000
```

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

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 40000 GloVe vectors instead. Run the following cells to:

1. Put 40000 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 [14]: 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 from
                 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)
             print("Shuffling words ...")
             random.seed(225)
             random.shuffle(words)
             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 [15]:
```
# ---------------------------------------------------------------
# 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 400000 words into word2ind and matrix M...
Done.
Running Truncated SVD over 400000 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.

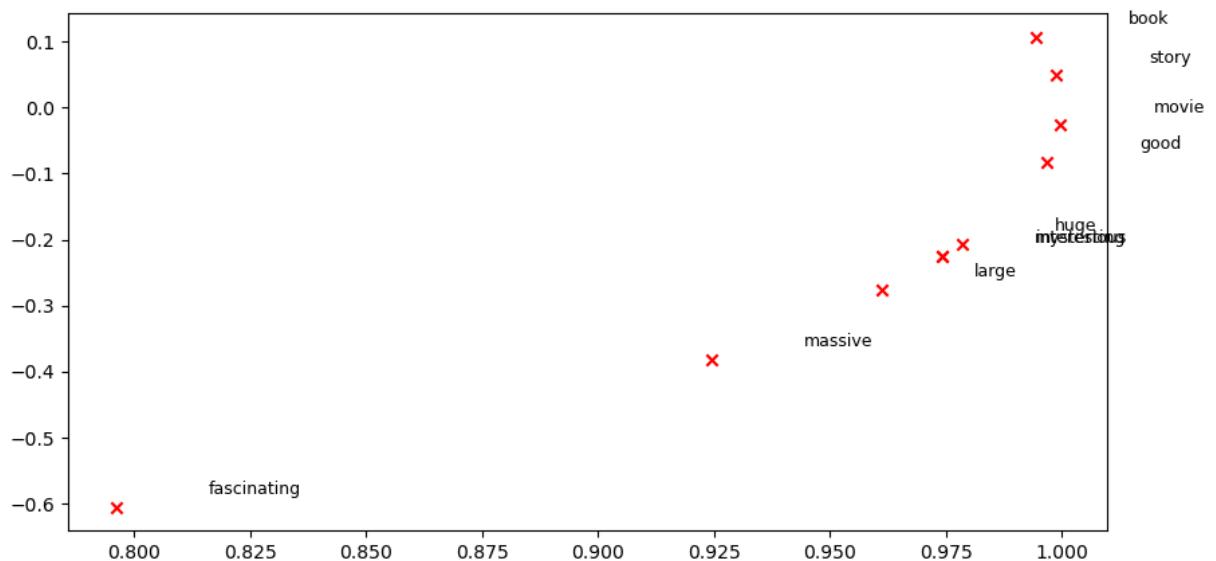## Question 2.1: GloVe Plot Analysis [written] (3 points)

Run the cell below to plot the 2D GloVe embeddings for `['movie', 'book', 'mysterious', 'story', 'fascinating', 'good', 'interesting', 'large', 'massive', 'huge']`.

In [16]:
```
words = ['movie', 'book', 'mysterious', 'story', 'fascinating', 'good', 'interestin

plot_embeddings(M_reduced_normalized, word2ind, words)
```

Verify that your figure matches "question_2.1.png" in the assignment zip. If not, use the figure in "question_2.1.png" (and the figure in "question_1.5.png", if applicable) to answer the next two questions.

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?

Answer

- It is different in that the co-ocurrence put interesting and fascinating close while GloVe plotted them far away.
- It is similiar in the fact it put book, story and movie together.

b. Why might the GloVe plot (question_2.1.png) differ from the plot generated earlier from the co-occurrence matrix (question_1.5.png)?

Answer

- GloVe use a window around words while calculating the probability of ocurrence, which gives the ability to capture syntactic and semantic information; while co-ocurrence matrix use the whole document, which gives ability to capture general topics.

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 and L2 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:

Image

Instead of computing the actual angle, we can leave the similarity in terms of $similarity = cos(\Theta)$. Formally the 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 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 most 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**.

```
In [17]:  # ------------------
          # Write your implementation here.
          word = 'right'
          polysemes = wv_from_bin.most_similar(word)
          print(polysemes) # Contain both meanings

          word = 'mouse'
          polysemes = wv_from_bin.most_similar(word)
          print(polysemes) # Contain both meanings


          word = 'band'
          polysemes = wv_from_bin.most_similar(word)
          print(polysemes) # Does not contain both meanings
          # ------------------
```

```
[('left', 0.716508150100708), ('if', 0.6925000548362732), ("n't", 0.677484571933746
3), ('back', 0.6770386099815369), ('just', 0.6740819215774536), ('but', 0.6677715778
35083), ('out', 0.6671876907348633), ('put', 0.665894091129303), ('hand', 0.66340833
90235901), ('want', 0.6615419983863831)]
[('mice', 0.6580957770347595), ('keyboard', 0.5548278093338013), ('rat', 0.543394982
8147888), ('rabbit', 0.5192376971244812), ('cat', 0.5077414512634277), ('cursor', 0.
5058691501617432), ('trackball', 0.5048902630805969), ('joystick', 0.498410522937774
66), ('mickey', 0.47242847084999084), ('clicks', 0.4722806215286255)]
[('bands', 0.8402174711227417), ('album', 0.7658628225326538), ('guitarist', 0.71624
95255470276), ('rock', 0.694769561290741), ('drummer', 0.6858715415000916), ('song',
0.6724692583084106), ('vocalist', 0.6676943302154541), ('punk', 0.6623162031173706),
('albums', 0.6582005023956299), ('musicians', 0.6396855115890503)]
```

Answer

- Mouse has the two meanings of the word in the top-10 related words. On one hand we
  have mice which is the plural of the animal related meaning of mouse, while on the
  other hand we have keyboard which is closely related to the computer related meaning
  of mouse.
- In my opinion, one possibility is that one of the meanings is not used very frequently, or
  that the related words to one of the meanings are not used with the same context
  words.

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**
for further assistance.

```
In [18]:  # ------------------
          # Write your implementation here.
          w1 = 'tall'
          w2 = 'big'
          w3 = 'short'
          print(wv_from_bin.distance(w1, w2))
          print(wv_from_bin.distance(w1, w3))

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

```
0.6236305236816406
0.6098593175411224
```

Answer

- In this case tall is closer to short than big. A possibility is because this antonyms are used in the same context more often than the synonims.

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**. 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). The answer to the analogy will have the highest cosine similarity (largest returned numerical value).

```
In [19]:   # Run this cell to answer the analogy -- man : grandfather :: woman : x
           pprint.pprint(wv_from_bin.most_similar(positive=['woman', 'grandfather'], negative=
```

```
[('grandmother', 0.7608445286750793),
 ('granddaughter', 0.7200808525085449),
 ('daughter', 0.7168302536010742),
 ('mother', 0.7151536345481873),
 ('niece', 0.7005682587623596),
 ('father', 0.6659888029098511),
 ('aunt', 0.6623408794403076),
 ('grandson', 0.6618767380714417),
 ('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, what is the expression in which we are maximizing cosine similarity with x ?

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?

Answer

x = g - m + w

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?

Answer

- Because those words are useed in the same context than near grandmother, so they are close in the word space

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 [20]:  # For example: x, y, a, b = ("", "", "", "")
          # ------------------
          # Write your implementation here.
          x, y, a, b = ("ferrari", "car", "ducati", "motorcycle")
          # ------------------

          # Test the solution
          assert wv_from_bin.most_similar(positive=[a, y], negative=[x])[0][0] == b
```

Answer

- Ferrari:Cars :: Ducati:Motorcycle

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 [21]:  pprint.pprint(wv_from_bin.most_similar(positive=['foot', 'glove'], negative=['hand'
         [('45,000-square', 0.4922032058238983),
          ('15,000-square', 0.4649604558944702),
          ('10,000-square', 0.45447564125061035),
          ('6,000-square', 0.44975772500038147),
          ('3,500-square', 0.4441334009170532),
          ('700-square', 0.44257497787475586),
          ('50,000-square', 0.43563973903656006),
          ('3,000-square', 0.43486514687538147),
          ('30,000-square', 0.4330596923828125),
          ('footed', 0.43236875534057617)]
```

Anwer

- This is because we don't have enought samples of the word foot being used as part of the body in the corpus. Instead most of the uses of foot are related to the metric square-foot.

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 [22]:  # For example: x, y, a, b = ("", "", "", "")
          # ------------------
          # Write your implementation here.
          x, y, a, b = ("windows", "microsoft", "ios", "apple")

          # ------------------
          pprint.pprint(wv_from_bin.most_similar(positive=[a, y], negative=[x]))
          assert wv_from_bin.most_similar(positive=[a, y], negative=[x])[0][0] != b
```

```
[('android', 0.595036506652832),
 ('iphone', 0.5914161801338196),
 ('ipad', 0.5612176656723022),
 ('app', 0.5501357316970825),
 ('google', 0.5390653610229492),
 ('cisco', 0.518763542175293),
 ('apps', 0.5090660452842712),
 ('smartphones', 0.5009835958480835),
 ('intel', 0.4872615933418274),
 ('smartphone', 0.4783719778060913)]
```

Answer

- In this case we tried Windows:Microsoft :: IOs: Apple but instead we retrieved Android or Iphone.

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 "man" and "profession" and most dissimilar to "woman" and (b) which terms are most similar to "woman" and "profession" and most dissimilar to "man". 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 [23]:  # Run this cell
          # Here `positive` indicates the list of words to be similar to and `negative` indic
          # most dissimilar from.
```

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

```
[('reputation', 0.5250177383422852),
 ('professions', 0.5178037881851196),
 ('skill', 0.49046966433525085),
 ('skills', 0.4900550842285156),
 ('ethic', 0.4897659420967102),
 ('business', 0.4875851273536682),
 ('respected', 0.485920250415802),
 ('practice', 0.482104629278183),
 ('regarded', 0.4778572618961334),
 ('life', 0.4760662019252777)]

[('professions', 0.5957458019256592),
 ('practitioner', 0.4988412857055664),
 ('teaching', 0.48292145133018494),
 ('nursing', 0.48211807012557983),
 ('vocation', 0.4788965880870819),
 ('teacher', 0.47160351276397705),
 ('practicing', 0.46937811374664307),
 ('educator', 0.46524322032928467),
 ('physicians', 0.4628995656967163),
 ('professionals', 0.4601393938064575)]
```

Answer

- In the case Man:Profession::Woman: we've got the answers Teaching, Nursing which is a
  gender bias in the data, because it says that the last are the most often female
  associated professions. While in the counterpart case Woman:Profession::Man: the
  results are neutral.

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 [24]:
```
# ------------------
# Write your implementation here.
pprint.pprint(wv_from_bin.most_similar(positive=['man', 'sports'], negative=['woman
pprint.pprint(wv_from_bin.most_similar(positive=['woman', 'sports'], negative=['man
# ------------------
```

```
[('sport', 0.6559334993362427),
 ('baseball', 0.6002753376960754),
 ('sporting', 0.5763919353485107),
 ('soccer', 0.5730219483375549),
 ('football', 0.5716603398323059),
 ('espn', 0.5338578224182129),
 ('entertainment', 0.5199192762374878),
 ('games', 0.5173460841178894),
 ('basketball', 0.5143382549285889),
 ('hockey', 0.4973979592323303)]
[('sport', 0.610360860824585),
 ('women', 0.5243306159973145),
 ('basketball', 0.5134516358375549),
 ('volleyball', 0.49318623542785645),
 ('soccer', 0.4838080108165741),
 ('athletes', 0.48111116886138916),
 ('athletics', 0.474430650472641),
 ('news', 0.4723058044910431),
 ('athlete', 0.46276983618736267),
 ('gymnastics', 0.4542972445487976)]
```

Answer

- In this case it assigns football and baseball as the most related sports to man and basketball and voleyball to women.

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. Your real-world example should be focused on word vectors, as opposed to bias in other AI systems (e.g., ChatGPT).

Anwer

Due to the corpus. We are using corpus that has bias.

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.

Answer

We can transform the words to genderless words before training the models. Or we can try to extract the "gender component" (after calculating it) from the vectors.

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.