A1

January 13, 2026

1 CS224N Assignment 1: Exploring Word Vectors (25 Points)

1.0.1 Due 4:30pm, Tue January 13th 2026

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.

[51]:
```
# 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] >= 8

from platform import python_version
assert int(python_version().split(".")[1]) >= 5, "Please upgrade your Python␣
 ↪version following the instructions in \
    the README.md file found in the same directory as this notebook. Your␣
 ↪Python version is " + python_version()

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", name="plain_text")
```

```
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)
# ----------------
```

1.1 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*".

1.2 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).

1.2.1 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

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.

1.2.2 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.

[52]:
```
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("␣
↪")] + [END_TOKEN] for f in files]
```

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

[53]:
```
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',
'store', 'because',
  'of', 'all', 'the', 'controversy', 'that', 'surrounded', 'it', 'when', 'it',
'was', 'first',
  'released', 'in', '1967', 'i', 'also', 'heard', 'that', 'at', 'first', 'it',
'was', 'seized',
  'by', 'us', 'customs', 'if', 'it', 'ever', 'tried', 'to', 'enter', 'this',
'country', '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',
'everything',
  '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',
'the', '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',
'classmates', 'and',
  'married', 'menbr', 'br', 'what', 'kills', 'me', 'about', 'i', 'am',
'curiousyellow', 'is',
  'that', '40', 'years', 'ago', 'this', 'was', 'considered', 'pornographic',
'really', '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', 'staple', 'in',
  'swedish', 'cinema', 'even', 'ingmar', 'bergman', 'arguably', 'their',
'answer', 'to', 'good',
  'old', 'boy', 'john', 'ford', 'had', 'sex', 'scenes', 'in', 'his', 'filmsbr',
'br', 'i', 'do',
  'commend', 'the', 'filmmakers', 'for', 'the', 'fact', 'that', 'any', 'sex',
'shown', 'in', 'the',
  'film', 'is', 'shown', 'for', 'artistic', 'purposes', 'rather', 'than',
'just', 'to', 'shock',
  'people', 'and', 'make', 'money', 'to', 'be', 'shown', 'in', 'pornographic',
'theaters', 'in',
  'america', 'i', 'am', 'curiousyellow', 'is', 'a', 'good', 'film', 'for',
'anyone', 'wanting',
  'to', 'study', 'the', 'meat', 'and', 'potatoes', 'no', 'pun', 'intended',
'of', 'swedish',
  'cinema', 'but', 'really', 'this', 'film', 'doesnt', 'have', 'much', 'of',
'a', 'plot', '<END>'],
 ['<START>', 'i', 'am', 'curious', 'yellow', 'is', 'a', 'risible', 'and',
'pretentious', 'steaming',
  'pile', 'it', 'doesnt', 'matter', 'what', 'ones', 'political', 'views', 'are',
'because', 'this',
  'film', 'can', 'hardly', 'be', 'taken', 'seriously', 'on', 'any', 'level',
'as', 'for', 'the',
  'claim', 'that', 'frontal', 'male', 'nudity', 'is', 'an', 'automatic', 'nc17',
'that', 'isnt',
  'true', 'ive', 'seen', 'rrated', 'films', 'with', 'male', 'nudity', 'granted',
'they', 'only',
  'offer', 'some', 'fleeting', 'views', 'but', 'where', 'are', 'the', 'rrated',
'films', 'with',
  'gaping', 'vulvas', 'and', 'flapping', 'labia', 'nowhere', 'because', 'they',
'dont', 'exist',
  'the', 'same', 'goes', 'for', 'those', 'crappy', 'cable', 'shows', 'schlongs',
'swinging', 'in',
  'the', 'breeze', 'but', 'not', 'a', 'clitoris', 'in', 'sight', 'and', 'those',
'pretentious',
  'indie', 'movies', 'like', 'the', 'brown', 'bunny', 'in', 'which', 'were',
'treated', 'to', 'the',
  'site', 'of', 'vincent', 'gallos', 'throbbing', 'johnson', 'but', 'not', 'a',
'trace', 'of',
  'pink', 'visible', 'on', 'chloe', 'sevigny', 'before', 'crying', 'or',
'implying',
  'doublestandard', 'in', 'matters', 'of', 'nudity', 'the', 'mentally',
'obtuse', 'should', 'take',
  'into', 'account', 'one', 'unavoidably', 'obvious', 'anatomical',
'difference', 'between', 'men',
  'and', 'women', 'there', 'are', 'no', 'genitals', 'on', 'display', 'when',
'actresses', 'appears',
  'nude', 'and', 'the', 'same', 'cannot', 'be', 'said', 'for', 'a', 'man', 'in',
```

```
'fact', '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',
'admittedly',
  'depressing', 'ability', 'to', 'come', 'to', 'terms', 'culturally', 'with',
'the', 'insides',
  'of', 'womens', 'bodies', '<END>'],
 ['<START>', 'if', 'only', 'to', 'avoid', 'making', 'this', 'type', 'of',
'film', 'in', 'the',
  'future', 'this', 'film', 'is', 'interesting', 'as', 'an', 'experiment',
'but', 'tells', 'no',
  'cogent', 'storybr', 'br', 'one', 'might', 'feel', 'virtuous', 'for',
'sitting', 'thru', 'it',
  'because', 'it', 'touches', 'on', 'so', 'many', 'important', 'issues', 'but',
'it', 'does', 'so',
  'without', 'any', 'discernable', 'motive', 'the', 'viewer', 'comes', 'away',
'with', 'no', 'new',
  'perspectives', 'unless', 'one', 'comes', 'up', 'with', 'one', 'while',
'ones', 'mind', '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
```

1.2.3 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.

[54]:
```
def distinct_words(corpus):
    """ Determine a list of distinct words for the corpus.
        Params:
            corpus (list of list of strings): corpus of documents
        Return:
            corpus_words (list of strings): sorted list of distinct words␣
 ↪across the corpus
            n_corpus_words (integer): number of distinct words across the corpus
```

```
    """
    corpus_words = []
    n_corpus_words = -1

    # ------------------
    # Write your implementation here.
    flattened_corpus = [word for word_list in corpus for word in word_list]
    word_set = set(flattened_corpus)
    corpus_words = sorted(word_set)
    n_corpus_words = len(corpus_words)
    # ------------------

    return corpus_words, n_corpus_words
```

[55]:
```
# ---------------------
# 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).split(" "), "{} All's well that ends well {}".format(START_TOKEN,␣
 ↪END_TOKEN).split(" ")]
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", "glitters", "isn't", "well", END_TOKEN])
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␣
 ↪words. Correct: {}. Yours: {}".format(ans_num_corpus_words, num_corpus_words)

# Test correct words
assert (test_corpus_words == ans_test_corpus_words), "Incorrect corpus_words.
 ↪\nCorrect: {}\nYours:   {}".format(str(ans_test_corpus_words),␣
 ↪str(test_corpus_words))

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

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

1.2.4 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.

[56]:
```
def compute_co_occurrence_matrix(corpus, window_size=4):
    """ Compute co-occurrence matrix for the given corpus and window_size␣
↪(default of 4).

        Note: Each word in a document should be at the center of a window.␣
↪Words near edges will have a smaller
              number of co-occurring words.

              For example, if we take the document "<START> All that glitters␣
↪is not gold <END>" with window size of 4,
              "All" will co-occur with "<START>", "that", "glitters", "is", and␣
↪"not".

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

    # ------------------
    # Write your implementation here.
    M = np.zeros((n_words, n_words), dtype=int)
    word2ind = {word : i for i, word in enumerate(words)}

    def increment_matrix(cur_idx, document, M):
        adj_word = document[cur_idx]
        wj = word2ind[adj_word]
        M[wi][wj] += 1
        return M
```

```
    for document in corpus:
        for i, w in enumerate(document):
            wi = word2ind[w]
            for j in range(1, window_size + 1):
                cur_idx = i - j
                if cur_idx < 0:
                    break
                M = increment_matrix(cur_idx, document,  M)
            for j in range(1, window_size + 1):
                cur_idx = i + j
                if cur_idx >= len(document):
                    break
                M = increment_matrix(cur_idx, document, M)
    # ------------------

    return M, word2ind
```

[57]:
```
# ---------------------
# 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).split(" "), "{} All's well that ends well {}".format(START_TOKEN,␣
 ↪END_TOKEN).split(" ")]
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", "glitters", "isn't", "well", END_TOKEN])
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:␣
 ↪{}\nYours: {}".format(word2ind_ans, word2ind_test)

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

# 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␣
 ↪matrix M. Yours has {} but should have {}.".format(idx1, idx2, w1, w2,␣
 ↪student, correct))

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

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

1.2.5 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.

[58]:
```
def reduce_to_k_dim(M, k=2):
    """ Reduce a co-occurence count matrix of dimensionality (num_corpus_words,␣
 ↪num_corpus_words)
        to a matrix of dimensionality (num_corpus_words, k) using the following␣
 ↪SVD function from Scikit-Learn:
```

```
            - http://scikit-learn.org/stable/modules/generated/sklearn.
 ↪decomposition.TruncatedSVD.html

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

    # ------------------
    # Write your implementation here.
    svd = TruncatedSVD(n_components=k, n_iter=n_iters)
    M_reduced = svd.fit_transform(M)
    # ------------------

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

[59]:
```
# ---------------------
# 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).split(" "), "{} All's well that ends well {}".format(START_TOKEN,␣
 ↪END_TOKEN).split(" ")]
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 {}".
 ↪format(M_test_reduced.shape[0], 10)
assert (M_test_reduced.shape[1] == 2), "M_reduced has {} columns; should have␣
 ↪{}".format(M_test_reduced.shape[1], 2)

# Print Success
```

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

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

1.2.6 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.

[60]:
```
def plot_embeddings(M_reduced, word2ind, words):
    """ Plot in a scatterplot the embeddings of the words specified in the list␣
 ↪"words".
        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 , 2)): matrix of 2-dimensioal word embeddings
            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:
        embedding = M_reduced[word2ind[word]]
        x = embedding[0]
        y = embedding[1]
        plt.scatter(x, y, marker='o', color='red')
        plt.text(x + .001, y + .001, s=f"{word}", fontsize=9)
    plt.show()
    # ------------------
```

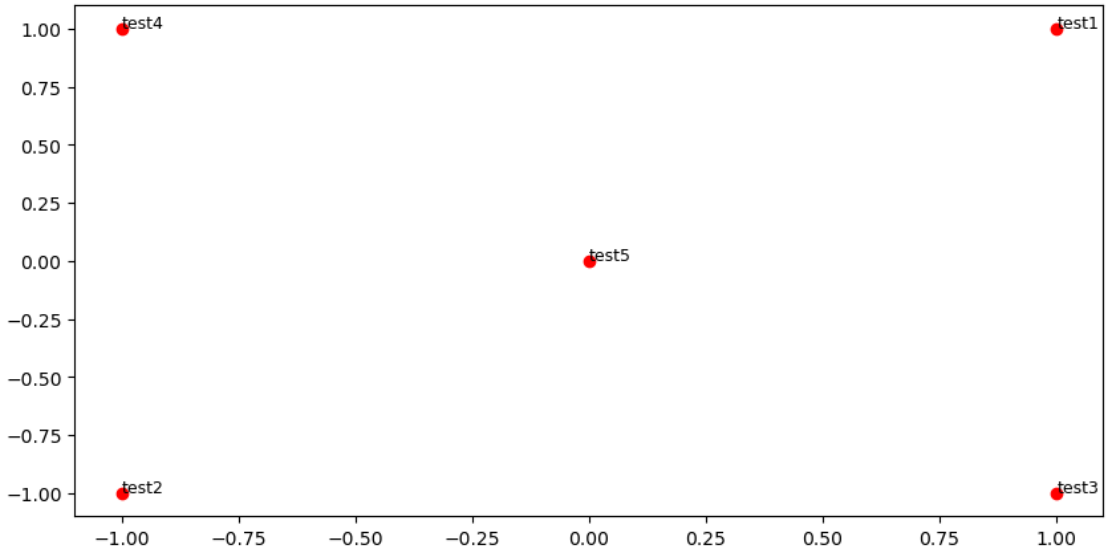
[61]:
```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness.
# The plot produced should look like the included file question_1.4_test.png
# ---------------------
```

```
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:
```




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

1.2.7 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.

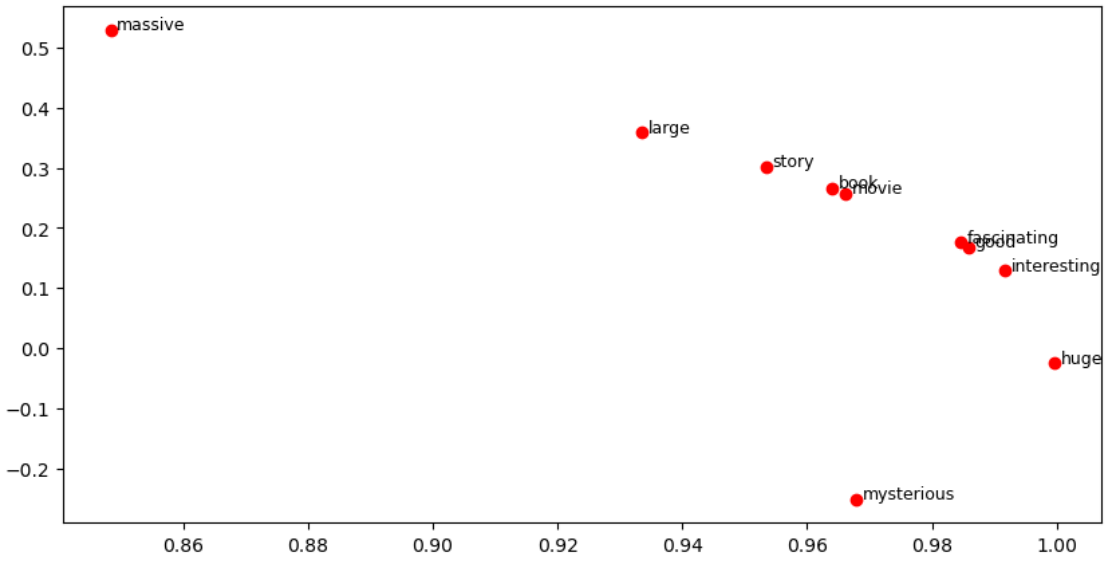
[62]:
```
# ----------------------------
# Run This Cell to Produce Your Plot
# ----------------------------
imdb_corpus = read_corpus()
M_co_occurrence, word2ind_co_occurrence =␣
 ↪compute_co_occurrence_matrix(imdb_corpus)
M_reduced_co_occurrence = reduce_to_k_dim(M_co_occurrence, k=2)

# 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',␣
 ↪'interesting', 'large', 'massive', 'huge']

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.

The words `book` and `movie` cluster together. This cluster happens because books and movies are often used in similar contexts. Many stories have books and movies attached to them. These words will be counted under the same scenarios and around

the same company of words thus have similarity in the embedding space as we define words by the company they keep. Next, the words `Fascinating` and `Good` are clustered together. Again, these words are utilized under similar scenarios and thus will often keep the same company/context. As such, they are near each other in the semantic embedding space.

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

Ituitively, I would think that `Large` and `Huge` would cluster together. They seem like they would be used in the same contexts, keeping the same company, having similar count entries, and subsequently similar embeddings. Next, I would also think `Mysterious` and `Intersting` would cluster together. They both have sense of intrigue and I would assume that they keep the same company. There could be a few reasons for these unexpected discrepancies: Size of data corpus, specific text in the data corpus, and loss in dimensionality reduction could be valid reasons we see this dissonance.

1.3 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.

[63]:
```
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.

1.3.1 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.

[64]:
```
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 file
        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
```

[65]:
```
# -----------------------------------------------------------------
# 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.
```

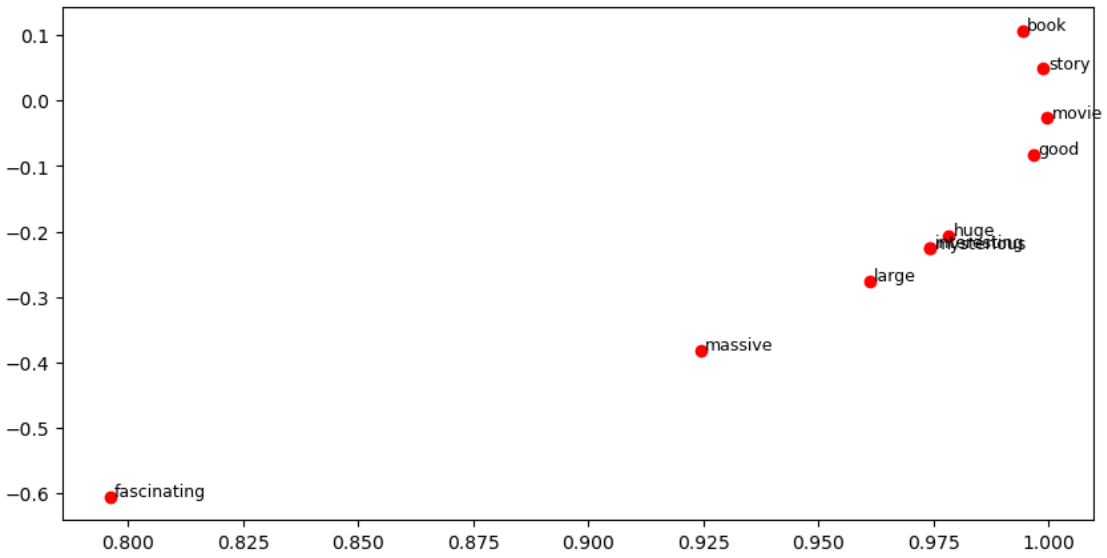
Note: If you are receiving out of memory issues on your local machine, try closing other applications to free more memory on your device. You may want to try restarting your machine so that you can free up extra memory. Then immediately run the jupyter notebook and see if you can load the word vectors 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.

1.3.2 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']`.

[66]:
```
words = ['movie', 'book', 'mysterious', 'story', 'fascinating', 'good',␣
 ↪'interesting', 'large', 'massive', 'huge']

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?

One difference is `Mysterious` and `interesting` are now basically entirely overlapping and clustered, whereas they were not nearly as close in image 1.5. One similarity is that story is still relatively close to book and movie.

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)?

GloVe learns its embeddings in a different manner than the co-occurence matrices. It learns global relationships and focus on probalities and ratios.

1.3.3 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:

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:

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

1.3.4 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**.

[67]:
```
# ------------------
# Write your implementation here.
pprint.pprint(wv_from_bin.most_similar("capital"))
# ------------------
```

```
[('city', 0.604853630065918),
 ('investment', 0.5984903573989868),
 ('outskirts', 0.5668216347694397),
 ('outside', 0.5614592432975769),
 ('cities', 0.561435341835022),
 ('government', 0.5524670481681824),
 ('downtown', 0.5211317539215088),
 ('in', 0.5146748423576355),
 ('central', 0.5084695219993591),
 ('province', 0.5082636475563049)]
```

The word I discovered is "capital", which exhibits two distinct meanings in its top 10 most similar words. The first meaning relates to a capital city (geographic/political sense), evidenced by words such as "city," "cities," "downtown," "central," "province," and "outskirts." The second meaning pertains to financial capital or money, represented by the word "investment," which ranks second with a similarity score of 0.598. Many polysemous and homonymic words fail to exhibit both meanings in their top 10 results because one sense typically dominates in frequency and context within the training corpus. When a word has a primary usage that appears far more frequently in text, the word embeddings will predominantly capture semantic relationships for that dominant meaning, causing the top similar words to cluster around only one sense. Words with more balanced usage across their multiple meanings, like "capital," are more likely to show representation of both senses in their nearest neighbors, though even here the geographic meaning appears more frequently than the financial one.

1.3.5 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.

[68]:
```
# fast/quick (synonyms) vs fast/slow (antonyms)
w1, w2, w3 = "fast", "quick", "slow"
dist_synonym = wv_from_bin.distance(w1, w2)
```

```
dist_antonym = wv_from_bin.distance(w1, w3)
print(f"{w1}-{w2} (synonyms): {dist_synonym:.4f}")
print(f"{w1}-{w3} (antonyms): {dist_antonym:.4f}")
print(f"Antonym closer? {dist_antonym < dist_synonym}\n")
```

```
fast-quick (synonyms): 0.3329
fast-slow (antonyms): 0.2523
Antonym closer? True
```

The words I found are w_1="fast", w_2="quick", and w_3="slow", where "fast" and "quick" are synonyms, and "fast" and "slow" are antonyms. The cosine distance between "fast" and "slow" (0.2523) is smaller than the distance between "fast" and "quick" (0.3329), meaning the antonym is closer than the synonym. This counter-intuitive result likely occurs because antonyms frequently appear in similar contexts as they describe the same dimension or property (speed) and often co-occur in comparative statements like "not fast but slow" or "fast or slow." In contrast, while "fast" and "quick" are synonyms, they may be used in slightly different contexts or registers. The GloVe embeddings capture distributional similarity based on co-occurrence patterns, so words that appear in similar linguistic contexts will have similar embeddings, regardless of whether they have opposite or similar meanings.

1.3.6 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).

[69]:
```
# Run this cell to answer the analogy -- man : grandfather :: woman : x
pprint.pprint(wv_from_bin.most_similar(positive=['woman', 'grandfather'],␣
 ↪negative=['man']))
```

```
[('grandmother', 0.7608445286750793),
 ('granddaughter', 0.7200808525085449),
 ('daughter', 0.7168302536010742),
 ('mother', 0.7151536345481873),
 ('niece', 0.7005683183670044),
 ('father', 0.6659887433052063),
 ('aunt', 0.6623408794403076),
 ('grandson', 0.6618767380714417),
 ('grandparents', 0.644661009311676),
 ('wife', 0.644535481929779)]
```

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?

x = w + (g - m). Grandfather minus man should represent this generation relationship that we are trying to uncover a corollary for for woman. This is what the most_similar is doing by finding the furtherest distance from man while closet to woman and grandfather.

1.3.7 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?

The vector expression w + (g - m) captures two key semantic dimensions: female gender (from "woman") and familial/generational relationships (from "grandfather" - "man"). Words like "granddaughter," "daughter," and "mother" rank highly because they satisfy both dimensions—they are female family members with generational connections. While "grandmother" is the exact analog with the same generational distance, these other words are semantically close because they share the dominant features of being female relatives within a family structure. The word vectors don't encode precise generational arithmetic; instead, they capture broader semantic similarity, causing related family terms to cluster together in the embedding 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!

[70]:
```
# For example: x, y, a, b = ("", "", "", "")
# ------------------
# Write your implementation here.

x, y, a, b = ("king", "queen", "man", "woman")
# ------------------

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

The analogy holds because both pairs represent a consistent gender transformation from male to female, a relationship strongly encoded in the word embeddings due to frequent parallel usage in text.

1.3.8 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?

[71]:
```
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', 0.44257503747940063),
 ('50,000-square', 0.4356396794319153),
 ('3,000-square', 0.43486514687538147),
 ('30,000-square', 0.4330596923828125),
 ('footed', 0.43236875534057617)]
```

The analogy likely failed because "foot" has a strong secondary meaning as a unit of measurement (i.e. "square foot"), which dominates its usage in the training corpus compared to the body part meaning. When the vector arithmetic combines "foot" + "glove" - "hand", the measurement context is amplified by further removing context when it might be relating to a body part, leading to results like "45,000-square" and other area measurements that commonly use "square foot" as a unit, event though intuitvely as humans we think of foot as the body part first.

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'**).

[72]:
```
# For example: x, y, a, b = ("", "", "", "")
# ------------------
# Write your implementation here.
x, y, a, b = ("pen", "write", "brush", "paint")
# ------------------
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
```

```
[('cover', 0.4890002906322479),
 ('you', 0.46037745475769043),
 ('done', 0.4517306685447693),
 ("'d", 0.4509442150592804),
 ('do', 0.443452924489975),
 ("'ll", 0.44196340441703796),
 ('let', 0.44050753116607666),
 ('things', 0.425790399312973),
 ('work', 0.4217274487018585),
 ('follow', 0.4135601818561554)]
```

Intended analogy: pen:write :: brush:paint

Incorrect result: cover

Explanation: The analogy likely failed because "brush" is highly polysemous, appearing frequently in contexts unrelated to painting (brushing hair, brushing teeth, brush as vegetation), which dilutes the tool-action relationship the analogy intended to capture.

1.3.9 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.

[73]:
```
# Run this cell
# Here `positive` indicates the list of words to be similar to and `negative`␣
 ↪indicates the list of words to be
# most dissimilar from.

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

```
[('reputation', 0.5250176787376404),
 ('professions', 0.5178037881851196),
 ('skill', 0.49046966433525085),
 ('skills', 0.49005505442619324),
 ('ethic', 0.4897659420967102),
 ('business', 0.487585186958313),
 ('respected', 0.4859202802181244),
 ('practice', 0.482104629278183),
 ('regarded', 0.4778572916984558),
 ('life', 0.4760662019252777)]

[('professions', 0.5957458019256592),
 ('practitioner', 0.4988412857055664),
 ('teaching', 0.48292139172554016),
 ('nursing', 0.48211804032325745),
 ('vocation', 0.4788965880870819),
 ('teacher', 0.47160348296165466),
 ('practicing', 0.46937811374664307),
 ('educator', 0.46524327993392944),
```

```
 ('physicians', 0.4628995656967163),
 ('professionals', 0.4601394236087799)]
```

Terms most similar to "man" and "profession" but dissimilar to "woman" include: reputation, professions, skill, skills, business, respected, practice, and regarded.

Terms most similar to "woman" and "profession" but dissimilar to "man" include: teaching, nursing, practitioner, teacher, educator, and vocation.

Difference and Gender Bias: The results reveal significant gender bias in the word embeddings. Male-associated professional terms are broad, abstract concepts like "skill," "business," "reputation," and "respected," suggesting authority and general expertise. In contrast, female-associated professional terms are overwhelmingly concentrated in stereotypically feminine, caregiving roles such as "teaching" and "nursing." This reflects societal biases present in the training corpus, where men are linguistically associated with diverse, high-status professions while women are disproportionately linked to nurturing, service-oriented careers. Such biases can perpetuate stereotypes when these embeddings are used in downstream applications.

1.3.10 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.

[74]:
```
# ------------------
# Write your implementation here.
# Socioeconomic bias
print("Socioeconomic Bias")
print("Similar to 'rich' and 'intelligent', dissimilar to 'poor':")
pprint.pprint(wv_from_bin.most_similar(positive=['rich', 'intelligent'],␣
 ↪negative=['poor']))
print("\nSimilar to 'poor' and 'intelligent', dissimilar to 'rich':")
pprint.pprint(wv_from_bin.most_similar(positive=['poor', 'intelligent'],␣
 ↪negative=['rich']))
# ------------------
```

```
Socioeconomic Bias
Similar to 'rich' and 'intelligent', dissimilar to 'poor':
[('wonderfully', 0.5110443830490112),
 ('thoughtful', 0.5088942646980286),
 ('interesting', 0.4909347593784332),
 ('smart', 0.4881010353565216),
 ('personable', 0.47142913937568665),
 ('incredibly', 0.46384137868881226),
 ('articulate', 0.45878952741622925),
 ('inventive', 0.4586627781391144),
 ('fascinating', 0.457888126373291),
 ('witty', 0.4540828764438629)]
```

```
Similar to 'poor' and 'intelligent', dissimilar to 'rich':
[('smart', 0.47221463918685913),
 ('decent', 0.4584115147590637),
 ('handicapped', 0.4577614367008209),
 ('inadequate', 0.4408053159713745),
 ('competent', 0.43579554557800293),
 ('indifferent', 0.4273439049720764),
 ('uneducated', 0.4257780909538269),
 ('creationism', 0.4245559871196747),
 ('caring', 0.41887524724006653),
 ('systems', 0.4168367087841034)]
[('smart', 0.47221463918685913),
 ('decent', 0.4584115147590637),
 ('handicapped', 0.4577614367008209),
 ('inadequate', 0.4408053159713745),
 ('competent', 0.43579554557800293),
 ('indifferent', 0.4273439049720764),
 ('uneducated', 0.4257780909538269),
 ('creationism', 0.4245559871196747),
 ('caring', 0.41887524724006653),
 ('systems', 0.4168367087841034)]
```

This example reveals significant socioeconomic bias in the word embeddings. When combining "rich" and "intelligent," the most similar terms are overwhelmingly positive descriptors like "wonderfully," "thoughtful," "articulate," "inventive," and "fascinating"—suggesting that wealth is linguistically associated with admirable intellectual and personal qualities. In contrast, combining "poor" and "intelligent" yields more neutral or negative terms like "decent," "inadequate," "handicapped," "indifferent," and "uneducated." This bias reflects societal stereotypes present in the training data that conflate wealth with intelligence and positive attributes, while implicitly linking poverty with lesser capabilities or negative characteristics, perpetuating harmful socioeconomic prejudices.

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

a. Give one possible explanation of how bias gets into the word vectors. Your explanation should be focused on word vectors, as opposed to bias in other AI systems (e.g., ChatGPT). You can use specific historical examples to back up your explanations if necessary.

Word vectors learn from large text corpora (such as Wikipedia, news articles, and web text) that reflect real-world language patterns, which inherently contain societal biases and stereotypes. Because word embeddings are trained using the distributional hypothesis—where words appearing in similar contexts receive similar vector representations—any biased associations present in the training data become encoded in the vectors. For example, if "doctor" frequently appears with male pronouns and "nurse" with female pronouns in the corpus, the word vectors will capture and reinforce these gender associations. Thus, bias enters word vectors directly through the biased language patterns present in human-generated text used for training.

b. What is one possible method you can use to mitigate bias exhibited by word vectors? Briefly explain the method and what the goal of the method was.

One approach might be to identify biases as we have done above and subsequently curate more balanced training data by intentionally including text that provides counter-stereotypical examples alongside traditional ones. For instance, we could supplement the training corpus with articles and texts that feature women in STEM careers, men in caregiving professions, and diverse representations across all fields and socioeconmic lines.

2 Submission Instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Select Edit -> Clear Outputs of All Cells. This will clear all the outputs from all cells (but will keep the content of all cells).
3. Select Run -> Run All Cells. This will run all the cells in order, and will take several minutes.
4. Once you've rerun everything, select File -> Save and Export Notebook as -> PDF (If you see errors like "nbconvert failed: Pandoc wasn't found", you can first save it as HTML). Select File -> Save and Export Notebook as -> HTML. This will save the notebook as an HTML file on your computer. Open the downloaded HTML file in your web browser. In the browser, press Ctrl + P (Windows/Linux) or Cmd + P (Mac) to open the print dialog. In the print dialog, change the destination to Save as PDF and click Save. 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.