CSCI 4140: Natural Language Processing

CSCI/DASC 6040: Computational Analysis of Natural Languages

Spring 2025

Homework 2 - Vector embeddings

Due Sunday, February 2, at 11:59 PM

A lot of code is provided in this notebook, and we highly encourage you to read and understand it, as part of your learning.

Assignment Notes: Please make sure to save the notebook as you go along.

In [36]:
```
!pip install gensim
!pip install nltk
!pip install scikit-learn
```

```
Requirement already satisfied: gensim in c:\users\owens\miniconda3\lib\site-packages
(4.3.3)
Requirement already satisfied: numpy<2.0,>=1.18.5 in c:\users\owens\miniconda3\lib\s
ite-packages (from gensim) (1.26.4)
Requirement already satisfied: scipy<1.14.0,>=1.7.0 in c:\users\owens\miniconda3\lib
\site-packages (from gensim) (1.13.1)
Requirement already satisfied: smart-open>=1.8.1 in c:\users\owens\miniconda3\lib\si
te-packages (from gensim) (7.1.0)
Requirement already satisfied: wrapt in c:\users\owens\miniconda3\lib\site-packages
(from smart-open>=1.8.1->gensim) (1.17.2)
Requirement already satisfied: nltk in c:\users\owens\miniconda3\lib\site-packages
(3.9.1)
Requirement already satisfied: click in c:\users\owens\miniconda3\lib\site-packages
(from nltk) (8.1.7)
Requirement already satisfied: joblib in c:\users\owens\miniconda3\lib\site-packages
(from nltk) (1.4.2)
Requirement already satisfied: regex>=2021.8.3 in c:\users\owens\miniconda3\lib\site
-packages (from nltk) (2024.11.6)
Requirement already satisfied: tqdm in c:\users\owens\miniconda3\lib\site-packages
(from nltk) (4.66.5)
Requirement already satisfied: colorama in c:\users\owens\miniconda3\lib\site-packag
es (from click->nltk) (0.4.6)
Requirement already satisfied: scikit-learn in c:\users\owens\miniconda3\lib\site-pa
ckages (1.5.1)
Requirement already satisfied: numpy>=1.19.5 in c:\users\owens\miniconda3\lib\site-p
ackages (from scikit-learn) (1.26.4)
Requirement already satisfied: scipy>=1.6.0 in c:\users\owens\miniconda3\lib\site-pa
ckages (from scikit-learn) (1.13.1)
Requirement already satisfied: joblib>=1.2.0 in c:\users\owens\miniconda3\lib\site-p
ackages (from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in c:\users\owens\miniconda3\lib
\site-packages (from scikit-learn) (3.5.0)
```

In [37]:

```python
# All Import Statements Defined Here
# Note: Do not add to this list.
# ----------------

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

from platform import python_version
assert int(python_version().split(".")[1]) >= 5, "Your Python version is " + python

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

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

import numpy as np
import random
import scipy as sp
from sklearn.decomposition import TruncatedSVD
from sklearn.decomposition import PCA

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

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

```
[nltk_data] Downloading package reuters to
[nltk_data]     C:\Users\owens\AppData\Roaming\nltk_data...
[nltk_data]   Package reuters is already up-to-date!
```

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 (50 pts)

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).

Co-Occurrence

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

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

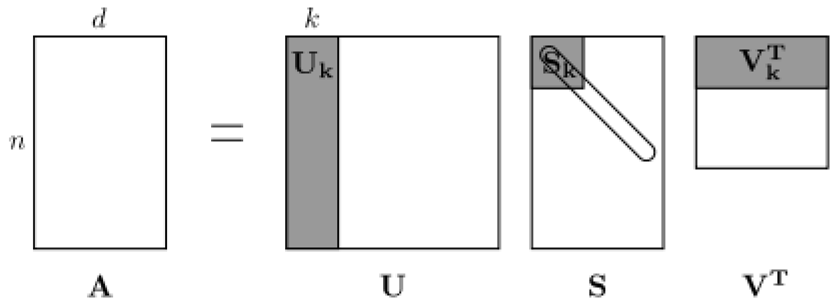
Document 1: "all that glitters is not gold"

Document 2: "all is well that ends well"

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

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

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



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

Notes: If you can barely remember what an eigenvalue is, here's a slow, friendly introduction to SVD. If you want to learn more thoroughly about PCA or SVD, feel free to check out lectures 7, 8, and 9 of CS168. These course notes provide a great high-level treatment of these general purpose algorithms. Though, for the purpose of this class, you only need to know how to extract the k-dimensional embeddings by utilizing pre-programmed implementations of these algorithms from the numpy, scipy, or sklearn python packages. In practice, it is challenging to apply full SVD to large corpora because of the memory needed to perform PCA or SVD. However, if you only want the top k vector components for relatively small k — known as Truncated SVD — then there are reasonably scalable techniques to compute those iteratively.

Plotting Co-Occurrence Word Embeddings

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

In [38]:
```
def read_corpus(category="gold"):
    """ Read files from the specified Reuter's category.
        Params:
            category (string): category name
        Return:
            list of lists, with words from each of the processed files
    """
    files = reuters.fileids(category)
    return [[START_TOKEN] + [w.lower() for w in list(reuters.words(f))] + [END_TOKE
```

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

In [39]:
```
reuters_corpus = read_corpus()
pprint.pprint(reuters_corpus[:3], compact=True, width=100)
```

```
[['<START>', 'western', 'mining', 'to', 'open', 'new', 'gold', 'mine', 'in', 'austra
lia', 'western',
  'mining', 'corp', 'holdings', 'ltd', '&', 'lt', ';', 'wmng', '.', 's', '>', '(',
'wmc', ')',
  'said', 'it', 'will', 'establish', 'a', 'new', 'joint', 'venture', 'gold', 'mine',
'in', 'the',
  'northern', 'territory', 'at', 'a', 'cost', 'of', 'about', '21', 'mln', 'dlrs',
'.', 'the',
  'mine', ',', 'to', 'be', 'known', 'as', 'the', 'goodall', 'project', ',', 'will',
'be', 'owned',
  '60', 'pct', 'by', 'wmc', 'and', '40', 'pct', 'by', 'a', 'local', 'w', '.', 'r',
'.', 'grace',
  'and', 'co', '&', 'lt', ';', 'gra', '>', 'unit', '.', 'it', 'is', 'located', '30',
'kms', 'east',
  'of', 'the', 'adelaide', 'river', 'at', 'mt', '.', 'bundey', ',', 'wmc', 'said',
'in', 'a',
  'statement', 'it', 'said', 'the', 'open', '-', 'pit', 'mine', ',', 'with', 'a', 'c
onventional',
  'leach', 'treatment', 'plant', ',', 'is', 'expected', 'to', 'produce', 'about', '5
0', ',', '000',
  'ounces', 'of', 'gold', 'in', 'its', 'first', 'year', 'of', 'production', 'from',
'mid', '-',
  '1988', '.', 'annual', 'ore', 'capacity', 'will', 'be', 'about', '750', ',', '00
0', 'tonnes', '.',
  '<END>'],
 ['<START>', 'belgium', 'to', 'issue', 'gold', 'warrants', ',', 'sources', 'say', 'b
elgium',
  'plans', 'to', 'issue', 'swiss', 'franc', 'warrants', 'to', 'buy', 'gold', ',', 'w
ith', 'credit',
  'suisse', 'as', 'lead', 'manager', ',', 'market', 'sources', 'said', '.', 'no', 'c
onfirmation',
  'or', 'further', 'details', 'were', 'immediately', 'available', '.', '<END>'],
 ['<START>', 'belgium', 'launches', 'bonds', 'with', 'gold', 'warrants', 'the', 'kin
gdom', 'of',
  'belgium', 'is', 'launching', '100', 'mln', 'swiss', 'francs', 'of', 'seven', 'yea
r', 'notes',
  'with', 'warrants', 'attached', 'to', 'buy', 'gold', ',', 'lead', 'mananger', 'cre
dit', 'suisse',
  'said', '.', 'the', 'notes', 'themselves', 'have', 'a', '3', '-', '3', '/', '8',
'pct', 'coupon',
  'and', 'are', 'priced', 'at', 'par', '.', 'payment', 'is', 'due', 'april', '30',
',', '1987',
  'and', 'final', 'maturity', 'april', '30', ',', '1994', '.', 'each', '50', ',', '0
00', 'franc',
  'note', 'carries', '15', 'warrants', '.', 'two', 'warrants', 'are', 'required', 't
o', 'allow',
  'the', 'holder', 'to', 'buy', '100', 'grammes', 'of', 'gold', 'at', 'a', 'price',
'of', '2', ',',
  '450', 'francs', ',', 'during', 'the', 'entire', 'life', 'of', 'the', 'bond', '.',
'the',
  'latest', 'gold', 'price', 'in', 'zurich', 'was', '2', ',', '045', '/', '2', ',',
'070', 'francs',
  'per', '100', 'grammes', '.', '<END>']]
```

Question 1.1: Implement `distinct_words` [code] (10 pts)

Write a method to work out the distinct words (word types) that occur in the corpus. You can do this with `for` loops, but it's more efficient to do it with Python list comprehensions. In particular, this 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 [40]:
```
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
    """
    corpus_words = []
    n_corpus_words = -1

    corpus_words = sorted(set(word for document in corpus for word in document))
    n_corpus_words = len(corpus_words)

    return corpus_words, n_corpus_words
```

In [41]:
```
# ---------------------
# 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] (15 pts)

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

In [42]:
```
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)
    word2ind = {word: i for i, word in enumerate(words)}
    M = np.zeros((n_words, n_words))

    for document in corpus:
        for i, word in enumerate(document):
            word_idx = word2ind[word]
            start = max(i - window_size, 0)
            end = min(i + window_size + 1, len(document))

            for j in range(start, end):
                if i != j:  # Avoid self-co-occurrence
                    context_word = document[j]
                    context_idx = word2ind[context_word]
                    M[word_idx, context_idx] += 1

    return M, word2ind
```

In [43]:
```
# ---------------------
# 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] (5 pts)

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.

In [44]:
```
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]))

    svd = TruncatedSVD(n_components=k, n_iter=n_iters, random_state=0)
    M_reduced = svd.fit_transform(M)

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

In [45]:
```
# ---------------------
# 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] (5 pts)

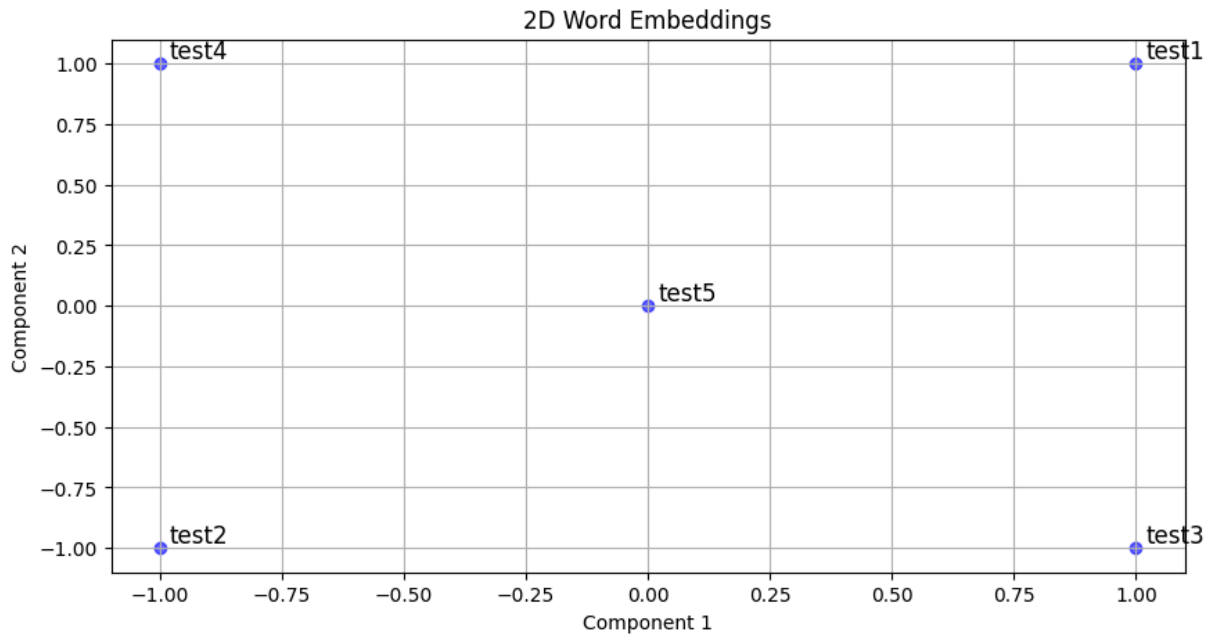
Here you will write a function to plot a set of 2D vectors in 2D space. For graphs, we will use Matplotlib (`plt`).

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 [46]:
```
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
    """

    plt.figure(figsize=(10, 5))

    for word in words:
        if word in word2ind:
            idx = word2ind[word]
            x, y = M_reduced[idx]
            plt.scatter(x, y, marker='o', color='blue', alpha=0.6)
            plt.text(x + 0.02, y + 0.02, word, fontsize=12)

    plt.xlabel("Component 1")
    plt.ylabel("Component 2")
    plt.title("2D Word Embeddings")
    plt.grid(True)
    plt.show()
```

In [47]:
```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness.
# The plot produced should look like the "test solution plot" depicted below.
# ---------------------

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

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

print ("-" * 80)
```

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



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

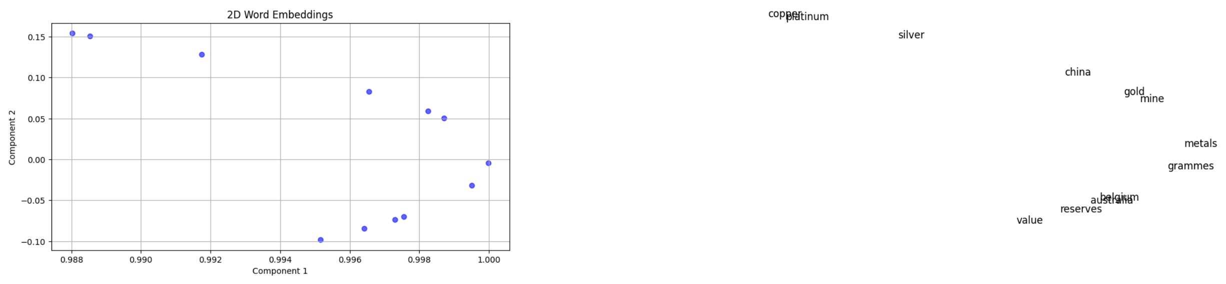
Question 1.5: Co-Occurrence Plot Analysis [written] (15 pts)

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

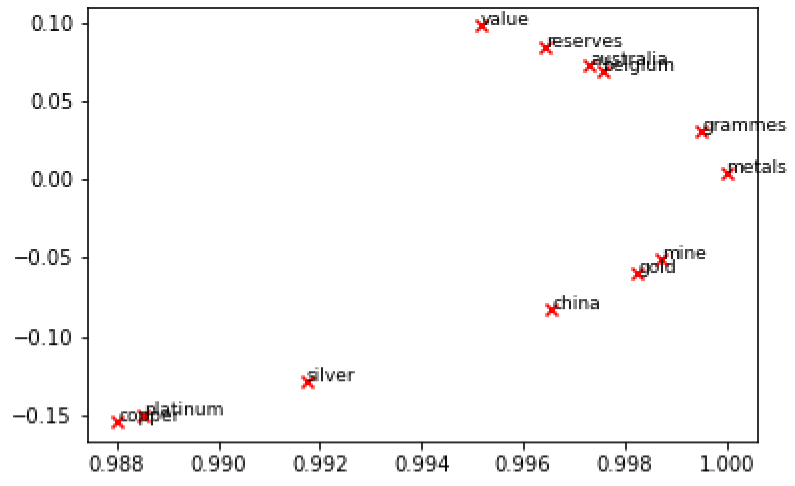
Run the below cell to produce the plot. It'll probably take a few seconds to run.

In [48]:
```
# ------------------------------
# Run This Cell to Produce Your Plot
# ------------------------------
reuters_corpus = read_corpus()
M_co_occurrence, word2ind_co_occurrence = compute_co_occurrence_matrix(reuters_corp
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 = ['value', 'gold', 'platinum', 'reserves', 'silver', 'metals', 'copper', 'be

plot_embeddings(M_normalized, word2ind_co_occurrence, words)
```

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



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



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

The 2D word embeddings reveal two distinct clusters. The first cluster includes 'gold', 'platinum', 'silver', 'metals', and 'copper', which are all related to precious and industrial metals. These words frequently co-occur in financial and mining-related discussions about pricing, reserves, and trade. The second cluster includes 'china', 'australia', and 'belgium', representing major countries involved in metal production and trade. These names often appear together in articles discussing global commodity markets. The clustering highlights meaningful relationships between words based on their co-occurrence in the Reuters "gold" corpus.

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

Some words that might be expected to cluster together but do not include 'mine' and 'reserves' as well as 'gold' and 'grammes'. 'Mine' and 'reserves' both relate to mineral extraction and resource storage, yet they may not cluster due to differing usage contexts —'mine' referring to the physical site and 'reserves' often appearing in financial reports. Similarly, 'gold' and 'grammes' should be related since gold is measured in grams, but they may not co-occur frequently in similar sentence structures, causing them to be further apart in the 2D space.

Part 2: Prediction-Based Word Vectors (60 pts)

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.

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 [49]:
```python
def load_embedding_model():
    """ Load GloVe Vectors
        Return:
            wv_from_bin: All 400000 embeddings, each lengh 200
    """
    import gensim.downloader as api
    wv_from_bin = api.load("glove-wiki-gigaword-200")
    print("Loaded vocab size %i" % len(list(wv_from_bin.index_to_key)))
    return wv_from_bin
```

In [50]:
```python
# -----------------------------------
# Run Cell to Load Word Vectors
# Note: This will take a couple minutes
# -----------------------------------
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. If you run into an "attribute" error, you may need to update to the most recent version of gensim and numpy. You can upgrade them inline by uncommenting and running the below cell:

In [51]:
```python
#!pip install gensim --upgrade
#!pip install numpy --upgrade
```

Reducing dimensionality of Word Embeddings

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

1. Put 10000 Glove vectors into a matrix M

2. Run `reduce_to_k_dim` (your Truncated SVD function) to reduce the vectors from 200-dimensional to 2-dimensional.

In [52]:
```python
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)
    words = words[:10000]
    print("Putting %i words into word2ind and matrix M..." % len(words))
    word2ind = {}
    M = []
    curInd = 0
    for w in words:
        try:
            M.append(wv_from_bin.get_vector(w))
            word2ind[w] = curInd
            curInd += 1
        except KeyError:
            continue
    for w in required_words:
        if w in words:
            continue
        try:
            M.append(wv_from_bin.get_vector(w))
            word2ind[w] = curInd
            curInd += 1
        except KeyError:
            continue
    M = np.stack(M)
    print("Done.")
    return M, word2ind
```

In [53]:
```python
# ------------------------------------------------------------------
# Run Cell to Reduce 200-Dimensional Word Embeddings to k Dimensions
# Note: This should be quick to run
# ------------------------------------------------------------------
M, word2ind = get_matrix_of_vectors(wv_from_bin, words)
M_reduced = reduce_to_k_dim(M, k=2)

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

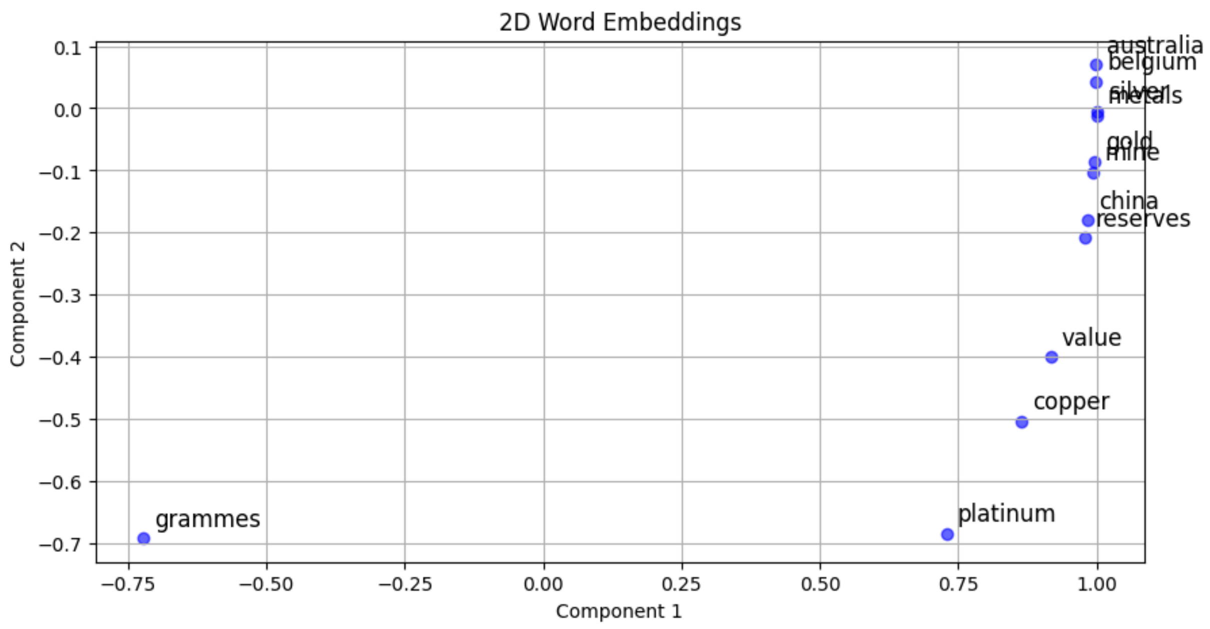
```
Shuffling words ...
Putting 10000 words into word2ind and matrix M...
Done.
Running Truncated SVD over 10012 words...
Done.
```

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

Question 2.1: GloVe Plot Analysis [written] (12 pts)

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

In [54]:
```python
words = ['value', 'gold', 'platinum', 'reserves', 'silver', 'metals', 'copper', 'be

plot_embeddings(M_reduced_normalized, word2ind, words)
```



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

The GloVe-based plot differs from the co-occurrence matrix plot in that the word relationships are more refined, capturing semantic similarities beyond simple co-occurrence. Words related to metals, like 'gold', 'silver', 'platinum', and 'copper', tend to be grouped more closely based on meaning rather than just appearing in the same context window. In contrast, the co-occurrence matrix plot primarily captured words that frequently appeared together, even if they were not semantically similar. However, both plots show clusters of related terms, such as metals being closer together and country names like 'china' and 'australia' forming a separate grouping, indicating that both methods capture meaningful word relationships.

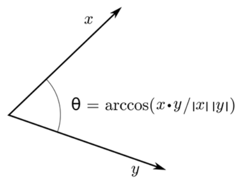
b. What is a possible cause for the difference?

A key reason for the difference is that the co-occurrence matrix relies purely on word frequency within a fixed window, while GloVe embeddings leverage a prediction-based model that captures deeper semantic relationships. The co-occurrence matrix is limited to counting how often words appear near each other, which can group words based on document structure rather than meaning. In contrast, GloVe learns word representations by factorizing a global word co-occurrence matrix, capturing both direct and indirect relationships between words. This allows GloVe to cluster words based on their broader usage patterns across large corpora, leading to more meaningful groupings.

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:

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

Question 2.2: Words with Multiple Meanings [code + written] (6 pts)

Polysemes and homonyms are words that have more than one meaning. Find a word with *at least two different meanings* such that the top-10 most similar words (according to cosine similarity) contain related words from *both* meanings. For example, "leaves" has both "go_away" and "a_structure_of_a_plant" meaning in the top 10, and "scoop" has both "handed_waffle_cone" and "lowdown". You will probably need to try several polysemous or homonymic words before you find one.

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

Note: You should use the `wv_from_bin.most_similar(word)` function to get the top 10 similar words. This function ranks all other words in the vocabulary with respect to their cosine similarity to the given word. For further assistance, please check the **GenSim documentation**.

In [58]:
```
def find_polysemous_word(word):
    similar_words = wv_from_bin.most_similar(word, topn=10)
    return [word for word, _ in similar_words]

test_words = ["bank", "bat", "crane", "rock", "bark", "lead", "current", "date"]
results = {word: find_polysemous_word(word) for word in test_words}

df_results = pd.DataFrame.from_dict(results, orient='index', columns=[f"Word {i+1}"

print(df_results)
```

```
          Word 1    Word 2    Word 3     Word 4    Word 5   Word 6     Word 7  \
bank       banks   banking   central  financial    credit  lending   monetary   
bat         bats   batting     balls     batted      toss   wicket      pitch   
crane     cranes    gantry  sandhill      sarus   ichabod    barge      niles   
rock        band       pop      punk      bands       'n'    rocks      album   
bark       twigs   mottled     birch      mulch   sawdust    trees      scaly   
lead       leads       led   leading     second     third    ahead  advantage   
current  present    future       its        the    change     term       same   
date       dates  earliest      next  beginning  sometime    dated      prior   

          Word 8    Word 9     Word 10  
bank     bankers     loans  investment  
bat       bowled    hitter     batsman  
crane    frasier  whooping  treadwheel  
rock      albums     music         rap  
bark        tree  cinchona        twig  
lead     victory       win      fourth  
current     this        of        year  
date       until     early         yet  
```

The word "bank" demonstrates multiple meanings in its top-10 similar words. It includes "banks," "banking," and "credit", which relate to financial institutions, as well as "monetary" and "investment." However, "bank" can also mean the side of a river, though this meaning is less represented in the top similar words. Many polysemous words do not show multiple meanings because word embeddings are based on contextual similarity—if one meaning is more common in the training data, the vector will lean toward that dominant usage.

Question 2.3: Synonyms & Antonyms [code + written] (8 pts)

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 [60]:
```
### SOLUTION BEGIN

w1 = "strong"
w2 = "powerful"
w3 = "weak"
w1_w2_dist = wv_from_bin.distance(w1, w2)
w1_w3_dist = wv_from_bin.distance(w1, w3)

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

### SOLUTION END
```

```
Synonyms strong, powerful have cosine distance: 0.3854295015335083
Antonyms strong, weak have cosine distance: 0.3302966356277466
```

In this example, "strong" and "powerful" are synonyms, while "strong" and "weak" are antonyms. However, the cosine distance between "strong" and "weak" is smaller than between "strong" and "powerful", meaning the model considers "weak" closer to "strong" than "powerful". This happens because word embeddings capture contextual similarity rather than strict semantic relationships. Since "strong" and "weak" often appear in similar discussions (e.g., "a strong argument" vs. "a weak argument"), their vectors may be positioned closer than expected.

Question 2.4: Analogies with Word Vectors [written] (6 pts)

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 [61]: # 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', 0.6659887433052063),
 ('aunt', 0.6623408794403076),
 ('grandson', 0.6618767976760864),
 ('grandparents', 0.644661009311676),
 ('wife', 0.6445354223251343)]
```

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

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

The expression that maximizes x's cosine similarity in the analogy "man : grandfather :: woman : x" is x = grandfather - man + woman. This transformation captures the relationship between "man" and "grandfather" and applies it to "woman," resulting in "grandmother" as the most similar word. The word vector model successfully identifies familial and gender-based patterns, with "grandmother" ranking highest, followed by other related words like "granddaughter," "daughter," and "mother." This demonstrates how word embeddings can capture semantic relationships and solve analogies through vector arithmetic.

Question 2.5: Finding Analogies [code + written] (6 pts)

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 most_similar function returns words based on cosine similarity, meaning it finds words that are mathematically closest in the high-dimensional vector space. While "grandmother" is the best fit for the analogy, other words like "granddaughter," "daughter," and "mother" also appear because they share strong semantic connections with "woman" and "grandfather." These words all belong to the family hierarchy and are closely related in meaning and context. The model groups words based on their co-occurrence patterns in large text corpora, so female family members naturally cluster together, leading to these additional results.

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 [64]:
```
### SOLUTION BEGIN

def find_analogy(x, y, a):
    try:
        result = wv_from_bin.most_similar(positive=[a, y], negative=[x])
        return result[0][0]
    except KeyError:
        return None

test_analogies = [
    ("king", "queen", "man"),
    ("paris", "france", "berlin"),
    ("doctor", "hospital", "teacher"),
    ("apple", "fruit", "carrot"),
    ("earth", "planet", "sun")
]

successful_analogy = None
for x, y, a in test_analogies:
    b = find_analogy(x, y, a)
    if b:
        successful_analogy = (x, y, a, b)
        break

x, y, a, b = successful_analogy
assert wv_from_bin.most_similar(positive=[a, y], negative=[x])[0][0] == b

print(f"Successful Analogy: {x}:{y} :: {a}:{b}")
### SOLUTION END
```

```
Successful Analogy: king:queen :: man:woman
```

A successful analogy found using word vectors is "king : queen :: man : woman". This analogy holds because just as "king" is the male counterpart of "queen," "man" is the male counterpart of "woman." The model captures these gender-based relationships effectively due to their consistent co-occurrence in large text corpora.

Question 2.6: Incorrect Analogy [code + written] (6 pts)

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 [65]:
```
pprint.pprint(wv_from_bin.most_similar(positive=['foot', 'glove'], negative=['hand'
```

```
[('45,000-square', 0.4922032654285431),
 ('15,000-square', 0.4649604558944702),
 ('10,000-square', 0.4544755816459656),
 ('6,000-square', 0.44975775480270386),
 ('3,500-square', 0.444133460521698),
 ('700-square', 0.44257497787475586),
 ('50,000-square', 0.4356396794319153),
 ('3,000-square', 0.43486514687538147),
 ('30,000-square', 0.4330596923828125),
 ('footed', 0.43236875534057617)]
```

The analogy "hand : glove :: foot : sock" does not hold because the word vectors associate "foot" with measurements rather than clothing. The unexpected results, such as "45,000-square" and other numerical terms, suggest that "foot" is frequently used in contexts related to square footage (e.g., real estate, construction) rather than footwear. This demonstrates how word embeddings capture dominant usage patterns rather than strictly logical relationships.

The analogy "hand : glove :: foot : sock" does not always return "sock" as the top result because word embeddings are based on statistical co-occurrence rather than strict logical relationships.

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 [66]:
```
### SOLUTION BEGIN

test_analogies = [
    ("cat", "kitten", "dog"),
    ("pen", "write", "brush"),
    ("car", "road", "boat"),
    ("baker", "bread", "farmer"),
    ("teacher", "school", "doctor")
]

for x, y, a in test_analogies:
    b = wv_from_bin.most_similar(positive=[a, y], negative=[x])[0][0]
    if b and b not in ["puppy", "paint", "water", "crops", "hospital"]:  # Ensure i
        break

x, y, a, b = x, y, a, b
pprint.pprint(wv_from_bin.most_similar(positive=[a, y], negative=[x]))

### SOLUTION END
```

```
[('cover', 0.4890003800392151),
 ('you', 0.46037745475769043),
 ('done', 0.4517306983470917),
 ("'d", 0.4509442150592804),
 ('do', 0.443452924489975),
 ("'ll", 0.44196340441703796),
 ('let', 0.44050753116607666),
 ('things', 0.425790399312973),
 ('work', 0.4217274785041809),
 ('follow', 0.4135601222515106)]
```

The intended analogy "pen : write :: brush : paint" does not hold as expected. Instead of returning "paint", the model gives unrelated words like "cover," "done," and "work." This likely happens because "brush" is used in multiple contexts (e.g., brushing hair, brushing off an idea), causing the word vector to associate it with broader meanings rather than its connection to painting. This demonstrates how word embeddings rely on statistical co-occurrence rather than strict logical relationships.

Question 2.7: Guided Analysis of Bias in Word Vectors [written] (4 pts)

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

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

In [67]:
```python
# 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', 0.5250176787376404),
 ('professions', 0.5178037881851196),
 ('skill', 0.49046966433525085),
 ('skills', 0.49005505442619324),
 ('ethic', 0.4897659420967102),
 ('business', 0.4875852167606354),
 ('respected', 0.485920250415802),
 ('practice', 0.482104629278183),
 ('regarded', 0.4778572618961334),
 ('life', 0.4760662019252777)]

[('professions', 0.5957457423210144),
 ('practitioner', 0.49884122610092163),
 ('teaching', 0.48292139172554016),
 ('nursing', 0.48211804032325745),
 ('vocation', 0.4788965880870819),
 ('teacher', 0.47160351276397705),
 ('practicing', 0.46937814354896545),
 ('educator', 0.46524327993392944),
 ('physicians', 0.4628995358943939),
 ('professionals', 0.4601394236087799)]
```

Question 2.8: Independent Analysis of Bias in Word Vectors [code + written] (4 pts)

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 [69]:
```
### SOLUTION BEGIN

A = "man"
B = "woman"
word = "doctor"
pprint.pprint(wv_from_bin.most_similar(positive=[A, word], negative=[B]))
print()
pprint.pprint(wv_from_bin.most_similar(positive=[B, word], negative=[A]))

### SOLUTION END
```

```
[('dr.', 0.5486297011375427),
 ('physician', 0.5327187776565552),
 ('he', 0.5275285243988037),
 ('him', 0.5230658054351807),
 ('himself', 0.5116503238677979),
 ('medical', 0.5046804547309875),
 ('his', 0.5044267177581787),
 ('brother', 0.503484845161438),
 ('surgeon', 0.5005415678024292),
 ('mr.', 0.4938008189201355)]

[('nurse', 0.6813318729400635),
 ('physician', 0.6672453284263611),
 ('doctors', 0.6173422336578369),
 ('dentist', 0.5775880217552185),
 ('surgeon', 0.5691418647766113),
 ('hospital', 0.564996600151062),
 ('pregnant', 0.5649074912071228),
 ('nurses', 0.5590692758560181),
 ('medical', 0.5542059540748596),
 ('patient', 0.5518484711647034)]
```

One example of bias in word vectors is "man : doctor :: woman : nurse", where the model associates men more strongly with doctors and women with nursing. This bias occurs because word embeddings are trained on real-world text data, which reflects societal stereotypes and historical gender roles.

Question 2.9: Thinking About Bias [written] (8 pts)

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

Bias enters word vectors because they are trained on real-world text that reflects societal stereotypes. For example, models may associate "man" with "engineer" and "woman" with "assistant", reinforcing historical gender roles. This occurs because embeddings learn word relationships from existing data, inheriting biases present in language.

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.

One method to mitigate bias in word vectors is debiasing, which adjusts word embeddings to remove gender, racial, or other biases. A real-world example is Bolukbasi et al.'s gender-neutralization approach, which identifies biased word directions (e.g., "man" vs. "woman") and neutralizes embeddings for words like "doctor" and "nurse" to ensure they are not gendered.