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

### Due 4:30pm, Tue Jan 17

Welcome to CS224N!

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

If you aren't super familiar with Python, Numpy, or Matplotlib, we recommend you check out the review session on Friday. The session will be recorded and the material will be made available on our website. 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.

```
In [7]:  # All Import Statements Defined Here
         # Note: Do not add to this list.
         # ---------------

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

         from platform import python_version
         assert int(python_version().split(".")[1]) >= 5, "Please upgrade your Python version following the instructions in \
             the README.txt 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]

         import nltk
         nltk.download('reuters') #to specify download location, optionally add the argument: download_dir='/specify/desired/path/'#可直接删去,不用重复下载
         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)
         # ---------------
```

## Word Vectors

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

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

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

Most word vector models start from the following idea:

*You shall know a word by the company it keeps (Firth, J. R. 1957:11)*

Many word vector implementations are driven by the idea that similar words, i.e., (near) synonyms, will be used in similar contexts. As a result, similar words will often be spoken or written along with a shared subset of words, i.e., contexts. By examining these contexts, we can try to develop embeddings for our words. With this intuition in mind, many "old school" approaches to constructing word vectors relied on word counts. Here we elaborate upon one of those strategies, *co-occurrence matrices* (for more information, see here or here).

### Co-Occurrence

A co-occurrence matrix counts how often things co-occur in some environment. Given some word $w_i$ occurring in the document, we consider the *context window* surrounding $w_i$. Supposing our fixed window size is $n$, then this is the $n$ preceding and $n$ subsequent words in that document, i.e. words $w_{i-n} \dots w_{i-1}$ 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 $U\_k$.


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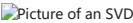
This reduced-dimensionality co-occurrence representation preserves semantic relationships between words, e.g. *doctor* and *hospital* will be closer than *doctor* and *dog*.

**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 [8]: 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_TOKEN] for f in files]
```

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

```
In [9]: reuters_corpus = read_corpus()
        pprint.pprint(reuters_corpus[:3], compact=True, width=100)
```

```
[['<START>', 'western', 'mining', 'to', 'open', 'new', 'gold', 'mine', 'in', 'australia', 'western',
  'mining', 'corp', 'holdings', 'ltd', '&', 'lt', ';', 'wmng', '.', 's', '>', '(', 'wmc', ')',
  'said', 'it', 'will', 'establish', 'a', 'new', 'joint', 'venture', 'gold', 'mine', 'in', 'the',
  'northern', 'territory', 'at', 'a', 'cost', 'of', 'about', '21', 'mln', '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', 'conventional',
  'leach', 'treatment', 'plant', ',', 'is', 'expected', 'to', 'produce', 'about', '50', ',', '000',
  'ounces', 'of', 'gold', 'in', 'its', 'first', 'year', 'of', 'production', 'from', 'mid', '-',
  '1988', '.', 'annual', 'ore', 'capacity', 'will', 'be', 'about', '750', ',', '000', 'tonnes', '.',
  '<END>'],
 ['<START>', 'belgium', 'to', 'issue', 'gold', 'warrants', ',', 'sources', 'say', 'belgium',
  'plans', 'to', 'issue', 'swiss', 'franc', 'warrants', 'to', 'buy', 'gold', ',', 'with', 'credit',
  'suisse', 'as', 'lead', 'manager', ',', 'market', 'sources', 'said', '.', 'no', 'confirmation',
  'or', 'further', 'details', 'were', 'immediately', 'available', '.', '<END>'],
 ['<START>', 'belgium', 'launches', 'bonds', 'with', 'gold', 'warrants', 'the', 'kingdom', 'of',
  'belgium', 'is', 'launching', '100', 'mln', 'swiss', 'francs', 'of', 'seven', 'year', 'notes',
  'with', 'warrants', 'attached', 'to', 'buy', 'gold', ',', 'lead', 'mananger', 'credit', 'suisse',
  'said', '.', 'the', 'notes', 'themselves', 'have', 'a', '3', '-', '3', '/', '8', 'pct', 'coupon',
  'and', 'are', 'priced', 'at', 'par', '.', 'payment', 'is', 'due', 'april', '30', ',', '1987',
  'and', 'final', 'maturity', 'april', '30', ',', '1994', '.', 'each', '50', ',', '000', 'franc',
  'note', 'carries', '15', 'warrants', '.', 'two', 'warrants', 'are', 'required', 'to', '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] (2 points)

Write a method to work out the distinct words (word types) that occur in the corpus. You can do this with `for` loops, but it's more efficient to do it with Python list comprehensions. In particular, this 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.

```python
In [10]: 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

             ### SOLUTION BEGIN
             corpus_words = [element for row in corpus for element in row]
             corpus_words = set(corpus_words)
             corpus_words = list(corpus_words)
             corpus_words.sort()
             n_corpus_words = len(corpus_words)
             ### SOLUTION END

             return corpus_words, n_corpus_words
```

```python
In [11]: # --------------------
         # 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!
--------------------------------------------------------------------------------
```

## 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.

```python
In [12]: 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 = {}

             ### SOLUTION BEGIN
             M = np.zeros((n_words,n_words),dtype = int)
             word2ind = {word:i for i,word in enumerate(words)}

             for document in corpus:
                 for i,element in enumerate(document):
                     ele_index = word2ind[element]

                     start = max(0,i-window_size)
                     end = min(len(document),i+window_size+1)

                     for j in range(start,end):
                         if i!=j:
                             context_word = document[j]
                             con_index = word2ind[context_word]
                             M[ele_index,con_index]+=1;

             ### SOLUTION END

             return M, word2ind
```

```python
In [13]: # ---------------------
         # 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., 1., 0., 0.,],
              [0., 1., 0., 0., 0., 0., 0., 0., 1.,],
              [0., 0., 0., 0., 0., 0., 0., 1., 1.,],
              [0., 0., 0., 0., 0., 0., 1., 1., 0.,],
              [1., 0., 0., 0., 0., 0., 1., 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))))
```

```python
# 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!
--------------------------------------------------------------------------------
```

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

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

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

```python
In [14]:  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]))

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

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

```python
In [15]:  # ---------------------
          # 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!
--------------------------------------------------------------------------------
```

## 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.

```python
In [16]:  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
              """

              ### SOLUTION BEGIN
              plt.figure(figsize = (10,10))

              for word in words:
                  idx = word2ind[word]
                  x,y = M_reduced[idx,0],M_reduced[idx,1]
                  plt.scatter(x,y,marker = 'o',color = 'red')
                  plt.text(x,y,word,fontsize = 20)

              plt.title('2D Word Embeddings')
              plt.xlabel('Dimension 1')
              plt.ylabel('Dimension 2')
              plt.grid(True)

              plt.show()
              ### SOLUTION END
```
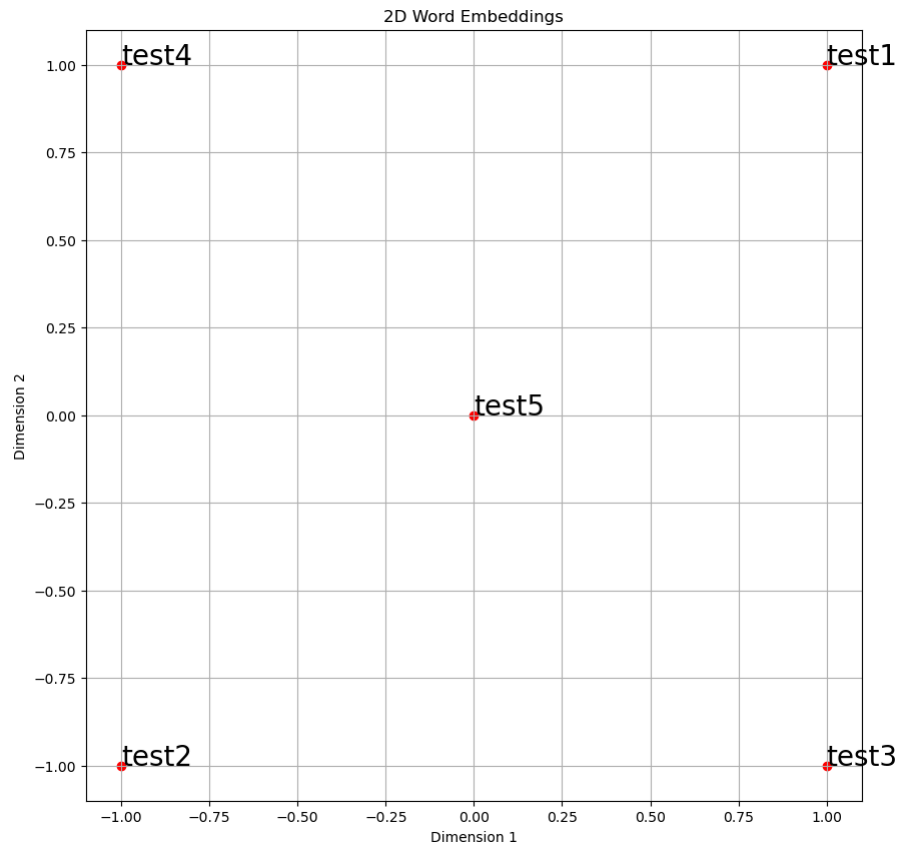
```python
In [17]:  # ---------------------
          # Run this sanity check
          # Note that this is not an exhaustive check for correctness.
          # The plot produced should look like the "test solution plot" depicted below.
          # ---------------------

          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:
```

2D Word Embeddings

--------------------------------------------------------------------------------

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

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

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

```
In [18]:  # -----------------------------
          # Run This Cell to Produce Your Plot
          # -----------------------------
          reuters_corpus = read_corpus()
          M_co_occurrence, word2ind_co_occurrence = compute_co_occurrence_matrix(reuters_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 = ['value', 'gold', 'platinum', 'reserves', 'silver', 'metals', 'copper', 'belgium', 'australia', 'china', 'grammes', "mine"]

          plot_embeddings(M_normalized, word2ind_co_occurrence, words)
```

```
Running Truncated SVD over 2830 words...
Done.
```
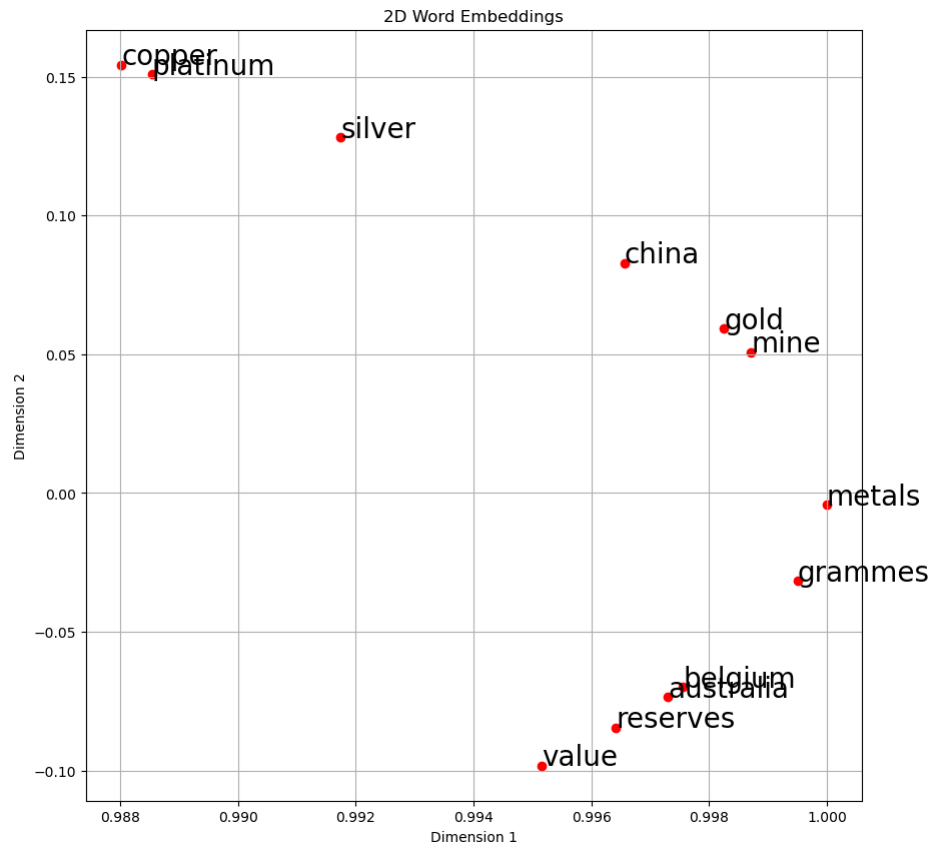
2D Word Embeddings

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

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

SOLUTION BEGIN

SOLUTION END

For example, the word "silver" and 'reserves' is belong to the same group, because they all mean some resources. The word "Platium" and "metal" is in the same group, because they all mean some metal or chemstry materials.

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

SOLUTION BEGIN

SOLUTION END

China and Australia should be clustered together, because they both are country. copper is a kind of metal but not be clustered together with "metal".

## Part 2: Prediction-Based Word Vectors (15 points)

As discussed in class, more recently prediction-based word vectors have demonstrated better performance, such as word2vec and GloVe (which also utilizes the benefit of counts). Here, we shall explore the embeddings produced by GloVe. Please revisit the class notes and lecture slides for more details on the word2vec and GloVe algorithms. If you're feeling adventurous, challenge yourself and try reading GloVe's original paper.

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

```
In [19]: from gensim.models import KeyedVectors

         def load_embedding_model():
             """ Load GloVe Vectors
                 Return:
                     wv_from_bin: All 400000 embeddings, each length 200
             """
             # 加载本地的 GloVe 文件
             glove_file = '/root/gensim-data/glove-wiki-gigaword-200.txt'
             wv_from_bin = KeyedVectors.load_word2vec_format(glove_file, binary=False)

             print("Loaded vocab size %i" % len(list(wv_from_bin.index_to_key)))
             return wv_from_bin
```

```
In [20]: # ----------------------------------
         # Run Cell to Load Word Vectors
         # Note: This will take a couple minutes
         # ----------------------------------
         # 运行加载词向量的函数
         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 [21]: #!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 [22]: 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)
             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 [23]: # ----------------------------------------------------------------
         # Run Cell to Reduce 200-Dimensional Word Embeddings to k Dimensions
         # Note: This should be quick to run
         # ----------------------------------------------------------------
```

```
M, word2ind = get_matrix_of_vectors(wv_from_bin, words)
M_reduced = reduce_to_k_dim(M, k=2)

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

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

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

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

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

```
In [24]: words = ['value', 'gold', 'platinum', 'reserves', 'silver', 'metals', 'copper', 'belgium', 'australia', 'china', 'grammes', "mine"]

plot_embeddings(M_reduced_normalized, word2ind, words)
```
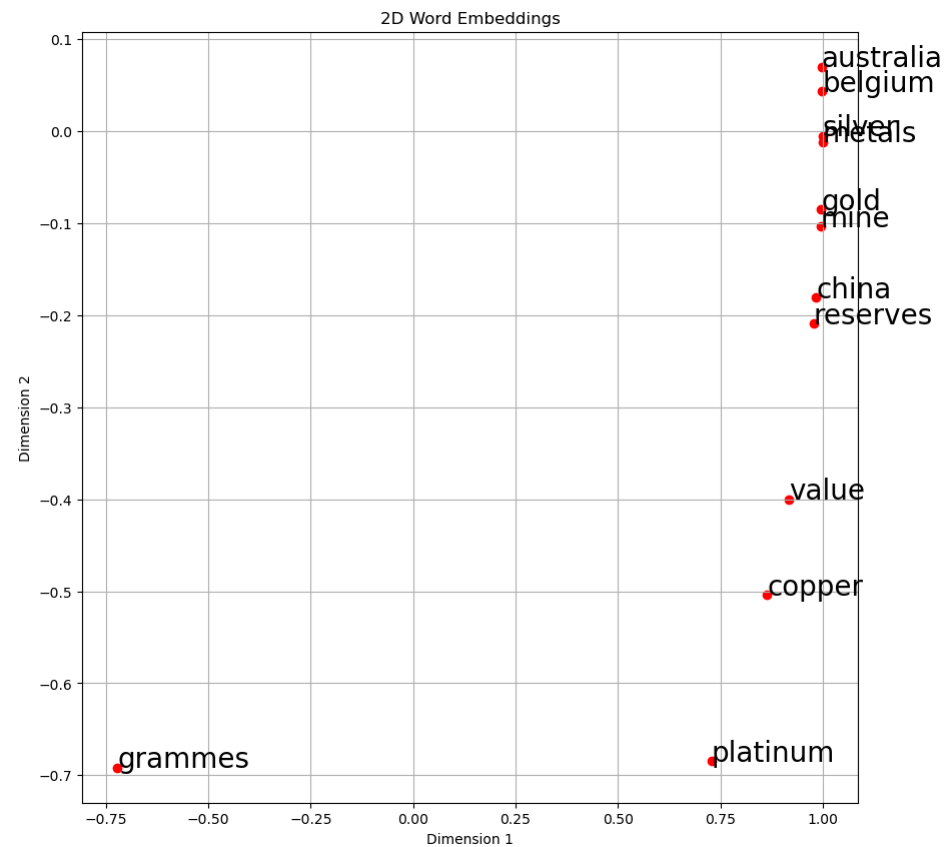


a. What is one way the plot is different from the one generated earlier from the co-occurrence matrix? What is one way it's similar?

The reduced features are centralized on the x-axis but distributed on the y-axis. But the co-occurence matrix's embedding vector is distributed on both two features.

SOLUTION BEGIN

SOLUTION END

b. What is a possible cause for the difference?

Because the Glove is trained on a lager dataset and use the deep learning method, the x-axis feature depart the word "grammes" and others and the y-axis feature contains more semantic information. The word group on the "right" have some family feature on x but have different features on the y-axis because they means differently in semantic representation.

SOLUTION BEGIN

SOLUTION END

## Cosine Similarity

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

We can think of n-dimensional vectors as points in n-dimensional space. If we take this perspective L1 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:


No description has been provided for this image

Instead of computing the actual angle, we can leave the similarity in terms of $similarity = cos(\Theta)$. Formally the Cosine Similarity $s$ between two vectors $p$ and $q$ is defined as:

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

## Question 2.2: Words with Multiple Meanings (1.5 points) [code + written]

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

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

**Note**: You should use the `wv_from_bin.most_similar(word)` function to get the top 10 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 [26]: ### SOLUTION BEGIN
         word = 'close'
         similar_words = wv_from_bin.most_similar(word,topn=10)

         for similar_word,similarity in similar_words:
             print(f"{similar_word}: {similarity}")

         ### SOLUTION END
```

```
up: 0.6898429989814758
closed: 0.6854836940765381
down: 0.6749923229217529
while: 0.6741016507148743
just: 0.6566178798675537
but: 0.6522998213768005
closing: 0.6294093728065491
point: 0.6282009482383728
far: 0.6234282851219177
time: 0.620608389377594
```

SOLUTION BEGIN

SOLUTION END

For convience, we will explain the word close by Chinese, meaning 1 means "关闭", and 2 means "近". So, the meaning 1 similar word contains: closed, the meaning 2 similar word contains: closing, up, down, far, time, while and so on. The word mostly shows similar words for only one meaning because the GloVe model is likely trained based on context and may not be sufficient to distinguish all possible meanings of the word. Additionally, some meanings may appear less frequently in the training data, so the model is unable to learn those meanings.

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

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

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

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

You should use the the `wv_from_bin.distance(w1, w2)` function here in order to compute the cosine distance between two words. Please see the **GenSim documentation** for further assistance.

```
In [27]: ### SOLUTION BEGIN

         w1 = "like"
```

```
w2 = "love"
w3 = "dislike"
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 like, love have cosine distance: 0.4338703751564026
Antonyms like, dislike have cosine distance: 0.7098513841629028
```

SOLUTION BEGIN

SOLUTION END

Distribution of Semantic Space and Contextual Overlap: Word embedding models (like GloVe) are trained based on the co-occurrence relationships of words in large-scale corpora. Although the words "like" and "dislike" are antonyms, they often appear in similar contexts in everyday usage. For example, both "I like ice cream" and "I dislike ice cream" describe a statement about personal preference. Therefore, in the word vector space, "like" and "dislike" might be relatively close due to the overlapping contexts in which they co-occur in the corpus. Limitations of Word Embedding Models: Word embedding models cannot capture the precise logical meaning of words; they generate vectors based only on the co-occurrence frequency of words. As a result, even though "like" and "love" are closer in terms of emotion, they may be slightly further apart in the vector space due to contextual differences. In contrast, even though "like" and "dislike" are antonyms, they may be closer than expected because they are used in similar contexts. Impact of Polysemy: Both "like" and "dislike" may have multiple meanings, and the GloVe model does not differentiate these meanings during training. Therefore, even though they are antonyms, their multiple meanings in the word vector space might cause them to be distributed closer together.

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

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

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

In the cell below, we show you how to use word vectors to find x using the `most_similar` function from the **GenSim documentation**. The function finds words that are most similar to the words in the `positive` list and most dissimilar from the words in the `negative` list (while omitting the input words, which are often the most similar; see this paper). The answer to the analogy will have the highest cosine similarity (largest returned numerical value).

```
In [28]:  # Run this cell to answer the analogy -- man : grandfather :: woman : x
          pprint.pprint(wv_from_bin.most_similar(positive=['woman', 'grandfather'], negative=['man']))
```

```
[('grandmother', 0.7608445882797241),
 ('granddaughter', 0.7200807929039001),
 ('daughter', 0.7168301939964294),
 ('mother', 0.7151536345481873),
 ('niece', 0.7005683183670044),
 ('father', 0.6659886837005615),
 ('aunt', 0.6623409390449524),
 ('grandson', 0.6618767380714417),
 ('grandparents', 0.6446609497070312),
 ('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, to what expression are we maximizing $x$'s cosine similarity?

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

SOLUTION BEGIN

SOLUTION END

"We have ( m - w = g - x ) then ( x = g - m + w ), the answer consider the gender reason."

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

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

SOLUTION BEGIN

SOLUTION END

The reason why the most_similar function returns words like "granddaughter," "daughter," or "mother" is because word embedding models (such as GloVe) build word vectors based on co-occurrence relationships in the corpus. These models do not understand the precise logic or familial relationships between words; instead, they capture semantic similarities and associations between words in a high-dimensional space. Here are several reasons why these words are considered similar: 1. Similarity in Family Relations: Words like "grandmother," "granddaughter," "daughter," and "mother" all represent female roles within a family. Word embedding models position these words similarly because they frequently appear in similar contexts within the corpus. So, even though "grandmother" is the most fitting word for the analogy, other words like "granddaughter" or "daughter" also have high similarity scores due to their similar familial roles. 2. Semantic Similarity: When building word vectors, models prioritize semantic similarity rather than specific logical relationships. Therefore, words related to "grandmother" may include all vocabulary associated with family or female roles, forming a semantic cluster in the model. This clustering causes the model to perceive "granddaughter," "daughter," or "mother" as being very close to "grandmother." 3. Model Generalization: Since the model constructs word vectors by observing the frequency of word co-occurrence, it tends to cluster words with similar contexts together. Thus, even though these words may differ in familial relationships, the model still represents them as similar vectors because they may frequently co-occur in large amounts of text. In summary, the most_similar function returns these words because the word embedding model clusters words representing familial and female roles together, reflecting their co-occurrence patterns in the large-scale corpus rather than precise logical relationships. This generalization sometimes causes the word embedding model to be unable to accurately pick the most suitable word for the analogy.

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!

```
### SOLUTION BEGIN

x, y, a, b = 'woman', 'man', 'actress', 'actor'
assert wv_from_bin.most_similar(positive=[a, y], negative=[x])[0][0] == b

### SOLUTION END
```

SOLUTION BEGIN

SOLUTION END

Because the difference between "actor" and "actress" is only gender and the difference between "man" and "woman" is only gender too.

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

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

In [30]:
```
pprint.pprint(wv_from_bin.most_similar(positive=['foot', 'glove'], negative=['hand']))
```

```
[('45,000-square', 0.4922032058238983),
 ('15,000-square', 0.4649604260921478),
 ('10,000-square', 0.4544755816459656),
 ('6,000-square', 0.44975775480270386),
 ('3,500-square', 0.4441334009170532),
 ('700-square', 0.44257497787475586),
 ('50,000-square', 0.4356396794319153),
 ('3,000-square', 0.43486520648002625),
 ('30,000-square', 0.4330596923828125),
 ('footed', 0.43236875534057617)]
```

SOLUTION BEGIN

SOLUTION END

The reason this analogy yields unexpected results is likely because the word embedding model (such as GloVe) does not accurately capture the specific relationship between "glove" and "sock." Instead, it determines the similarity between words based on co-occurrence frequencies and context. In this case, the model returns words that include "square" or similar terms because, in the corpus, "foot" might frequently co-occur with these words in contexts related to describing area (e.g., "square footage"). 1. Corpus Bias: The word embedding model is trained on a large-scale corpus. If, in the corpus, "foot" more frequently appears in contexts describing area or architecture rather than in contexts related to clothing, the model will be more inclined to associate "foot" with these contexts, overlooking "sock" as a more appropriate word. 2. Contextual Overlap: While "hand" and "glove" have a clear association in context, "foot" and "sock" might not have as strong a co-occurrence frequency in the corpus. As a result, the model may not find a strong connection between "foot" and "sock," leading it to return other words based on the frequent contexts where "foot" appears. 3. Impact of Polysemy: The word "foot" has multiple meanings, such as referring to a unit of length (foot) or a body part. If the usage frequency of "foot" as a length unit is higher in the training data than as a body part, the model might lean towards this interpretation, returning words related to area or building measurements. In summary, this analogy produces unexpected results because the model tends to determine similarity based on frequently co-occurring contexts rather than accurately capturing the specific association between "foot" and "sock."

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 [31]:
```
### SOLUTION BEGIN

x, y, a, b = 'shoes', 'hat', 'up', 'down'
pprint.pprint(wv_from_bin.most_similar(positive=[a, y], negative=[x]))

### SOLUTION END
```

```
[('out', 0.5788853168487549),
 ('just', 0.5641639828681946),
 ('put', 0.5537455081939697),
 ('off', 0.5531725287437439),
 ('second', 0.5463935732841492),
 ('another', 0.5417174696922302),
 ('trick', 0.5410613417625427),
 ('down', 0.5368677973747253),
 ('set', 0.5307223796844482),
 ('got', 0.527569055557251)]
```

SOLUTION BEGIN

SOLUTION END

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

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

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

In [32]:
```
# 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', 0.5250177383422852),
 ('professions', 0.5178037881851196),
 ('skill', 0.4904696047306061),
 ('skills', 0.49005505442619324),
 ('ethic', 0.4897659420967102),
 ('business', 0.4875850975513458),
 ('respected', 0.485920250415802),
 ('practice', 0.482104629278183),
 ('regarded', 0.47785720229148865),
 ('life', 0.4760662913322449)]

[('professions', 0.5957458019256592),
 ('practitioner', 0.4988412857055664),
 ('teaching', 0.48292145133018494),
 ('nursing', 0.4821180701257983),
 ('vocation', 0.4788966476917267),
 ('teacher', 0.47160351276397705),
 ('practicing', 0.46937811374664307),
 ('educator', 0.46524322032928467),
 ('physicians', 0.4628995358943939),
 ('professionals', 0.46013936400413513)]
```

SOLUTION BEGIN

SOLUTION END

From the two lists, several noticeable differences can be observed: 1. Words related to "woman" and "profession": - The words include "reputation," "professions," "skill," "ethic," etc. - These terms are more neutral or relate to professional ethics and skills, generally describing abstract or broader professional characteristics. 2. Words related to "man" and "profession": - The words include "practitioner," "teaching," "nursing," "vocation," "physicians," etc. - These terms are more specific, referring to concrete professional roles and job types, such as "teaching" or "nursing." Reflection of Gender Bias The reflection of gender bias in the word embedding model is that, although "profession" is a neutral term, the words associated with "woman" lean more towards abstract attributes, such as "skills" or "reputation," while the words associated with "man" are more concrete, linked to actual professions and job roles. The appearance of terms like "nursing" may indicate the traditional view of these roles as female occupations, showing that the model contains gender stereotypes. This bias suggests that the word embedding model forms implicit associations based on the gender distribution and social stereotypes present in the training corpus. Such bias could further reinforce gender inequality in real-world applications. For example, an automated hiring system might make unfair decisions based on these inherent biases.

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

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

```
In [33]:  ### SOLUTION BEGIN

          A = 'black'
          B = 'white'
          word = 'job'
          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
```

```
[('jobs', 0.6843502521514893),
 ('work', 0.6047077775001526),
 ('hiring', 0.5963166356086731),
 ('doing', 0.5790296196937561),
 ('better', 0.5735870003700256),
 ('good', 0.5692481398582458),
 ('employment', 0.568367600440979),
 ('because', 0.5603122115135193),
 ('working', 0.5544289946556091),
 ('even', 0.5500229001045227)]

[('jobs', 0.6257917881011963),
 ('office', 0.5849531888961792),
 ('administration', 0.583185613155365),
 ('doing', 0.5742583870887756),
 ('done', 0.569618821144104),
 ('working', 0.5639634728431702),
 ('clinton', 0.5559402108192444),
 ('work', 0.5546744465827942),
 ('staff', 0.5506452322006226),
 ('hiring', 0.5501008033752441)]
```

SOLUTION BEGIN

SOLUTION END

From the results, it can be observed that when we use the most_similar function to find analogies with the word "job," and use "black" as the positive word and "white" as the negative word, the returned words include "hiring," "employment," and others related to the labor market or job opportunities. However, when "white" is the positive word and "black" is the negative word, the returned words include "office," "administration," and others, which are more associated with office work or administrative roles. Explanation of the Bias This indicates the presence of implicit racial bias within the model. In the first case, the

model tends to associate "black" with labor-intensive jobs or the employment market, while in the second case, it associates "white" with higher-level jobs (such as office or administrative roles). This bias may stem from the historical or societal stereotypes present in the corpus used to train the model, leading it to reinforce these prejudices when associating these words.

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

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

Bias in word vectors typically arises from historical or social stereotypes present in the training data. Word embedding models (like GloVe or Word2Vec) are trained on large text corpora that reflect real-world human language usage. If these corpora contain gender, racial, or other social biases, the model learns and replicates these biases during training. Causes of Bias 1. Bias in Data Sources: Training data often comes from sources such as the internet, news articles, books, or social media, which inevitably contain societal biases. For example, when describing certain professions, the text may favor specific gender pronouns (e.g., "male doctor" and "female nurse"), or when discussing crime, there might be biased descriptions of certain races. 2. Impact of Co-occurrence Relationships: Word embedding models build vector representations based on the co-occurrence frequency of words. If certain words frequently appear alongside specific gender, race, or other characteristic-related words, the model associates them together, resulting in biased representations in the vector space. Real-world Example In practice, if a word embedding model is trained using text from social media, that text might contain many gender role stereotypes. For instance, when describing household roles, "woman" might frequently co-occur with words like "cooking" or "cleaning," while "man" might more often appear with words like "leader" or "work." This pattern leads the model to reinforce and replicate these gender stereotypes when inferring word relationships, potentially causing unfair assessments of female candidates in real-world applications like automated hiring systems.

SOLUTION BEGIN

SOLUTION END

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

One method to mitigate bias exhibited by word vectors is to use word embedding debiasing techniques. A common approach is to remove the gender or other characteristic dimension from the word vectors through projection. This ensures that the model is not influenced by gender or other biases when processing these words, leading to fairer representations. Method: Debiasing Word Embeddings A typical method is the debiasing algorithm proposed by Bolukbasi et al. The main steps of this algorithm include: 1. Identifying the Bias Direction: First, extract the gender direction from pairs of words that contain gender bias (e.g., "he - she," "man - woman"). These word pairs help determine the dimension in the vector space that represents gender. 2. Neutralization and Equalization: - Neutralization: For words that should not have a gender attribute (e.g., "doctor" or "programmer"), project their word vectors onto the orthogonal space of the gender direction, making them gender-neutral. - Equalization: For words that have gendered counterparts (e.g., "prince - princess"), adjust their vectors so that they are equidistant along the gender dimension while maintaining the distance between them. Real-world Example In an automated hiring system, if the word embedding model contains gender bias, certain job positions (such as "engineer" or "nurse") might be strongly associated with a specific gender. This could lead the system to unfairly favor one gender when recommending or screening candidates. By applying the debiasing algorithm mentioned above, it is possible to neutralize the gender-biased vectors for these roles without changing the primary semantic relationships between words. This ensures that words like "engineer" are no longer strongly associated with either "male" or "female," helping the hiring system to evaluate candidates fairly based on their actual skills and experience rather than gender, thus reducing the likelihood of gender discrimination.

SOLUTION BEGIN

SOLUTION END

# Submission Instructions

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