exploring_word_vectors

January 12, 2024

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

1.0.1 Due 3:15pm, Tue Jan 16 2024

Welcome to CS224N!

Before you start, make sure you read the README.md in the same directory as this notebook for important setup information. 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.

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

import sys
assert sys.version_info[0] == 3
assert sys.version_info[1] >= 8

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

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

import nltk
nltk.download('reuters') #to specify download location, optionally add the␣
↪argument: download_dir='/specify/desired/path/'
```

```python
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\ckn79\AppData\Roaming\nltk_data…
[nltk_data]   Package reuters is already up-to-date!
```

1.1 Word Vectors

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

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

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

Most word vector models start from the following idea:

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

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

1.2.1 Co-Occurrence

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

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

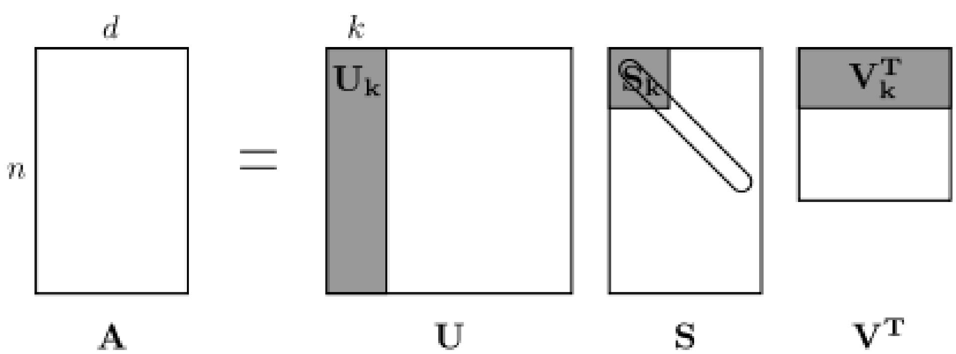
Document 1: "all that glitters is not gold"

Document 2: "all is well that ends well"

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

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.

1.2.2 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.

[2]:
```python
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….

[3]:
```python
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>']]
```

1.2.3 Question 1.1: Implement `distinct_words` [code] (2 points)

Write a method to work out the distinct words (word types) that occur in the corpus.

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

Your returned `corpus_words` should be sorted. You can use python's `sorted` function for this.

You may find it useful to use Python sets to remove duplicate words.

[4]:
```
def distinct_words(corpus):
    """ Determine a list of distinct words for the corpus.
        Params:
            corpus (list of list of strings): corpus of documents
        Return:
            corpus_words (list of strings): sorted list of distinct words␣
↪across the corpus
            n_corpus_words (integer): number of distinct words across the corpus
    """
    corpus_words = []
    n_corpus_words = -1

    # ------------------
    # Write your implementation here.
    corpus_words = sorted(set([word for row in corpus for word in row])) #␣
↪flatten using list comprehension
    n_corpus_words = len(corpus_words)

    # ------------------

    return corpus_words, n_corpus_words
```

[5]:
```
# ---------------------
# Run this sanity check
# Note that this not an exhaustive check for correctness.
# ---------------------

# Define toy corpus
test_corpus = ["{} All that glitters isn't gold {}".format(START_TOKEN,␣
↪END_TOKEN).split(" "), "{} All's well that ends well {}".format(START_TOKEN,␣
↪END_TOKEN).split(" ")]
test_corpus_words, num_corpus_words = distinct_words(test_corpus)

# Correct answers
ans_test_corpus_words = sorted([START_TOKEN, "All", "ends", "that", "gold",␣
↪"All's", "glitters", "isn't", "well", END_TOKEN])
ans_num_corpus_words = len(ans_test_corpus_words)

# Test correct number of words
```

```
assert(num_corpus_words == ans_num_corpus_words), "Incorrect number of distinct␣
↪words. Correct: {}. Yours: {}".format(ans_num_corpus_words, num_corpus_words)

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

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

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

1.2.4 Question 1.2: Implement `compute_co_occurrence_matrix` [code] (3 points)

Write a method that constructs a co-occurrence matrix for a certain window-size n (with a default of 4), considering words n before and n after the word in the center of the window. Here, we start to use `numpy (np)` to represent vectors, matrices, and tensors. If you're not familiar with NumPy, there's a NumPy tutorial in the second half of this cs231n Python NumPy tutorial.

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

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

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

        Params:
            corpus (list of list of strings): corpus of documents
            window_size (int): size of context window
        Return:
            M (a symmetric numpy matrix of shape (number of unique words in the␣
↪corpus , number of unique words in the corpus)):
                Co-occurence matrix of word counts.
                The ordering of the words in the rows/columns should be the␣
↪same as the ordering of the words given by the distinct_words function.
```

```
            word2ind (dict): dictionary that maps word to index (i.e. row/
↪column number) for matrix M.
    """
    words, n_words = distinct_words(corpus)
    M = None
    word2ind = {}

    # ------------------
    # Write your implementation here.
    M = np.zeros((n_words, n_words))
    word2ind = dict(zip(words, range(n_words)))

    for doc in corpus:
        for i, word in enumerate(doc):
            start = max(0, i - window_size) # find starting index for window
            end = min(i + window_size + 1, len(doc)) # find end index for window
            # iterate through each word in window
            for j in range(start, end):
                if j != i:
                    M[word2ind[word], word2ind[doc[j]]] += 1

    # ------------------

    return M, word2ind
```

[7]:
```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness.
# ---------------------

# Define toy corpus and get student's co-occurrence matrix
test_corpus = ["{} All that glitters isn't gold {}".format(START_TOKEN,␣
↪END_TOKEN).split(" "), "{} All's well that ends well {}".format(START_TOKEN,␣
↪END_TOKEN).split(" ")]
M_test, word2ind_test = compute_co_occurrence_matrix(test_corpus, window_size=1)

# Correct M and word2ind
M_test_ans = np.array(
    [[0., 0., 0., 0., 0., 0., 1., 0., 0., 1.,],
     [0., 0., 1., 1., 0., 0., 0., 0., 0., 0.,],
     [0., 1., 0., 0., 0., 0., 0., 0., 1., 0.,],
     [0., 1., 0., 0., 0., 0., 0., 0., 0., 1.,],
     [0., 0., 0., 0., 0., 0., 0., 0., 1., 1.,],
     [0., 0., 0., 0., 0., 0., 0., 1., 1., 0.,],
     [1., 0., 0., 0., 0., 0., 0., 1., 0., 0.,],
     [0., 0., 0., 0., 0., 1., 1., 0., 0., 0.,],
     [0., 0., 1., 0., 1., 1., 0., 0., 0., 1.,],
```

```
    [1., 0., 0., 1., 1., 0., 0., 0., 1., 0.,]]
)
ans_test_corpus_words = sorted([START_TOKEN, "All", "ends", "that", "gold",␣
↪"All's", "glitters", "isn't", "well", END_TOKEN])
word2ind_ans = dict(zip(ans_test_corpus_words,␣
↪range(len(ans_test_corpus_words))))

# Test correct word2ind
assert (word2ind_ans == word2ind_test), "Your word2ind is incorrect:\nCorrect:␣
↪{}\nYours: {}".format(word2ind_ans, word2ind_test)

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

# Test correct M values
for w1 in word2ind_ans.keys():
    idx1 = word2ind_ans[w1]
    for w2 in word2ind_ans.keys():
        idx2 = word2ind_ans[w2]
        student = M_test[idx1, idx2]
        correct = M_test_ans[idx1, idx2]
        if student != correct:
            print("Correct M:")
            print(M_test_ans)
            print("Your M: ")
            print(M_test)
            raise AssertionError("Incorrect count at index ({}, {})=({}, {}) in␣
↪matrix M. Yours has {} but should have {}.".format(idx1, idx2, w1, w2,␣
↪student, correct))

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

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

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

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

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

[8]:
```
def reduce_to_k_dim(M, k=2):
    """ Reduce a co-occurence count matrix of dimensionality (num_corpus_words,␣
 ↪num_corpus_words)
        to a matrix of dimensionality (num_corpus_words, k) using the following␣
 ↪SVD function from Scikit-Learn:
            - http://scikit-learn.org/stable/modules/generated/sklearn.
 ↪decomposition.TruncatedSVD.html

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

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

    # ------------------

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

[9]:
```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness
# In fact we only check that your M_reduced has the right dimensions.
# ---------------------

# Define toy corpus and run student code
test_corpus = ["{} All that glitters isn't gold {}".format(START_TOKEN,␣
 ↪END_TOKEN).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)
```

```python
# Test proper dimensions
assert (M_test_reduced.shape[0] == 10), "M_reduced has {} rows; should have {}".
 ↪format(M_test_reduced.shape[0], 10)
assert (M_test_reduced.shape[1] == 2), "M_reduced has {} columns; should have␣
 ↪{}".format(M_test_reduced.shape[1], 2)

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

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

1.2.6 Question 1.4: Implement `plot_embeddings` [code] (1 point)

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

For this example, you may find it useful to adapt this code. In the future, a good way to make a plot is to look at the Matplotlib gallery, find a plot that looks somewhat like what you want, and adapt the code they give.

[10]:
```python
def plot_embeddings(M_reduced, word2ind, words):
    """ Plot in a scatterplot the embeddings of the words specified in the list␣
 ↪"words".
        NOTE: do not plot all the words listed in M_reduced / word2ind.
        Include a label next to each point.

        Params:
            M_reduced (numpy matrix of shape (number of unique words in the␣
 ↪corpus , 2)): matrix of 2-dimensioal word embeddings
            word2ind (dict): dictionary that maps word to indices for matrix M
            words (list of strings): words whose embeddings we want to visualize
    """

    # ------------------
    # Write your implementation here.
    for word in words:
        row = word2ind[word]
        x = M_reduced[row, 0]
        y = M_reduced[row, 1]
        plt.scatter(x, y, marker='x', color='red')
        plt.text(x, y, word, fontsize=9)
```

```
    plt.show()

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

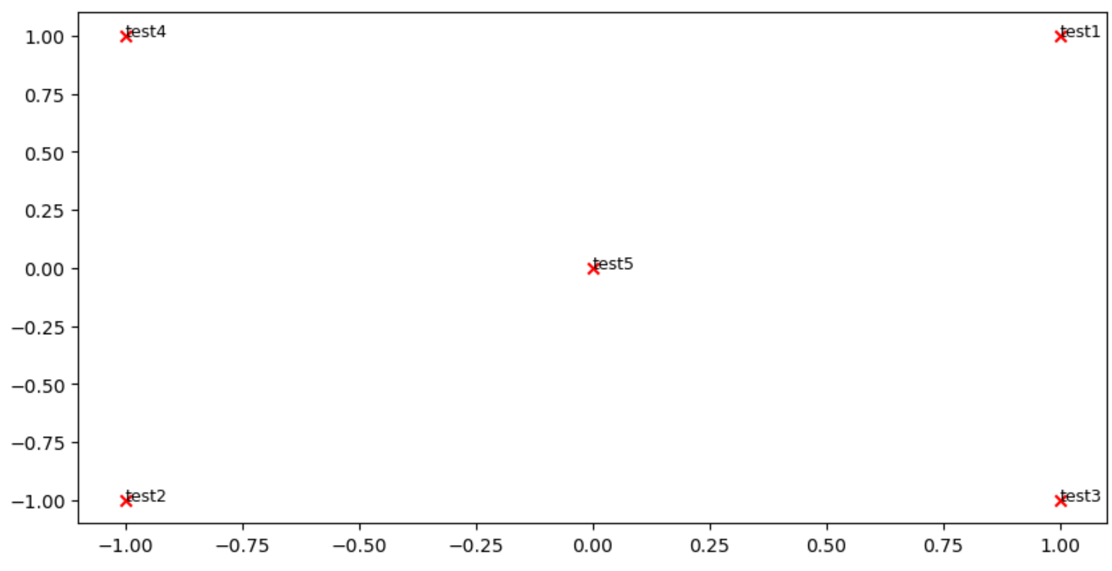
[11]:
```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness.
# The plot produced should look like the included file question_1.4_test.png
# ---------------------

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

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

print ("-" * 80)
```

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




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

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

Now we will put together all the parts you have written! We will compute the co-occurrence matrix with fixed window of 4 (the default window size), over the 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.

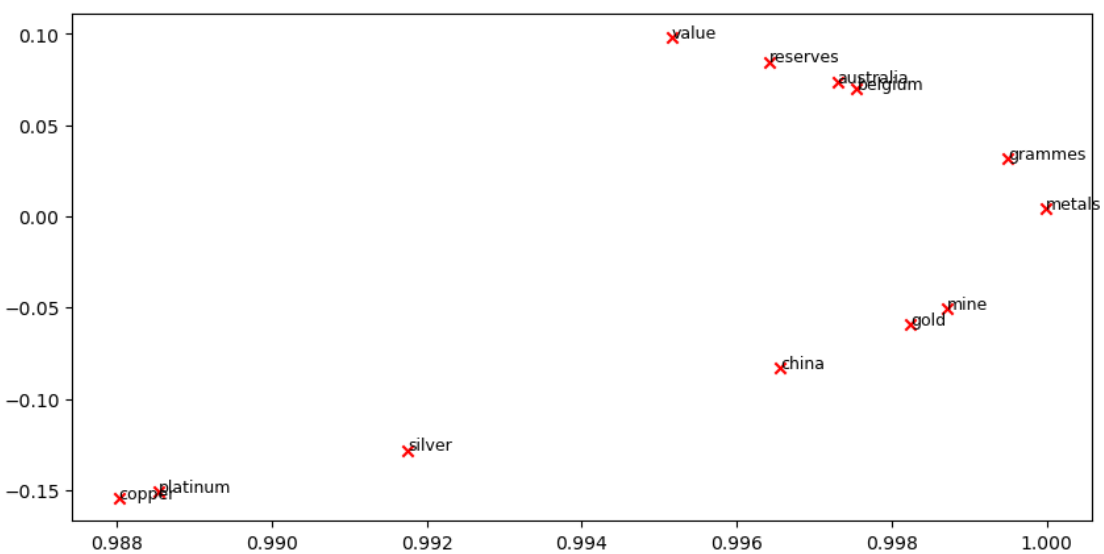
[12]:
```
# ----------------------------
# Run This Cell to Produce Your Plot
# -----------------------------
reuters_corpus = read_corpus()
M_co_occurrence, word2ind_co_occurrence =␣
 ↪compute_co_occurrence_matrix(reuters_corpus)
M_reduced_co_occurrence = reduce_to_k_dim(M_co_occurrence, k=2)

# 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.
```



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

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

Group 1 containing copper, platinum, and silver as they are types of metals and most likely were part of a list in the sentence structure. Group 2 containing australia and belgium has the closest clustering of the words because both are countries and like Group 1 were used near each other in the sentence. Group 3 with gold and mine because gold can be closely associated with mines or mining operations and the usage of "gold" would provide contextual clues in what was in the mine.

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

I expected gold to cluster closer to the other metals or the word "metals" being set between Group 1 and Group 3 making it singular cluster. Additionally, I think china should be clustered with australia and belgium being that all three are countries.

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

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

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

[13]:
```
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
wv_from_bin = load_embedding_model()
```

```
Loaded vocab size 400000
```

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

1.3.1 Reducing dimensionality of Word Embeddings

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

[14]:
```
def get_matrix_of_vectors(wv_from_bin, required_words):
    """ Put the GloVe vectors into a matrix M.
        Param:
            wv_from_bin: KeyedVectors object; the 400000 GloVe vectors loaded␣
↪from 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
```

[15]:
```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.
```

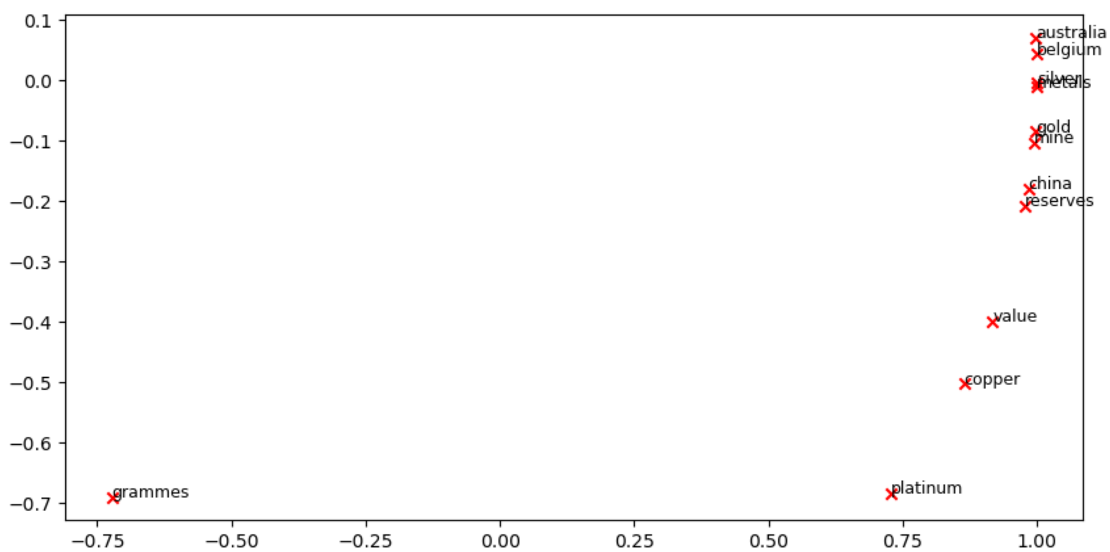
Note: If you are receiving out of memory issues on your local machine, try closing other applications to free more memory on your device. You may want to try restarting your machine so that you can free up extra memory. Then immediately run the jupyter notebook and see if you can load the word vectors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

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

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

[16]:
```python
words = ['value', 'gold', 'platinum', 'reserves', 'silver', 'metals', 'copper',␣
↪'belgium', 'australia', 'china', 'grammes', "mine"]

plot_embeddings(M_reduced_normalized, word2ind, words)
```



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

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

A difference of the GloVe plot is the proximity of words to new clusters. Platinum, copper, and silver are closer to the gold/mine cluster. There are some clusters that remain similar to the earlier plot such as australia and belgium.

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

GloVe uses both global and local context of each word to create a co-occurrence matrix while the earlier co-occurrence matrix was made by counting the number of times two words appeared within a window of the given corpus. This difference allows the GloVe plot to cluster words that provide more/different meaning to other words (i.e. value being closer to words that intrinsically provide value like platinum and copper) opposed to clustering by high co-occurence count.

1.3.3 Cosine Similarity

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

We can think of n-dimensional vectors as points in n-dimensional space. If we take this perspective L1 and L2 Distances help quantify the amount of space "we must travel" to get between these two points. Another approach is to examine the angle between two vectors. From trigonometry we know that:

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

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

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

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

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

Note: You should use the `wv_from_bin.most_similar(word)` function to get the top 10 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**.

[17]:
```
# ------------------
# Write your implementation here.
wv_from_bin.most_similar("light")

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

[17]:
```
[('bright', 0.6242774724960327),
 ('dark', 0.6141002178192139),
 ('lights', 0.6013951897621155),
 ('lighter', 0.558175265789032),
 ('heavy', 0.54083651304245),
 ('sunlight', 0.5362918972969055),
 ('blue', 0.5349379777908325),
 ('colored', 0.5282375812530518),
 ('sky', 0.5239453315734863),
 ('color', 0.513929009437561)]
```

My discovered word is "light"(weight of an object/visible energy) and the similar words given pertaining to each different meaning are "heavy, lighter" and "bright, dark, sunlight". Some words don't work because one of its meanings has higher frequency of usage than the other. For example, a "fan" in terms of the air-blowing device is less used now than a person who follows a celebrity.

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

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

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

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

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

[18]:
```
# ------------------
# Write your implementation here.
w1 = "light"
w2 = "airy"
w3 = "heavy"
synonym = wv_from_bin.distance(w1, w2)
antonym = wv_from_bin.distance(w1, w3)
print("Antonym: {}, Synonym: {}, Antonym < Synonym: {}".format(antonym,␣
 ↪synonym, antonym<synonym))
# ------------------
```

```
Antonym: 0.4591635465621948, Synonym: 0.7668143808841705, Antonym < Synonym:
True
```

"Light" is closer to "heavy" as "heavy" can be used to compare two different objects, therefore increasing similarity. Additionally, the usage of "airy" is less frequent than "light" making the similarity lower which in turn increases distance.

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

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

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

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

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

```
[('grandmother', 0.7608445286750793),
 ('granddaughter', 0.7200808525085449),
```

```
('daughter', 0.7168302536010742),
('mother', 0.7151536345481873),
('niece', 0.7005682587623596),
('father', 0.6659888029098511),
('aunt', 0.6623408794403076),
('grandson', 0.6618767380714417),
('grandparents', 0.6446609497070312),
('wife', 0.6445354223251343)]
```

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

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

$x = g - m + w$

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

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

An explanation for words like "granddaughter", "daughter", and "mother" is that they are closely similar to "grandmother" in terms of family relation association.

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!

[20]:
```python
# For example: x, y, a, b = ("", "", "", "")
# ------------------
# Write your implementation here.
x, y, a, b = ("bird", "air", "fish", "water")

# ------------------

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

bird:air::fish:water

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

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

[21]:
```python
pprint.pprint(wv_from_bin.most_similar(positive=['foot', 'glove'],␣
↪negative=['hand']))
```

```
[('45,000-square', 0.4922032058238983),
 ('15,000-square', 0.4649604558944702),
 ('10,000-square', 0.4544764125061035),
 ('6,000-square', 0.44975772500038147),
 ('3,500-square', 0.4441334009170532),
 ('700-square', 0.44257497787475586),
 ('50,000-square', 0.43563973903656006),
 ('3,000-square', 0.43486514687538147),
 ('30,000-square', 0.4330596923828125),
 ('footed', 0.43236875534057617)]
```

A reason for the most similar being "45,000-square" is that the training data may have higher frequencies of this usage in comparison to usages of foot with 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'**).

[22]:
```python
# For example: x, y, a, b = ("", "", "", "")
# ------------------
# Write your implementation here.
x, y, a, b = ("red", "stop", "green", "go")

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

```
[('stopping', 0.6390065550804138),
 ('stops', 0.5932266712188721),
 ('stopped', 0.58301842212677),
 ('going', 0.5744050145149231),
 ('doing', 0.5701577067375183),
 ('trying', 0.5597556233406067),
 ('instead', 0.5584668517112732),
 ('way', 0.5553772449493408),
 ('anyone', 0.5373578667640686),
 ('putting', 0.5353236198425293)]
```

red:stop::green:stopping

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

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

Run the cell below, to examine (a) which terms are most similar to "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.

[23]:
```
# 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', 0.49046966433525085),
 ('skills', 0.49005508422851156),
 ('ethic', 0.4897659420967102),
 ('business', 0.4875851273536682),
 ('respected', 0.485920250415802),
 ('practice', 0.482104629278183),
 ('regarded', 0.4778572618961334),
 ('life', 0.4760662019252777)]

[('professions', 0.5957458019256592),
 ('practitioner', 0.49884128570556664),
 ('teaching', 0.48292145133018494),
 ('nursing', 0.48211807012557983),
 ('vocation', 0.4788965880870819),
 ('teacher', 0.47160351276397705),
 ('practicing', 0.46937811374664307),
 ('educator', 0.46524322032928467),
 ('physicians', 0.4628995656967163),
 ('professionals', 0.4601393938064575)]
```

There is a clear bias when comparing female-associated words to male-associated words. Many of the female-associated words are stereotypical social connotations of what professions a woman should have, like nursing and teacher. On the other hand, male-associated words don't have a specific profession but more of positive descriptors or qualities one looks for while in their profession such as skills, respected, and regarded.

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

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

[24]:
```python
# ------------------
# Write your implementation here.
pprint.pprint(wv_from_bin.most_similar(positive=['man', 'parent'],␣
    ↪negative=['woman']))
print()
pprint.pprint(wv_from_bin.most_similar(positive=['woman', 'parent'],␣
    ↪negative=['man']))

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

```
[('company', 0.533617377281189),
 ('subsidiary', 0.5269285440444946),
 ('group', 0.520355224609375),
 ('unit', 0.5042728185653687),
 ('conglomerate', 0.4979137182235718),
 ('inc.', 0.49362319707870483),
 ('ltd.', 0.48146069049835205),
 ('holdings', 0.47016191482543945),
 ('viacom', 0.46933019161224365),
 ('owned', 0.4572541117668152)]

[('parents', 0.5398458242416382),
 ('sister', 0.5310081243515015),
 ('spouse', 0.5216396450996399),
 ('mothers', 0.5120096206665039),
 ('mother', 0.506610095500946),
 ('pregnant', 0.5012178421020508),
 ('caregiver', 0.49629634618759155),
 ('child', 0.4731905460357666),
 ('daughter', 0.4659813642501831),
 ('children', 0.4635886251926422)]
```

When looking at man and woman association with the word parent, the bias shown is that women are given analogies to parenting while men are more corporate analogies. This can be seen as a bias towards both sides, men not being associated with parenting and women confined to just parenting.

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

a. Give one explanation of how bias gets into the word vectors. Briefly describe a real-world example that demonstrates this source of bias. Your real-world example should be focused on word vectors, as opposed to bias in other AI systems (e.g., ChatGPT).

Bias can get into word vectors through its training data. Gender stereotypes, for example, can be within the trained on corpus leading word vectors to learn on this bias. For instance, news articles used for data could include sentences associating engineers and doctors to men or nurses and teachers to women which further strengthens the gender stereotypes in the word vectors.

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.

A method for mitigating bias is by debiasing during training. A known way to do this is called Hard Debiasing which is to first identify the gender subspace to obtain the direction of bias then neutralize the vectors in this subspace such that they are orthogonal to the direction. The gender-specific terms are then equalized to ensure equal distance with neutral terms. For example, "man-woman" pairs are made equidistant to "doctor".

2 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.