CSCI5800 Assignment 1: Exploring Word Vectors (100 Points)

Due 11:59pm, Friday Sep 20

Welcome to CSCI5800!

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 review the resources and references that we provided last week.

Assignment Notes: Please make sure to save the notebook as you go along. Submission Instructions are located at the bottom of the notebook.

```
# 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/'
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\inuku\AppData\Roaming\nltk_data...
[nltk_data]   Package reuters is already up-to-date!
```

Word Vectors

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

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

Part 1: Count-Based Word Vectors (40 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).

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 the following resources 7, 8, and 9. 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.

```
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….

```
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] (10 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
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
    words = [word for doc in corpus for word in doc]
    corpus_words = sorted(set(words))
    n_corpus_words = len(corpus_words)

    ### SOLUTION END

    return corpus_words, n_corpus_words

# ---------------------
# 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] (10 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.

```
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))
    word2ind = {word: i for i, word in enumerate(words)}
    for doc in corpus:
        for i, word in enumerate(doc):
            for j in range(max(0, i - window_size), min(len(doc), i +
window_size + 1)):
```

```
                if i != j:
                    M[word2ind[word], word2ind[doc[j]]] += 1
    ### SOLUTION END

    return M, word2ind

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

Question 1.3: Implement `reduce_to_k_dim` [code] (5 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.

```
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-dimensional 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)
    M_reduced = svd.fit_transform(M)
    ### SOLUTION END

    print("Done.")
    return M_reduced

# ---------------------
# 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] (5 point)

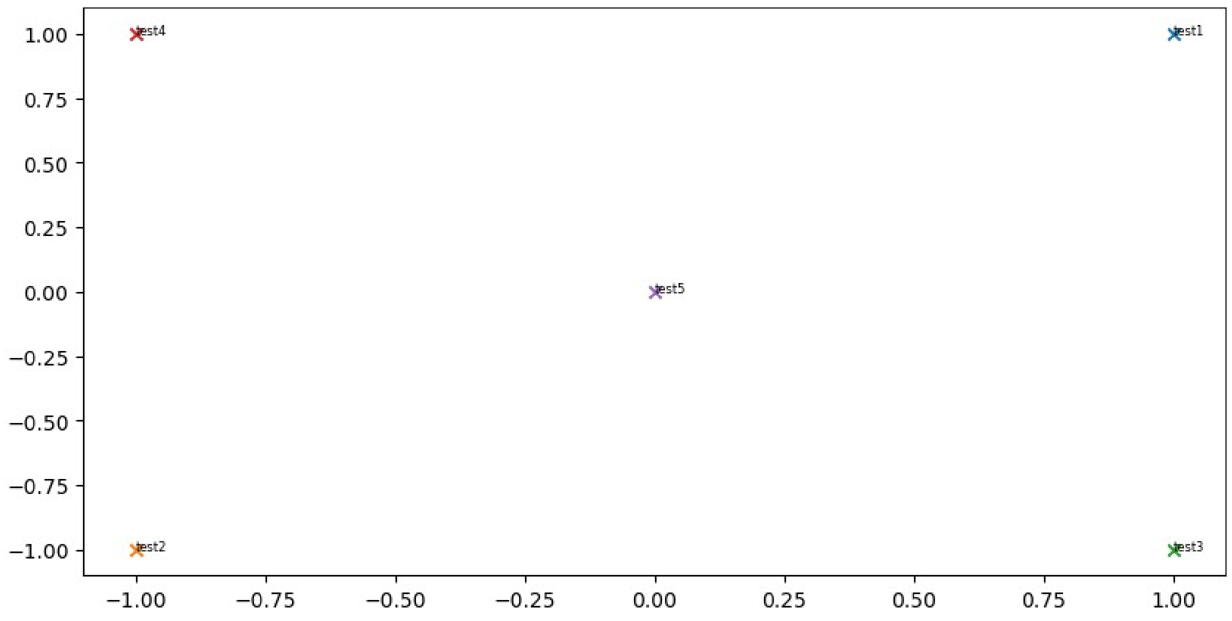
Here you will write a function to plot a set of 2D vectors in 2D space. For graphs, we will use Matplotlib (`plt`).

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.

```
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
    for word in words:
        x, y = M_reduced[word2ind[word]]
        plt.scatter(x, y, marker='x')
        plt.annotate(word, (x, y), fontsize=6)
    plt.show()
    ### SOLUTION END

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

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

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

```
print ("-" * 80)
```

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



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

Question 1.5: Co-Occurrence Plot Analysis [written] (10 points)

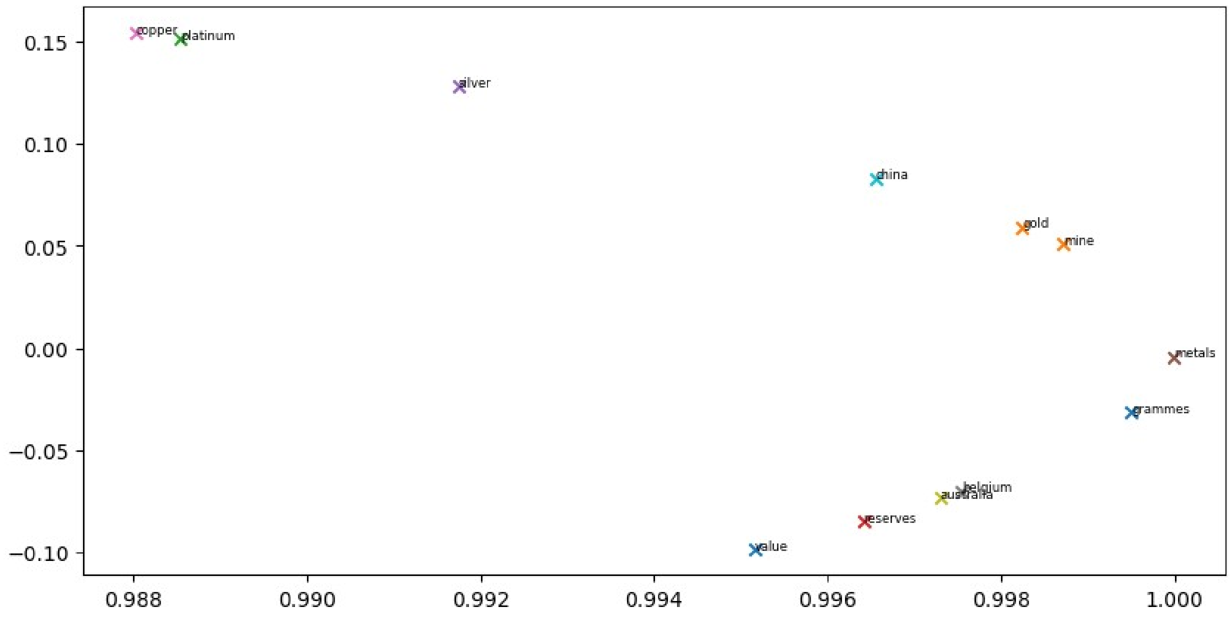
Now we will put together all the parts you have written! We will compute the co-occurrence matrix with fixed window of 4 (the default window size), over the Reuters "gold" corpus. Then we will use TruncatedSVD to compute 2-dimensional embeddings of each word. TruncatedSVD returns U*S, so we need to normalize 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.

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

In the figure, there are 2 groups of clusters that are together in the 2-dimensional embeding space. The clusters are ((australia, belgium), (copper, platinum)). For the cluster 1 i.e; (australia, belgium) we can confidently assume that these are countries and also Australia is known for its mining industry, particularly coal, copper and metal ores. In the same way, belgium is known for its steel industry which plays a crucial part in their economy. So that's why they are grouped together. For cluster 2 i.e; (copper, platinum) which are obviously metals. Apart from that property, these 2 metals have high melting points compared to the other metals(Copper - 1983, Platinum - 3220). Also, these 2 metals have high ductility and malleability.

SOLUTION END

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

SOLUTION BEGIN

In my opinion, all the metals(copper, platinum, gold, silver, metals) should have been clustered together. Also, all the countries(china, australia, belgium) should have been clustered together.

SOLUTION END

Part 2: Prediction-Based Word Vectors (60 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.

```
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

# -----------------------------------
# Run Cell to Load Word Vectors
# Note: This will take a couple minutes
# -----------------------------------
wv_from_bin = load_embedding_model()
```

```
Loaded vocab size 400000
```

Note: If you are receiving a "reset by peer" error, rerun the cell to restart the download. If you run into an "attribute" error, you may need to update to the most recent version of gensim and numpy. You can upgrade them inline by uncommenting and running the below cell:

```
#!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.

```
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

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

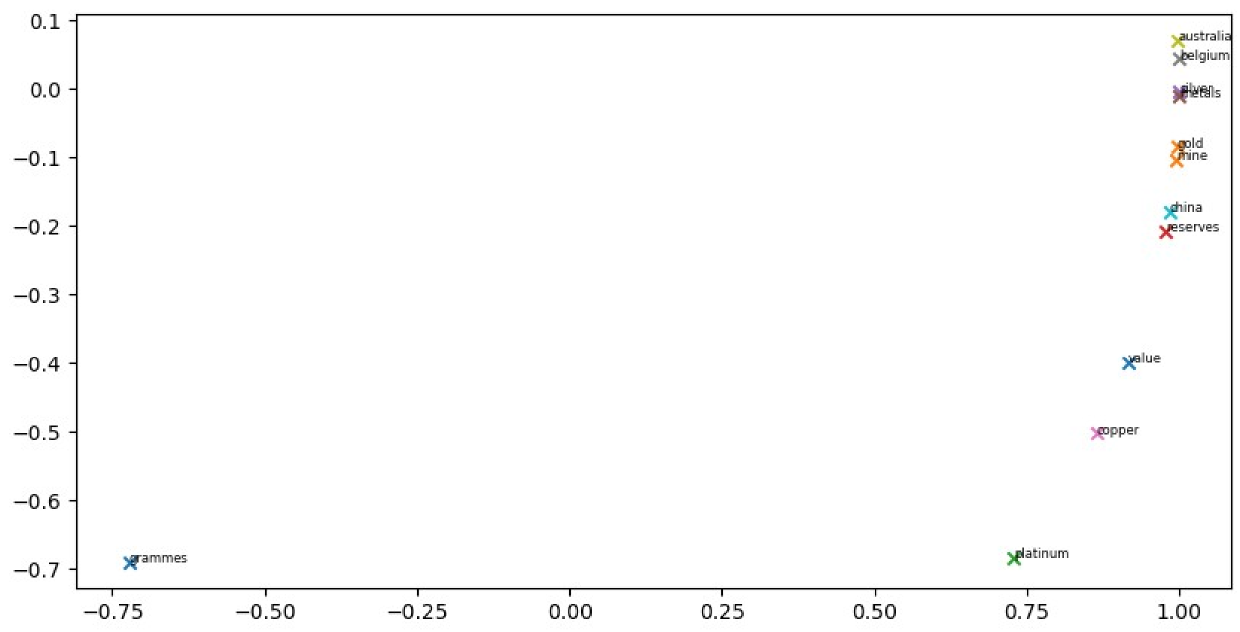
```
Shuffling words ...
Putting 10000 words into word2ind and matrix M...
Done.
Running Truncated SVD over 10012 words...
Done.
```

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] (10 points)

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

```
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?

SOLUTION BEGIN

Difference: In the plot generated from the co-occurence matrix, "copper" and "platinum" were clustered together. But in the plot generated using GloVe, they are apart. Not only that, "china" and "reserves" are clustered together but in the plot generated from the co-occurence matrix, they are apart.

Similarity: The similarities are "belgium" and "australia" are still close to each other and also "gold" and "mine" are also clustered together.

SOLUTION END

b. What is a possible cause for the difference?

SOLUTION BEGIN

The main difference is that in the earlier plot that is computed using the co-occurence matrix, it only computes the occurences based on the given corpus(it only counts the number of times the words come together) so the embeddings are purely based on the co-occurnece counts whereas GloVe uses a combination of global word-word co-occurence statistics and the local context window method. This helps in generating the word embeddings that contains both the global and local context therefore being different from the other.

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:

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

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

Question 2.2: Words with Multiple Meanings (10 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**.

```
### SOLUTION BEGIN
word = 'sink'
display(wv_from_bin.most_similar(word))
word = "bank"
display(wv_from_bin.most_similar(word))
### SOLUTION END

[('sinks', 0.6829907298088074),
 ('sinking', 0.588735818862915),
 ('bottom', 0.5451675057411194),
 ('sunk', 0.5268896222114563),
 ('drain', 0.5154818892478943),
 ('sank', 0.502458930015564),
 ('capsize', 0.4759066104888916),
 ('tub', 0.46636176109313965),
 ('bathroom', 0.4648846685886383),
 ('water', 0.45993712544441223)]
```

```
[('banks', 0.7625691294670105),
 ('banking', 0.6818838119506836),
 ('central', 0.6283639073371887),
 ('financial', 0.6166561841964722),
 ('credit', 0.6049751043319702),
 ('lending', 0.5980607867240906),
 ('monetary', 0.5963001251220703),
 ('bankers', 0.5913101434707642),
 ('loans', 0.5802939534187317),
 ('investment', 0.574020266532898)]
```

SOLUTION BEGIN

I have discovered the word sink. The word sink has different meanings like "To descend to the bottom of a body of water" or "A bowl-shaped fixture may or may not contain water". The top 10 most similar words contain these 2 meanings. The words (sinks, sinking, bottom, sunk, sank) refer to the meaning bottom and the words (drain, tub, bathroom, water) refer to the meaning fixture. I have also tried different words like bank, spring, date, etc. They always had only one meaning in the top 10. This is because though the word contains different meanings it might not be distinctive enough to generate multiple set of associated words i.e; they might not be used a lot with respect to the context that appears in the corpus. For example, the word bank is always referred to as the financial institution though it also refers to the side of a river. In the similar way, spring is always referred to as the season than that of the device that provides more jumping ability.

SOLUTION END

Question 2.3: Synonyms & Antonyms (8 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.

```
### SOLUTION BEGIN

w1 = "good"
w2 = "nice"
w3 = "bad"
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 good, nice have cosine distance: 0.3369309902191162
Antonyms good, bad have cosine distance: 0.2890373468399048
```

SOLUTION BEGIN

This might have happened because there might be more contexts in which "good" and "bad" come together. In the corpus there might be instances where polysemous words might have been used instead of "nice" to express the context. One more reason might be that "good" and "nice" are synonyms that they are less likely to be used in the same context. These reasons might be a reason that the similarity between "good" and "bad" is higher than "good" and "nice".

SOLUTION END

Question 2.4: Analogies with Word Vectors [written] (8 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).

```
# 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', 0.7200807332992554),
 ('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, 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

x = w + g - m

SOLUTION END

Question 2.5: Finding Analogies [code + written] (8 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

The explanation would be that the similar words for "grandmother" are "granddaughter", "daughter", and "mother".

SOLUTION END

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 = "good", "better", "bad", "worse"
assert wv_from_bin.most_similar(positive=[a, y], negative=[x])[0][0]
== b
```

```
### SOLUTION END
```

SOLUTION BEGIN

"Better" is the comparitive degree of "good" and "worse" is comparitive degree of "bad"

SOLUTION END

Question 2.6: Incorrect Analogy [code + written] (8 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?

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

```
[('45,000-square', 0.4922032058238983),
 ('15,000-square', 0.4649604558944702),
 ('10,000-square', 0.45447564125061035),
 ('6,000-square', 0.4497581440734863),
 ('3,500-square', 0.4441334009170532),
 ('700-square', 0.44257503747940063),
 ('50,000-square', 0.4356396794319153),
 ('3,000-square', 0.43486514687538147),
 ('30,000-square', 0.4330596923828125),
 ('footed', 0.43236875534057617)]
```

SOLUTION BEGIN

This might be because the corpus that the model is trained on is not diverse enough to produce the correct analogical output. Also, there might not be enough instances to show the relationship between "foot" and "sock".

SOLUTION END

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'**).

```
### SOLUTION BEGIN

x, y, a, b = "hammer", "nail", "comb", "hair"
pprint.pprint(wv_from_bin.most_similar(positive=[a, y], negative=[x]))

### SOLUTION END
```

```
[('combing', 0.5292600989341736),
 ('combed', 0.5028147101402283),
 ('biters', 0.47489652037620544),
 ('fingernail', 0.4577341675758362),
 ('stains', 0.4522605538368225),
 ('tooth', 0.4444238841533661),
 ('biter', 0.4298933148384094),
 ('scouring', 0.42609766125679016),
 ('scour', 0.4234057366847992),
 ('sealants', 0.4202999174594879)]
```

SOLUTION BEGIN

The correct value of the analogy needs to be "hair" but instead we recieved "combing", "combed", "biters", etc.

SOLUTION END

Question 2.7: Guided Analysis of Bias in Word Vectors [written] (5 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.

```
# Run this cell
# Here `positive` indicates the list of words to be similar to and
`negative` indicates the list of words to be
# most dissimilar from.

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

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

[('professions', 0.5957457423210144),
 ('practitioner', 0.49884128570556 64),
 ('teaching', 0.48292139172554016),
 ('nursing', 0.48211804032325745),
 ('vocation', 0.4788965880870819),
 ('teacher', 0.47160351276397705),
 ('practicing', 0.46937811374664307),
 ('educator', 0.46524322032928467),
 ('physicians', 0.46289944648742676),
 ('professionals', 0.4601393938064575)]
```

SOLUTION BEGIN

The difference is that in the first list it contains more skilled and desired qualities of a highly paid job whereas in the second list it contains low paying jobs when compared to the first list. Since, they are listed as most similar to "man" and the "woman" contains a very different set of words, it implies that the word embedding model shows bias towards "men".

SOLUTION END

Question 2.8: Independent Analysis of Bias in Word Vectors [code + written] (5 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.

```
### SOLUTION BEGIN

A = "hispanic"
B = "asian"
word = "migration"
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
```

```
[('latino', 0.5569910407066345),
 ('hispanics', 0.5489482283592224),
 ('latinos', 0.5167452096939087),
 ('emigration', 0.5153346657752991),
 ('undocumented', 0.4990084171295166),
 ('immigrant', 0.4896863102912903),
 ('immigrants', 0.466378390789032),
 ('immigration', 0.4577556848526001),
 ('population', 0.4564073085784912),
 ('migrant', 0.4309920072555542)]

[('asia', 0.6410027742385864),
 ('thailand', 0.49570488929748535),
 ('migratory', 0.4921424090862274),
 ('europe', 0.48791617155075073),
 ('migrations', 0.4777598977088928),
 ('economic', 0.46717625856399536),
 ('china', 0.45631271600723267),
 ('malaysia', 0.4490801990032196),
 ('japan', 0.4481150507926941),
 ('countries', 0.4432101547718048)]
```

SOLUTION BEGIN

Hispanic immigrants appear to more often in the context of negative words like undocumented whereas the asian immigrants does not have any negative words. This shows racial bias.

SOLUTION END

Question 2.9: Thinking About Bias [written] (6 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.

SOLUTION BEGIN

The simple explanation is that bias gets into the word vectors is from the training corpus. One real world example of this source of bias is the word embeddings that were generated using the Google news corpus. The word vectors were found be exhibiting gender bias. For example, the words like "engineer" were closely associated with male pronouns and the words like "receptionist" were closely associated with female pronouns.

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.

SOLUTION BEGIN

The method used to mitigate bias is called as Debiasing. This means that identifying gender-related biases in the word embeddings and then modifying the vectors to remove these biases. The study by Bolukbasi et al. applied a debiasing technique to the word embeddings learned from the Google News corpus to reduce gender bias. They first identified gender-specific words and contexts by looking at the cosine similarity between pairs of words and analyzing which pairs showed the largest differences between male and female pronouns. Then, they removed the gender associations from these words by projecting them onto a subspace that removed the gender component.

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.
6. Submit both your PDF and your jupyter notebook.