```
#%% md
## NLP with DL Assignment 1:
# Exploring Word Vectors
### <font color='red'> Due 12:00am, Sun April 27 </font>



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

If you aren't super familiar with Python, Numpy, or Matplotlib, we recommend you check out a Stanford's [tutorial](https://cs231n.github.io/python-numpy-tutorial/) about Python/Numpy.

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

```
#%% md
## 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](https://en.wikipedia.org/wiki/Word_embedding)
```

```
53 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*".
54 #%% md
55 ## Part 1: Count-Based Word Vectors
56
57 Most word vector models start from the following
   idea:
58
59 *You shall know a word by the company it keeps ([
   Firth, J. R. 1957:11](https://en.wikipedia.org/wiki
   /John_Rupert_Firth))*
60
61 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
   Word Embedding lecture.
62 #%% md
63 ### Co-Occurrence
64
65 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.
66
```

```
67 **Example: Co-Occurrence with Fixed Window of n=1
   **:
68
69 Document 1: "all that glitters is not gold"
70
71 Document 2: "all is well that ends well"
72
73
74 |     *     | `<START>` | all | that | glitters |
   is   | not  | gold  | well | ends | `<END>` |
75 |-----------|-------|-----|------|----------|------
   |------|-------|------|------|-----|
76 | `<START>`     | 0     | 2   | 0    | 0        | 0
       | 0    | 0     | 0    | 0    | 0   |
77 | all       | 2     | 0   | 1    | 0        | 1
       | 0    | 0     | 0    | 0    | 0   |
78 | that      | 0     | 1   | 0    | 1        | 0
       | 0    | 0     | 1    | 1    | 0   |
79 | glitters | 0     | 0   | 1    | 0        | 1
       | 0    | 0     | 0    | 0    | 0   |
80 | is        | 0     | 1   | 0    | 1        | 0
       | 1    | 0     | 1    | 0    | 0   |
81 | not       | 0     | 0   | 0    | 0        | 1
       | 0    | 1     | 0    | 0    | 0   |
82 | gold      | 0     | 0   | 0    | 0        | 0
       | 1    | 0     | 0    | 0    | 1   |
83 | well      | 0     | 0   | 1    | 0        | 1
       | 0    | 0     | 0    | 1    | 1   |
84 | ends      | 0     | 0   | 1    | 0        | 0
       | 0    | 0     | 1    | 0    | 0   |
85 | `<END>`       | 0     | 0   | 0    | 0        | 0
       | 0    | 1     | 1    | 0    | 0   |
86
87 **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.
88
```

```
89 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 $U_k$.
90
91 ![Picture of an SVD](./imgs/svd.png "SVD")
92
93 This reduced-dimensionality co-occurrence
   representation preserves semantic relationships
   between words, e.g. *doctor* and *hospital* will
   be closer than *doctor* and *dog*.
94
95 **Notes:** If you can barely remember what an
   eigenvalue is, here's [a slow, friendly
   introduction to SVD](https://davetang.org/file/
   Singular_Value_Decomposition_Tutorial.pdf). 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](https://en.wikipedia.
   org/wiki/Singular_value_decomposition#
   Truncated_SVD) - then there are reasonably
   scalable techniques to compute those iteratively.
96 #%% md
97 ### Plotting Co-Occurrence Word Embeddings
98
99 Here, we will be using the Reuters (business and
   financial news) corpus. If you haven't run the
```

```
 99 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.
100 #%%
101 def read_corpus(category="gold"):
102     """ Read files from the specified Reuter's
    category.
103         Params:
104             category (string): category name
105         Return:
106             list of lists, with words from each of
     the processed files
107     """
108     files = reuters.fileids(category)
109     return [[START_TOKEN] + [w.lower() for w in
    list(reuters.words(f))] + [END_TOKEN] for f in
    files]
110
111 #%% md
112 Let's have a look what these documents are like….
113 #%%
114 reuters_corpus = read_corpus()
115 pprint.pprint(reuters_corpus[:3], compact=True,
    width=100)
116 #%% md
117 ### Question 1.1: Implement `distinct_words` [code
    ]
118
119 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
```

```
119 do it with Python list comprehensions. In
    particular, [this](https://coderwall.com/p/rcmaea/
    flatten-a-list-of-lists-in-one-line-in-python) may
     be useful to flatten a list of lists. If you're
    not familiar with Python list comprehensions in
    general, here's [more information](https://python-
    3-patterns-idioms-test.readthedocs.io/en/latest/
    Comprehensions.html).
120
121 Your returned `corpus_words` should be sorted. You
     can use python's `sorted` function for this.
122
123 You may find it useful to use [Python sets](https
    ://www.w3schools.com/python/python_sets.asp) to
    remove duplicate words.
```

```
124 #%%
125 def distinct_words(corpus):
126     """ Determine a list of distinct words for the
     corpus.
127         Params:
128             corpus (list of list of strings):
    corpus of documents
129         Return:
130             corpus_words (list of strings): sorted
     list of distinct words across the corpus
131             n_corpus_words (integer): number of
    distinct words across the corpus
132     """
133     corpus_words = []
134     n_corpus_words = -1
135
136     ### SOLUTION BEGIN
137     flattened_list = [y for x in corpus for y in x
    ]
138     corpus_words = sorted(set(flattened_list))
139     n_corpus_words = len(corpus_words)
140     ### SOLUTION END
141
142     return corpus_words, n_corpus_words
```

```
143 #%%
144 # ---------------------
```

```
145 # Run this sanity check
146 # Note that this not an exhaustive check for
    correctness.
147 # ---------------------
148
149 # Define toy corpus
150 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(" ")]
151
152 test_corpus_words, num_corpus_words =
    distinct_words(test_corpus)
153
154
155 # Correct answers
156 ans_test_corpus_words = sorted([START_TOKEN, "All"
    , "ends", "that", "gold", "All's", "glitters", "
    isn't", "well", END_TOKEN])
157 ans_num_corpus_words = len(ans_test_corpus_words)
158
159 # Test correct number of words
160 assert(num_corpus_words == ans_num_corpus_words),
    "Incorrect number of distinct words. Correct: {}.
    Yours: {}".format(ans_num_corpus_words,
    num_corpus_words)
161
162 # Test correct words
163 assert (test_corpus_words == ans_test_corpus_words
    ), "Incorrect corpus_words.\nCorrect: {}\nYours
    :   {}".format(str(ans_test_corpus_words), str(
    test_corpus_words))
164
165 # Print Success
166 print ("-" * 80)
167 print("Passed All Tests!")
168 print ("-" * 80)
```

```
169 #%% md
170 ### Question 1.2: Implement `
    compute_co_occurrence_matrix` [code]
171
```

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
    # create initial M in size nXn
    M = np.zeros((n_words, n_words))
    word2ind = dict(zip(words, range(n_words)))

    for document in corpus:
        for i, center_word in enumerate(document):
            center_index = word2ind[center_word]
            left_bound = max(0, i - window_size)
            right_bound = min(len(document), i + window_size + 1)

            for j in range(left_bound, right_bound):
                if i != j:
                    neighbor_word = document[j]
                    neighbor_index = word2ind[neighbor_word]
                    M[center_index, neighbor_index] += 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]
```

```
256         correct = M_test_ans[idx1, idx2]
257         if student != correct:
258             print("Correct M:")
259             print(M_test_ans)
260             print("Your M: ")
261             print(M_test)
262             raise AssertionError("Incorrect count
    at index ({}, {})=({}, {}) in matrix M. Yours has
     {} but should have {}.".format(idx1, idx2, w1, w2
    , student, correct))
263 
264 # Print Success
265 print ("-" * 80)
266 print("Passed All Tests!")
267 print ("-" * 80)
```

```
268 #%% md
269 ### Question 1.3: Implement `reduce_to_k_dim` [
    code]
270 
271 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.
272 
273 **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](https://
    scikit-learn.org/stable/modules/generated/sklearn.
    decomposition.TruncatedSVD.html).
```

```
274 #%%
275 def reduce_to_k_dim(M, k=2):
276     """ Reduce a co-occurence count matrix of
    dimensionality (num_corpus_words, num_corpus_words
    )
277         to a matrix of dimensionality (
    num_corpus_words, k) using the following SVD
```

```
277 function from Scikit-Learn:
278             - http://scikit-learn.org/stable/
    modules/generated/sklearn.decomposition.
    TruncatedSVD.html
279 
280         Params:
281             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
282             k (int): embedding size of each word
    after dimension reduction
283         Return:
284             M_reduced (numpy matrix of shape (
    number of corpus words, k)): matrix of k-
    dimensioal word embeddings.
285                     In terms of the SVD from math
    class, this actually returns U * S
286     """
287     n_iters = 10     # Use this parameter in your
    call to `TruncatedSVD`
288     M_reduced = None
289     print("Running Truncated SVD over %i words..."
     % (M.shape[0]))
290 
291     ### SOLUTION BEGIN
292     svd = TruncatedSVD(n_components=k, n_iter=
    n_iters)
293     M_reduced = svd.fit_transform(M)
294     ### SOLUTION END
295 
296     print("Done.")
297     return M_reduced
298 #%%
299 # ---------------------
300 # Run this sanity check
301 # Note that this is not an exhaustive check for
    correctness
302 # In fact we only check that your M_reduced has
    the right dimensions.
303 # ---------------------
```

```
304
305 # Define toy corpus and run student code
306 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(" ")]
307 M_test, word2ind_test =
    compute_co_occurrence_matrix(test_corpus,
    window_size=1)
308 M_test_reduced = reduce_to_k_dim(M_test, k=2)
309
310 # Test proper dimensions
311 assert (M_test_reduced.shape[0] == 10), "M_reduced
     has {} rows; should have {}".format(
    M_test_reduced.shape[0], 10)
312 assert (M_test_reduced.shape[1] == 2), "M_reduced
    has {} columns; should have {}".format(
    M_test_reduced.shape[1], 2)
313
314 # Print Success
315 print ("-" * 80)
316 print("Passed All Tests!")
317 print ("-" * 80)
318 #%% md
319 ### Question 1.4: Implement `plot_embeddings` [
    code]
320
321 Here you will write a function to plot a set of 2D
     vectors in 2D space. For graphs, we will use
    Matplotlib (`plt`).
322
323 For this example, you may find it useful to adapt
     [this code](http://web.archive.org/web/
    20190924160434/https://www.pythonmembers.club/2018
    /05/08/matplotlib-scatter-plot-annotate-set-text-
    at-label-each-point/). In the future, a good way
    to make a plot is to look at [the Matplotlib
    gallery](https://matplotlib.org/gallery/index.html
    ), find a plot that looks somewhat like what you
    want, and adapt the code they give.
324 #%%
```

```
325
326 def plot_embeddings(M_reduced, word2ind, words):
327     """ Plot in a scatterplot the embeddings of
    the words specified in the list "words".
328         NOTE: do not plot all the words listed in
    M_reduced / word2ind.
329         Include a label next to each point.
330
331         Params:
332             M_reduced (numpy matrix of shape (
    number of unique words in the corpus , 2)): matrix
     of 2-dimensioal word embeddings
333             word2ind (dict): dictionary that maps
    word to indices for matrix M
334             words (list of strings): words whose
    embeddings we want to visualize
335     """
336
337     ### SOLUTION BEGIN
338     plt.figure(figsize=(10, 10))
339     for word in words:
340         if word in word2ind:
341             index = word2ind[word]
342             embedding = M_reduced[index]
343             plt.scatter(embedding[0], embedding[1
    ])
344             plt.annotate(word, xy=(embedding[0],
    embedding[1]), xytext=(5, 2), textcoords='offset
    points', ha='right', va='bottom')
345     plt.show()
346     ### SOLUTION END
```

```
347 #%%
348 # ----------------------
349 # Run this sanity check
350 # Note that this is not an exhaustive check for
    correctness.
351 # The plot produced should look like the "test
    solution plot" depicted below.
352 # ----------------------
353
354 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)
```

```
#%% md
### Question 1.5: Co-Occurrence Plot Analysis [written]

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](https://jakevdp.github.io/PythonDataScienceHandbook/02.05-computation-on-arrays-broadcasting.html).

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)
```

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

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

```
#%% md
### SOLUTION BEGIN
Groups of words that cluster together in 2-dimensional embedding space:
1. The group of 'copper', 'platinum' & 'silver' which are all types of metals cluster together. However, silver appears slightly farther from copper and platinum, which makes sense given the broader semantic silver might have compared to the other two.
2. The group of 'belgium' & 'australia' form a tight cluster, which aligns with their semantic similarity as country names.

### SOLUTION END
```

```
396
397 #%% md
398 b. What doesn't cluster together that you might
    think should have? Describe at least two examples.
399 #%% md
400 ### SOLUTION BEGIN
401
402 1. I would expect that 'metals' and 'gold' will
    also cluster with the group of  'copper', '
    platinum' & 'silver', due to their semantic
    relations.
403 2. I would expect that also 'china' will be in the
     cluster of 'belgium' & 'australia' as a country
    name as well.
404 ### SOLUTION END
405 #%% md
406 ## Part 2: Prediction-Based Word Vectors
407
408 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](https://nlp.stanford.edu/pubs/
    glove.pdf).
409
410 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.
411 #%%
412 def load_embedding_model():
413     """ Load GloVe Vectors
414         Return:
```

```
415             wv_from_bin: All 400000 embeddings,
    each lengh 200
416     """
417     import gensim.downloader as api
418     wv_from_bin = api.load("glove-wiki-gigaword-
    200")
419     print("Loaded vocab size %i" % len(list(
    wv_from_bin.index_to_key)))
420     return wv_from_bin
421 #%%
422 # ----------------------------------
423 # Run Cell to Load Word Vectors
424 # Note: This will take a couple minutes
425 # ----------------------------------
426 wv_from_bin = load_embedding_model()
427 #%% md
428 #### 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:
429 #%%
430 #!pip install gensim --upgrade
431 #!pip install numpy --upgrade
432 #%% md
433 ### Reducing dimensionality of Word Embeddings
434 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.
435 Run the following cells to:
436
437 1. Put 10000 Glove vectors into a matrix M
438 2. Run `reduce_to_k_dim` (your Truncated SVD
    function) to reduce the vectors from 200-
    dimensional to 2-dimensional.
439 #%%
440 def get_matrix_of_vectors(wv_from_bin,
    required_words):
441     """ Put the GloVe vectors into a matrix M.
```

```
442         Param:
443             wv_from_bin: KeyedVectors object; the
    400000 GloVe vectors loaded from file
444         Return:
445             M: numpy matrix shape (num words, 200
    ) containing the vectors
446             word2ind: dictionary mapping each word
     to its row number in M
447     """
448     import random
449     words = list(wv_from_bin.index_to_key)
450     print("Shuffling words ...")
451     random.seed(225)
452     random.shuffle(words)
453     words = words[:10000]
454     print("Putting %i words into word2ind and
    matrix M..." % len(words))
455     word2ind = {}
456     M = []
457     curInd = 0
458     for w in words:
459         try:
460             M.append(wv_from_bin.get_vector(w))
461             word2ind[w] = curInd
462             curInd += 1
463         except KeyError:
464             continue
465     for w in required_words:
466         if w in words:
467             continue
468         try:
469             M.append(wv_from_bin.get_vector(w))
470             word2ind[w] = curInd
471             curInd += 1
472         except KeyError:
473             continue
474     M = np.stack(M)
475     print("Done.")
476     return M, word2ind
477 #%%
478 # ---------------------------------------------
```

```
478 -----------------
479 # Run Cell to Reduce 200-Dimensional Word
    Embeddings to k Dimensions
480 # Note: This should be quick to run
481 # ----------------------------------------------
    -----------------
482 M, word2ind = get_matrix_of_vectors(wv_from_bin,
    words)
483 M_reduced = reduce_to_k_dim(M, k=2)
484
485 # Rescale (normalize) the rows to make them each
    of unit-length
486 M_lengths = np.linalg.norm(M_reduced, axis=1)
487 M_reduced_normalized = M_reduced / M_lengths[:, np
    .newaxis] # broadcasting
488 #%% md
489 **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.**
490 #%% md
491 ### Question 2.1: GloVe Plot Analysis [written]
492
493 Run the cell below to plot the 2D GloVe embeddings
     for `['value', 'gold', 'platinum', 'reserves', '
    silver', 'metals', 'copper', 'belgium', 'australia
    ', 'china', 'grammes', "mine"]`.
494 #%%
495 words = ['value', 'gold', 'platinum', 'reserves',
    'silver', 'metals', 'copper', 'belgium', '
    australia', 'china', 'grammes', "mine"]
496
497 plot_embeddings(M_reduced_normalized, word2ind,
    words)
498 #%% md
```

```
499 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?
500 #%% md
501 ### SOLUTION BEGIN
502 1. This GloVe plot different by clustering silver
    , metal, and gold together as expected given their
     strong semantic relationships.
503 2. This GloVe plot is similar to the previous one
    in that it clusters australia and belgium together
    , while china isn't cluster with them as might
    expect.
504 ### SOLUTION END
505 #%% md
506 b. What is a possible cause for the difference?
507 #%% md
508 ### SOLUTION BEGIN
509 Since GloVe leverages global word co-occurrence
    statistics across the entire corpus, the
    differences between this plot and the previous one
     may reflect broader contextual similarities,
    rather than just local window-based associations.
510 ### SOLUTION END
511 #%% md
512 ### Cosine Similarity
513 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.
514
515 We can think of n-dimensional vectors as points in
     n-dimensional space. If we take this perspective
     [L1](http://mathworld.wolfram.com/L1-Norm.html)
    and [L2](http://mathworld.wolfram.com/L2-Norm.html
    ) 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:
516
517 <img src="./imgs/inner_product.png" width=20%
```

```
style="float: center;"></img>

Instead of computing the actual angle, we can leave the similarity in terms of $similarity = cos(\Theta)$. Formally the [Cosine Similarity](https://en.wikipedia.org/wiki/Cosine_similarity) $s$ between two vectors $p$ and $q$ is defined as:

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

```
#%% md
### Question 2.2: Words with Multiple Meanings [code + written]
Polysemes and homonyms are words that have more than one meaning (see this [wiki page](https://en.wikipedia.org/wiki/Polysemy) 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](https://radimrehurek.com/gensim/
```

```
models/keyedvectors.html#gensim.models.keyedvectors.FastTextKeyedVectors.most_similar)__.
#%%
### SOLUTION BEGIN
wv_from_bin.most_similar("light")
### SOLUTION END
#%% md
### SOLUTION BEGIN
The word I discovered is 'light,' whose meanings include both 'brightness' and 'low weight' and more. This polysemy is reflected in its top similar words, such as 'bright' (number 1) and 'heavy' (number 5).

In my opinion, many other polysemous or homonymic words didn't work because one of their meanings presented much more in the data and therefore in the embedding while the other is underrepresentation.
### SOLUTION END
#%% md
### Question 2.3: Synonyms & Antonyms [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,
```

```
548 w2)` function here in order to compute the cosine
    distance between two words. Please see the __[
    GenSim documentation](https://radimrehurek.com/
    gensim/models/keyedvectors.html#gensim.models.
    keyedvectors.FastTextKeyedVectors.distance)__ for
    further assistance.
549 #%%
550 ### SOLUTION BEGIN
551
552 w1 = "light"
553 w2 = "illuminate"
554 w3 = "heavy"
555
556 w1_w2_dist = wv_from_bin.distance(w1, w2)
557 w1_w3_dist = wv_from_bin.distance(w1, w3)
558
559 print("Synonyms {}, {} have cosine distance: {}".
    format(w1, w2, w1_w2_dist))
560 print("Antonyms {}, {} have cosine distance: {}".
    format(w1, w3, w1_w3_dist))
561
562 ### SOLUTION END
563 #%% md
564 ### SOLUTION BEGIN
565 A possible explanation for why this counter-
    intuitive result may have happened is that
    although synonyms share similar meanings, antonyms
     often co-occur in similar contexts and therefore
    share some semantic features that represented much
     more in the data than the synonyms.
566 In this case, light and heavy are both related to
    weight, and probably appear in the same context in
     the date more than light and illuminate.
567
568 ### SOLUTION END
569 #%% md
570 ### Question 2.4: Analogies with Word Vectors [
    written]
571 Word vectors have been shown to *sometimes*
    exhibit the ability to solve analogies.
572
```

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](https://radimrehurek.com/gensim/models/keyedvectors.html#gensim.models.keyedvectors.KeyedVectors.most_similar)__. 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](https://www.aclweb.org/anthology/N18-2039.pdf)). 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']))
```

#%% md

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?

#%% md

SOLUTION BEGIN

Using vector arithmetic, we can express relationship between `man` & `grandfather` as:

```
586
587 g-m
588
589 This represents the vector that, when added to "
    man," results in "grandfather."
590
591 To find the analogous relationship for "woman," we
     will need add this difference vector to "woman":
592
593 w+(g-m)
594
595 Therefore, we can express X as:
596 x = w + (g - m)
597 ### SOLUTION END
598 #%% md
599 ### Question 2.5: Finding Analogies [code +
    written]
600 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?
601 #%% md
602 ### SOLUTION BEGIN
603 These following similar words probably occur in
    similar contexts which is why they are ranked
    highly.  This is likely due to the fact that these
     words share similar semantic features, such as
    being related to familial relationships.
604 ### SOLUTION END
605 #%% md
606 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.
607
608 **Note**: You may have to try many analogies to
    find one that works!
609 #%%
```

```
### SOLUTION BEGIN

x = 'paris'
y = 'france'
a = 'berlin'
b = 'germany'
assert wv_from_bin.most_similar(positive=[a, y],
    negative=[x])[0][0] == b

### SOLUTION END
```

```
#%% md
### SOLUTION BEGIN
The full analogy is: "paris : france :: berlin :
    germany".
This analogy holds because Paris is the capital
    city of France, just as Berlin is the capital city
     of Germany. The vectors for these words are
    likely to be close to each other in the vector
    space due to their shared semantic relationship as
     capital cities and countries.
### SOLUTION END
```

```
#%% md
### Question 2.6: Incorrect Analogy [code +
    written]
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']))
```

```
#%% md
### SOLUTION BEGIN
A potential reason for this unexpected result that
     a 'foot' is a polysemous that often associated
    with a measurement of area (square feet).

The model has thus found words that describe areas
     instead of the expected association of "foot".

```

```
This could be due to the fact that the word "foot" is more often used in contexts related to measurements, and the model has learned to associate it with other words that are commonly used in those contexts.
### SOLUTION END
```

```
#%% md
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

# correct analogy
x, y, a, b = "start", "stop", "execute", "cancel"
# incorrect analogy
pprint.pprint(wv_from_bin.most_similar(positive=[a, y], negative=[x]))

### SOLUTION END
```

```
#%% md
### SOLUTION BEGIN
The expected analogy is "start : stop :: execute : cancel", where "execute" is meant in the sense of "to launch" or "to carry out". However, the model instead returns words related to capital punishment, such as "kill" and "punish".
### SOLUTION END
```

```
#%% md
### Question 2.7: Guided Analysis of Bias in Word Vectors [written]

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']))
```

```
#%% md
### SOLUTION BEGIN
The difference between the two lists reveals a gender bias in how these word vectors represent the relationship between gender and professions. The women vector tends to contain more "soft" and specific words like: vocation and teaching, while the men vector contains more powerful and abstract words like: reputation, ethic, business.
### SOLUTION END
```

```
#%% md
### Question 2.8: Independent Analysis of Bias in Word Vectors [code + written]

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

```

```
677 A = "caucasian"
678 B = "african"
679 word = "education"
680 pprint.pprint(wv_from_bin.most_similar(positive=[A
    , word], negative=[B]))
681 print()
682 pprint.pprint(wv_from_bin.most_similar(positive=[B
    , word], negative=[A]))
683
684 ### SOLUTION END
```

```
685 #%% md
686 ### SOLUTION BEGIN
687 The model associates "Caucasian + education" with
    formal, academic concepts, and "African +
    education" with development, aid, and public
    health — reinforcing existing stereotypes about
    educational access and systems based on race.
688 ### SOLUTION END
```

```
689 #%% md
690 ### Question 2.9: Thinking About Bias [written]
691
692 a. Give one explanation of how bias gets into the
    word vectors. Briefly describe a real-world
    example that demonstrates this source of bias.
```

```
693 #%% md
694 ### SOLUTION BEGIN
695 Bias gets into the word vectors primarily through
    the data they are trained on. If the data contains
     societal biases, those biases can be reflected in
     the word vectors. For example, if the training
    data frequently associates certain professions
    with one gender, the word vectors will pick up on
    these associations. The data might contain a
    higher frequency of phrases like "female nurse"
    and "male doctor," reflecting the historical
    gender roles in these professions. As a result,
    the trained word vectors might place "nurse"
    closer to "woman" and "doctor" closer to "man" in
    the embedding space, perpetuating and even
    amplifying this gender bias.
696 ### SOLUTION END
```

```
#%% md
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.
#%% md

### SOLUTION BEGIN
One method to mitigate this bias is to preprocess the data and modify biased occurrences or by augmenting the dataset with additional examples that counterbalance the bias.
One more method is to use debiasing algorithms that adjust the word vectors after training to reduce the bias.
A real world example of this is the "Word Embedding Association Test" (WEAT), which evaluates the bias in word embeddings by measuring the association between target words (e.g., professions) and attribute words (e.g., gendered terms). By using this test, researchers can identify biased associations in word vectors and then apply debiasing techniques to reduce these biases.


### SOLUTION END
#%% md
# <font color="red"> Submission Instructions</font>

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).
2. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
3. 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. <font color='blue'> Make sure all
```

```
714 your solutions especially the coding parts are
    displayed in the pdf</font>, it's okay if the
    provided codes get cut off because lines are not
    wrapped in code cells).
715 4. 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!
716 5. Submit your PDF on **MAMA** system.
```