

ASSIGNMENT 01: EXPLORING WORD VECTORS

▼ PART 01: COUNT-BASED WORD VECTORS

```
# All Import Statements Defined Here
# Note: Do not add to this list.
# All the dependencies you need, can be installed by running .
# -----

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

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')
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
%matplotlib inline

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

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

[nltk_data] Downloading package reuters to /root/nltk_data...

```
def read_corpus(category="grain"):
    """ 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]
```

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

```
[[ '<START>', 'china', 'daily', 'says', 'vermin', 'eat', '7', '-', '12', 'pct', 'grain', 'stocks',
  'a', 'survey', 'of', '19', 'provinces', 'and', 'seven', 'cities', 'showed', 'vermin', 'consume',
  'between', 'seven', 'and', '12', 'pct', 'of', 'china', '"', 's', 'grain', 'stocks', ',', 'the',
  'china', 'daily', 'said', '.', 'it', 'also', 'said', 'that', 'each', 'year', '1', '.', '575',
  'mln', 'tonnes', ',', 'or', '25', 'pct', ',', 'of', 'china', '"', 's', 'fruit', 'output', 'are',
  'left', 'to', 'rot', ',', 'and', '2', '.', '1', 'mln', 'tonnes', ',', 'or', 'up', 'to', '30',
  'pct', ',', 'of', 'its', 'vegetables', '.', 'the', 'paper', 'blamed', 'the', 'waste', 'on',
  'inadequate', 'storage', 'and', 'bad', 'preservation', 'methods', '.', 'it', 'said', 'the',
  'government', 'had', 'launched', 'a', 'national', 'programme', 'to', 'reduce', 'waste', ',',
  'calling', 'for', 'improved', 'technology', 'in', 'storage', 'and', 'preservation', ',', 'and',
  'greater', 'production', 'of', 'additives', '.', 'the', 'paper', 'gave', 'no', 'further',
  'details', '.', '<END>'],
[ '<START>', 'thai', 'trade', 'deficit', 'widens', 'in', 'first', 'quarter', 'thailand', '"', 's',
  'trade', 'deficit', 'widened', 'to', '4', '.', '5', 'billion', 'baht', 'in', 'the', 'first',
  'quarter', 'of', '1987', 'from', '2', '.', '1', 'billion', 'a', 'year', 'ago', ',', 'the',
  'business', 'economics', 'department', 'said', '.', 'it', 'said', 'janunary', '/', 'march',
  'imports', 'rose', 'to', '65', '.', '1', 'billion', 'baht', 'from', '58', '.', '7', 'billion',
  '.', 'thailand', '"', 's', 'improved', 'business', 'climate', 'this', 'year', 'resulted', 'in',
  'a', '27', 'pct', 'increase', 'in', 'imports', 'of', 'raw', 'materials', 'and', 'semi', '-',
  'finished', 'products', '.', 'the', 'country', '"', 's', 'oil', 'import', 'bill', ',', 'however',
  ',', 'fell', '23', 'pct', 'in', 'the', 'first', 'quarter', 'due', 'to', 'lower', 'oil', 'prices',
  '.', 'the', 'department', 'said', 'first', 'quarter', 'exports', 'expanded', 'to', '60', '.', '6',
  'billion', 'baht', 'from', '56', '.', '6', 'billion', '.', 'export', 'growth', 'was', 'smaller',
  'than', 'expected', 'due', 'to', 'lower', 'earnings', 'from', 'many', 'key', 'commodities',
  'including', 'rice', 'whose', 'earnings', 'declined', '18', 'pct', ',', 'maize', '66', 'pct', ',',
  'sugar', '45', 'pct', ',', 'tin', '26', 'pct', 'and', 'canned', 'pineapples', 'seven', 'pct', '.',
```

```
'products', 'registering', 'high', 'export', 'growth', 'were', 'jewellery', 'up', '64', 'pct',
',', 'clothing', '57', 'pct', 'and', 'rubber', '35', 'pct', '.', '<END>'],
['<START>', 'sri', 'lanka', 'gets', 'usda', 'approval', 'for', 'wheat', 'price', 'food',
'department', 'officials', 'said', 'the', 'u', '.', 's', '.', 'department', 'of', 'agriculture',
'approved', 'the', 'continental', 'grain', 'co', 'sale', 'of', '52', ',', '500', 'tonnes', 'of',
'soft', 'wheat', 'at', '89', 'u', '.', 's', '.', 'dlrs', 'a', 'tonne', 'c', 'and', 'f', 'from',
'pacific', 'northwest', 'to', 'colombo', '.', 'they', 'said', 'the', 'shipment', 'was', 'for',
'april', '8', 'to', '20', 'delivery', '.', '<END>']]
```

▼ 1.1 IMPLEMENT DISTINCT WORDS CODE

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

```
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): list of distinct words across the corpus, sorted (using python 'sorted' function)
            num_corpus_words (integer): number of distinct words across the corpus
    """

    # -----
    # Write your implementation here.
    corpus_words = sorted(list(set([word for words_list in corpus for word in words_list])))
    num_corpus_words = len(corpus_words)

    # -----

    return corpus_words, num_corpus_words
```

```
# -----
# Run this sanity check
# Note that this not an exhaustive check for correctness.
# -----

# Define toy corpus
test_corpus = ["START All that glitters isn't gold END".split(" "), "START All's well that ends well END".split(" ")]
test_corpus_words, num_corpus_words = distinct_words(test_corpus)

# Correct answers
ans_test_corpus_words = sorted(list(set(["START", "All", "ends", "that", "gold", "All's", "glitters", "isn't", "well", "END"])))
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), test_corpus_words)

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

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

▼ 1.2 IMPLEMENT COMPUTE CO OCCURRENCE MATRIX CODE

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".
```



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

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

▼ 1.3 IMPLEMENT REDUCE TO K-DIM CODE

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](http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.TruncatedSVD.html).

```
def reduce_to_k_dim(M, k=2):
    """ Reduce a co-occurrence 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 corpus words, number of corpus words)): co-occurrence 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]))

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

    # -----

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

```
# -----
# Run this sanity check
# Note that this 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 = ["START All that glitters isn't gold END".split(" "), "START All's well that ends well END".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!
-----
```

▼ 1.4 IMPLEMENT PLOT EMBEDDINGS CODE

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

```
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 , k)): matrix of k-dimensioal word embeddings
word2Ind (dict): dictionary that maps word to indices for matrix M
words (list of strings): words whose embeddings we want to visualize

"""

Write your implementation here.

x_coords = M_reduced[:, 0]

y_coords = M_reduced[:, 1]

for word in words:

 idx = word2Ind[word]

 embedding = M_reduced[idx]

 x = embedding[0]

 y = embedding[1]

 plt.scatter(x, y, marker='x', color='red')

 plt.text(x, y, word, fontsize=9)

Run this sanity check

Note that this 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]])

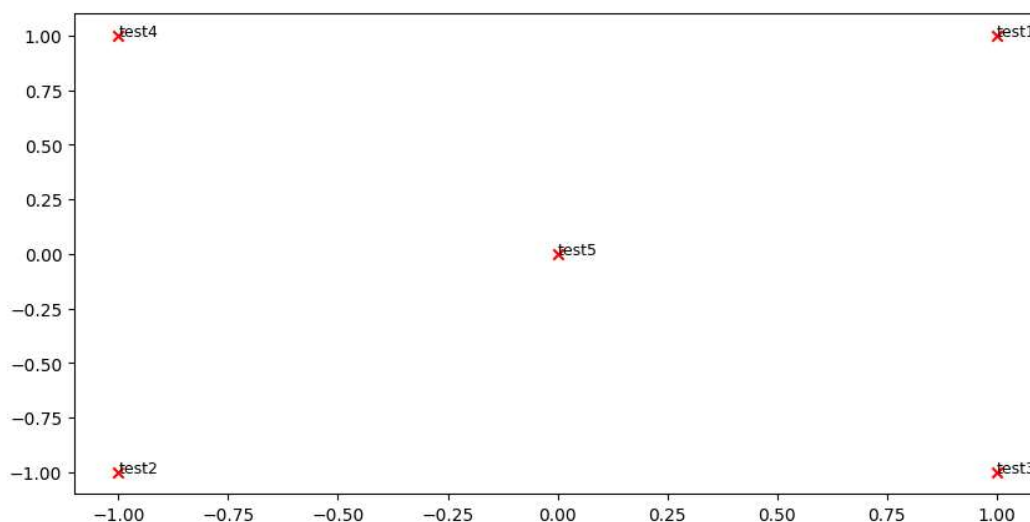
word2Ind_plot_test = {'test1': 0, 'test2': 1, 'test3': 2, 'test4': 3, 'test5': 4}

words = ['test1', 'test2', 'test3', 'test4', 'test5']

plot_embeddings(M_reduced_plot_test, word2Ind_plot_test, words)

print ("-" * 80)

Outputted Plot:



▼ 1.5 CO-OCCURRENCE PLOT ANALYSIS

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 "grain" corpus. Then we will use TruncatedSVD to compute 2-dimensional embeddings of each word. TruncatedSVD returns $U \cdot S$, 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.

Run the below cell to produce the plot. It'll probably take a few seconds to run. What clusters together in 2-dimensional embedding space? What doesn't cluster together that you might think should have?

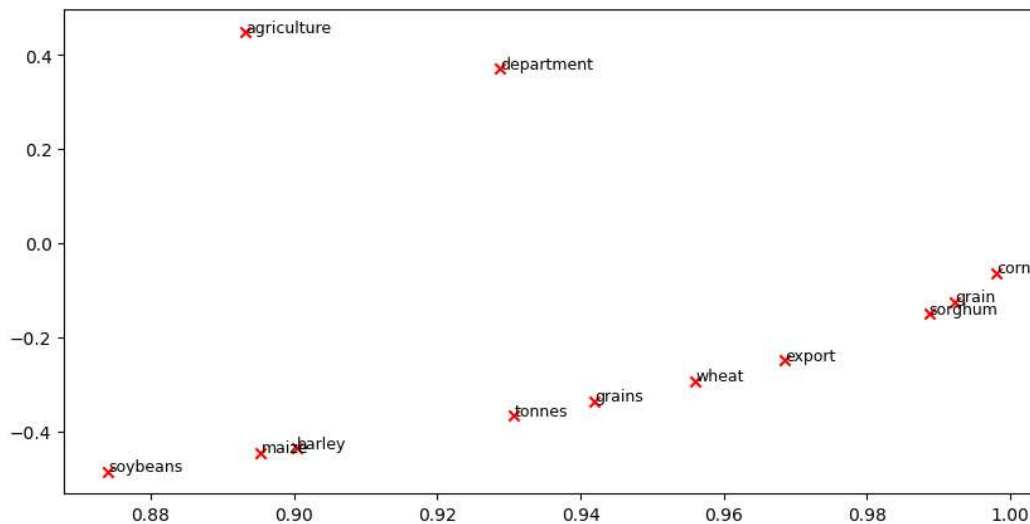
```
# -----
# Run This Cell to Produce Your Plot
# -----
reuters_corpus = read_corpus()
M_co_occurrence, word2ind_co_occurrence = compute_co_occurrence_matrix(reuters_corpus)
M_reduced_co_occurrence = reduce_to_k_dim(M_co_occurrence, k=2)

# 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 = ['tonnes', 'grain', 'wheat', 'agriculture', 'corn', 'maize', 'export', 'department', 'barley', 'grains', 'soybeans', 'sorghum']

plot_embeddings(M_normalized, word2ind_co_occurrence, words)
```

Running Truncated SVD over 7146 words...
Done.



▼ Part 2: PREDICTION-BASED WORD VECTORS

```
def load_embedding_model():
    """ Load GloVe Vectors
    Return:
        wv_from_bin: All 400000 embeddings, each length 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()
```

[=====] 100.0% 252.1/252.1MB downloaded
Loaded vocab size 400000

```
def get_matrix_of_vectors(wv_from_bin, required_words=['tonnes', 'grain', 'wheat', 'agriculture', 'corn', 'maize', 'export', 'department']):
    """ 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)
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.

```

▼ 2.1 GloVe PLOT ANALYSIS

Run the cell below to plot the 2D GloVe embeddings for:

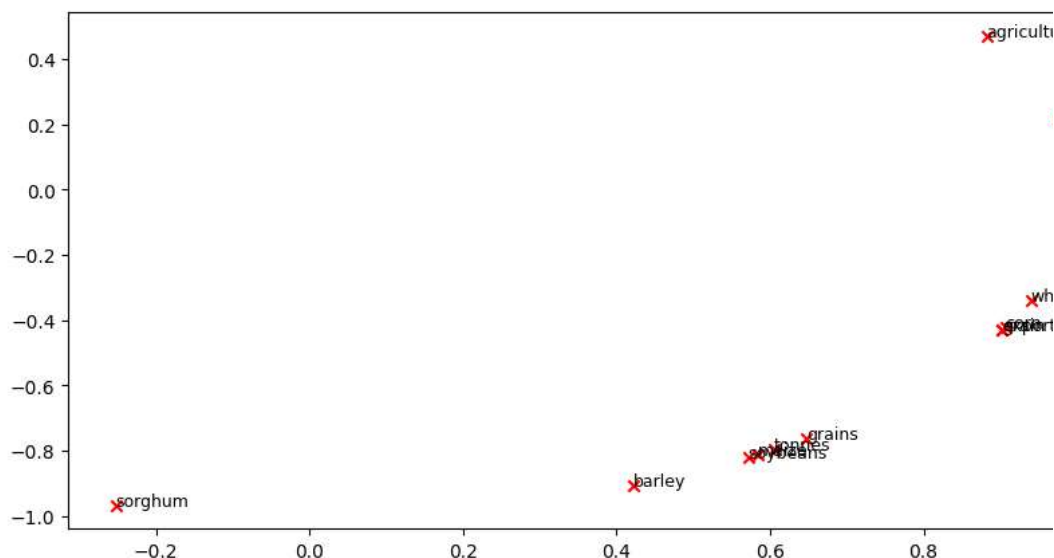
['tonnes', 'grain', 'wheat', 'agriculture', 'corn', 'maize', 'export', 'department', 'barley', 'grains', 'soybeans', 'sorghum'].

What clusters together in 2-dimensional embedding space? What doesn't cluster together that you think should have? How is the plot different from the one generated earlier from the co-occurrence matrix? What is a possible cause for the difference?

```

words = ['tonnes', 'grain', 'wheat', 'agriculture', 'corn', 'maize', 'export', 'department', 'barley', 'grains', 'soybeans', 'sorghum']
plot_embeddings(M_reduced_normalized, word2ind, words)

```



▼ 2.2 WORDS WITH MULTIPLE MEANINGS

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

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

Note: You should use the `wv_from_bin.most_similar(word)` function to get the top 10 similar words. This function ranks all other words in the vocabulary with respect to their cosine similarity to the given word.

```
# -----
# Write your polysemous word exploration code here.
```

```
wv_from_bin.most_similar("leaves")
```

```
# -----

[('ends', 0.6128067970275879),
 ('leaf', 0.6027014255523682),
 ('stems', 0.5998532176017761),
 ('takes', 0.5902854204177856),
 ('leaving', 0.5761634707450867),
 ('grows', 0.5663397312164307),
 ('flowers', 0.5600922107696533),
 ('turns', 0.5536050796508789),
 ('leave', 0.5496848821640015),
 ('goes', 0.5434925556182861)]
```

▼ 2.3 SYNONYMS & ANTONYMS

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

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 $\text{Cosine Distance}(w_1, w_3) < \text{Cosine Distance}(w_1, w_2)$.

As an example, $w_1 = \text{"happy"}$ is closer to $w_3 = \text{"sad"}$ than to $w_2 = \text{"cheerful"}$. 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 `wv_from_bin.distance(w1, w2)` function here in order to compute the cosine distance between two words.

```
# -----
# Write your synonym & antonym exploration code here.
```

```
w1 = "happy"
w2 = "sad"
w3 = "cheerful"
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))
```

```
# -----

Synonyms happy, sad have cosine distance: 0.4040136933326721
Antonyms happy, cheerful have cosine distance: 0.5172466933727264
```

▼ 2.4 ANALOGIES WITH WORD VECTORS

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. 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. 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', 0.7200808525085449),
 ('daughter', 0.7168302536010742),
```



```
( 'mother', 0.7151536345481873),
( 'niece', 0.7005682587623596),
( 'father', 0.6659887433052063),
( 'aunt', 0.6623408794403076),
( 'grandson', 0.6618767976760864),
( 'grandparents', 0.644661009311676),
( 'wife', 0.6445354223251343)]
```

▼ 2.5 FINDING ANALOGIES

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!

```
# -----
# Write your analogy exploration code here.

pprint.pprint(wv_from_bin.most_similar(positive=['queen', 'king'], negative=['male']))

# -----

[( 'elizabeth', 0.5484128594398499),
( 'prince', 0.5440689325332642),
( 'monarch', 0.515655517578125),
( 'princess', 0.5155462026596069),
( 'ii', 0.5129325985908508),
( 'majesty', 0.5116104483604431),
( 'kingdom', 0.5021030902862549),
( 'sihanouk', 0.5017128586769104),
( 'crown', 0.4983074367046356),
( 'reign', 0.49080339074134827)]
```

▼ 2.6 GUIDED ANALYSIS OF BIAS IN WORD VECTORS

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 "girl" and "toy" and most dissimilar to "boy", and (b) which terms are most similar to "boy" and "toy" and most dissimilar to "girl". 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=['girl', 'toy'], negative=['boy']))
print()
pprint.pprint(wv_from_bin.most_similar(positive=['boy', 'toy'], negative=['girl']))

[( 'toys', 0.7094953060150146),
( 'doll', 0.5932914614677429),
( 'dolls', 0.570662260055542),
( 'barbie', 0.5407706499099731),
( 'mattel', 0.5328551530838013),
( 'accessories', 0.5206909775733948),
( 'hasbro', 0.49227219820022583),
( 'jewelry', 0.47385692596435547),
( 'lego', 0.4690813422203064),
( 'apparel', 0.46136239171028137)]

[( 'toys', 0.71570885181427),
( 'hasbro', 0.5164632797241211),
( 'robot', 0.47317108511924744),
( 'pet', 0.4670490324497223),
( 'manufacturer', 0.4668163061141968),
( 'mattel', 0.4582391679286957),
( 'lego', 0.45811763405799866),
( 'miniature', 0.4441472291946411),
( 'makers', 0.44298243522644043),
( 'manufactured', 0.44275349378585815)]
```

▼ 2.7 INDEPENDENT ANALYSIS OF BIAS IN WORD VECTORS

Use the *most_similar* function to find another case where some bias is exhibited by the vectors. Please briefly explain the example of bias that you discover.

```
# -----
# Write your bias exploration code here.

pprint.pprint(wv_from_bin.most_similar(positive=['male', 'job'], negative=['female']))
print()
pprint.pprint(wv_from_bin.most_similar(positive=['female', 'job'], negative=['male']))

# -----

[('jobs', 0.7089871168136597),
 ('doing', 0.6467409729957581),
 ('work', 0.6321760416030884),
 ('done', 0.6304946541786194),
 ('better', 0.6154061555862427),
 ('employment', 0.6153915524482727),
 ('good', 0.614001989364624),
 ('because', 0.6058511734008789),
 ('even', 0.5974840521812439),
 ('working', 0.5962860584259033)]

[('jobs', 0.7322045564651489),
 ('hiring', 0.6630983948707581),
 ('work', 0.6436149477958679),
 ('working', 0.6352189183235168),
 ('doing', 0.625247061252594),
 ('employment', 0.6032665967941284),
 ('hire', 0.6000232696533203),
 ('worked', 0.5974962711334229),
 ('he', 0.5893245935440063),
 ('better', 0.5890937447547913)]
```