NLPDL Assignment 1: Exploring Word Vectors (10 points)

Due 11:59pm, Sept 18

Welcome to NLPDL!

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

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

In [ ]:
```
# 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 ver
    the README.txt file found in the same directory as this notebook. Your Pythc

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 argum
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\28152\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 (10 points)

Most word vector models start from the following idea:

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

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

Co-Occurrence

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

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

Document 1: "all that glitters is not gold"

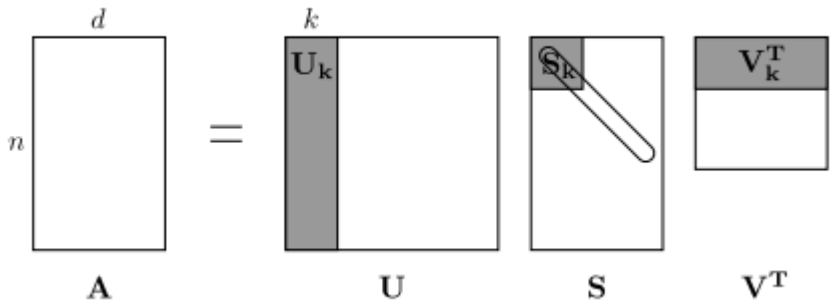
Document 2: "all is well that ends well"

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

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

Note: In NLP, we often add `<START>` and `<END>` tokens to represent the beginning and end of sentences, paragraphs or documents. In this case we imagine `<START>` and `<END>` tokens encapsulating each document, e.g., "`<START>` All that glitters is not gold `<END>`", and include these tokens in our co-occurrence counts.

The rows (or columns) of this matrix provide one type of word vectors (those based on word-word co-occurrence), but the vectors will be large in general (linear in the number of distinct words in a corpus). Thus, our next step is to run *dimensionality reduction*. In particular, we will run *SVD (Singular Value Decomposition)*, which is a kind of generalized *PCA (Principal Components Analysis)* to select the top k principal components. Here's a visualization of dimensionality reduction with SVD. In this picture our co-occurrence matrix is A with n rows corresponding to n words. We obtain a full matrix decomposition, with the singular values ordered in the diagonal S matrix, and our new, shorter length-k word vectors in U_k.



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

Notes: If you can barely remember what an eigenvalue is, here's a slow, friendly introduction to SVD. If you want to learn more thoroughly about PCA or SVD, feel free to check out lectures 7, 8, and 9 of CS168. These course notes provide a great high-level treatment of these general purpose algorithms. Though, for the purpose of this class, you only need to know how to extract the k-dimensional embeddings by utilizing pre-programmed implementations of these algorithms from the numpy, scipy, or sklearn python packages. In practice, it is challenging to apply full SVD to large corpora because of the memory needed to perform PCA or SVD. However, if you only want the top k vector components for relatively small k — known as Truncated SVD — then there are reasonably scalable techniques to compute those iteratively.

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 "grain" (i.e. news articles about corn, wheat, etc.) category. The function also adds `<START>` and `<END>` tokens to each of the documents, and lowercases words. You do **not** have to perform any other kind of pre-processing.

In [ ]:
```
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_T
```

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

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

```
[['<START>', 'china', 'daily', 'says', 'vermin', 'eat', '7', '-', '12', 'pct', 'g
rain', 'stocks',
  'a', 'survey', 'of', '19', 'provinces', 'and', 'seven', 'cities', 'showed', 've
rmin', 'consume',
  'between', 'seven', 'and', '12', 'pct', 'of', 'china', "'", 's', 'grain', 'stoc
ks', ',', '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', 'u
p', 'to', '30',
  'pct', ',', 'of', 'its', 'vegetables', '.', 'the', 'paper', 'blamed', 'the', 'w
aste', 'on',
  'inadequate', 'storage', 'and', 'bad', 'preservation', 'methods', '.', 'it', 's
aid', 'the',
  'government', 'had', 'launched', 'a', 'national', 'programme', 'to', 'reduce',
'waste', ',',
  'calling', 'for', 'improved', 'technology', 'in', 'storage', 'and', 'preservati
on', ',', 'and',
  'greater', 'production', 'of', 'additives', '.', 'the', 'paper', 'gave', 'no',
'further',
  'details', '.', '<END>'],
 ['<START>', 'thai', 'trade', 'deficit', 'widens', 'in', 'first', 'quarter', 'tha
iland', "'", 's',
  'trade', 'deficit', 'widened', 'to', '4', '.', '5', 'billion', 'baht', 'in', 't
he', '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', 'bil
l', ',', 'however',
  ',', 'fell', '23', 'pct', 'in', 'the', 'first', 'quarter', 'due', 'to', 'lowe
r', 'oil', 'prices',
  '.', 'the', 'department', 'said', 'first', 'quarter', 'exports', 'expanded', 't
o', '60', '.', '6',
  'billion', 'baht', 'from', '56', '.', '6', 'billion', '.', 'export', 'growth',
'was', 'smaller',
  'than', 'expected', 'due', 'to', 'lower', 'earnings', 'from', 'many', 'key', 'c
ommodities',
  'including', 'rice', 'whose', 'earnings', 'declined', '18', 'pct', ',', 'maiz
e', '66', 'pct', ',',
  'sugar', '45', 'pct', ',', 'tin', '26', 'pct', 'and', 'canned', 'pineapples',
'seven', 'pct', '.',
  'products', 'registering', 'high', 'export', 'growth', 'were', 'jewellery', 'u
p', '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', 'o
f', 'agriculture',
  'approved', 'the', 'continental', 'grain', 'co', 'sale', 'of', '52', ',', '50
0', 'tonnes', 'of',
```

```
 'soft', 'wheat', 'at', '89', 'u', '.', 's', '.', 'dlrs', 'a', 'tonne', 'c', 'an
d', 'f', 'from',
 'pacific', 'northwest', 'to', 'colombo', '.', 'they', 'said', 'the', 'shipmen
t', 'was', 'for',
 'april', '8', 'to', '20', 'delivery', '.', '<END>']]
```

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

Write a method to work out the distinct words (word types) that occur in the corpus. You can do this with `for` loops, but it's more efficient to do it with Python list comprehensions. In particular, this may be useful to flatten a list of lists. If you're not familiar with Python list comprehensions in general, here's more information.

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

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

In [ ]:
```
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
            n_corpus_words (integer): number of distinct words across the corpus
    """
    corpus_words = []
    n_corpus_words = -1

    # ------------------
    # Write your implementation here.

    corpus_words_unordered = [word for sentence in corpus for word in sentence]
    corpus_words_unordered = set(corpus_words_unordered)
    corpus_words_unordered = list(corpus_words_unordered)
    corpus_words = sorted(corpus_words_unordered)
    n_corpus_words = len(corpus_words)
    # ------------------

    return corpus_words, n_corpus_words
```

In [ ]:
```
# ---------------------
# 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_TOKE
test_corpus_words, num_corpus_words = distinct_words(test_corpus)

# Correct answers
ans_test_corpus_words = sorted([START_TOKEN, "All", "ends", "that", "gold", "All
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
```

```
# Test correct words
assert (test_corpus_words == ans_test_corpus_words), "Incorrect corpus_words.\nC

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

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

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

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

In [ ]:
```
def compute_co_occurrence_matrix(corpus, window_size=4):
    """ Compute co-occurrence matrix for the given corpus and window_size (defau

        Note: Each word in a document should be at the center of a window. Words
              number of co-occurring words.

              For example, if we take the document "<START> All that glitters is
              "All" will co-occur with "<START>", "that", "glitters", "is", and

        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
                Co-occurence matrix of word counts.
                The ordering of the words in the rows/columns should be the same
            word2ind (dict): dictionary that maps word to index (i.e. row/column
    """
    words, n_words = distinct_words(corpus)
    M = None
    word2ind = {}

    # ------------------
    # Write your implementation here.

    corpus_word, n_corpus_word = distinct_words(corpus)
    index = 0
    for word in corpus_word:
        word2ind.update({word:index})
        index = index + 1

    ma = []
    for i in range(n_corpus_word):
        row = []
        for j in range(n_corpus_word):
            row.append(0)
```

```
        ma.append(row)

    for sentence in corpus:
        number = 0
        for word in sentence:
            length = len(sentence)
            row = word2ind[word]
            for i in range(max(number-window_size,0),number):
                column = word2ind[sentence[i]]
                ma[row][column]+=1
            for i in range(number+1,min(number+window_size+1,length)):
                column = word2ind[sentence[i]]
                ma[row][column]+=1
            number += 1

    M = np.matrix(ma)
    # ------------------

    return M, word2ind
```

In [ ]:

```
# ---------------------
# 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_TOKE
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
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: {

# Test correct M shape
assert (M_test.shape == M_test_ans.shape), "M matrix has incorrect shape.\nCorre

# 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

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

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

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

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

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

In [ ]:
```
def reduce_to_k_dim(M, k=2):
    """ Reduce a co-occurence count matrix of dimensionality (num_corpus_words,
        to a matrix of dimensionality (num_corpus_words, k) using the following
            - http://scikit-learn.org/stable/modules/generated/sklearn.decomposi

        Params:
            M (numpy matrix of shape (number of unique words in the corpus , num
            k (int): embedding size of each word after dimension reduction
        Return:
            M_reduced (numpy matrix of shape (number of corpus words, k)): matri
                    In terms of the SVD from math class, this actually returns U
    """
    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 = np.asarray(M)
    M_reduced = svd.fit_transform(M)
    # ------------------

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

In [ ]:
```
# ---------------------
# 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_TOKE
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 {}".
assert (M_test_reduced.shape[1] == 2), "M_reduced has {} columns; should have {}

# 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` (1 point)

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

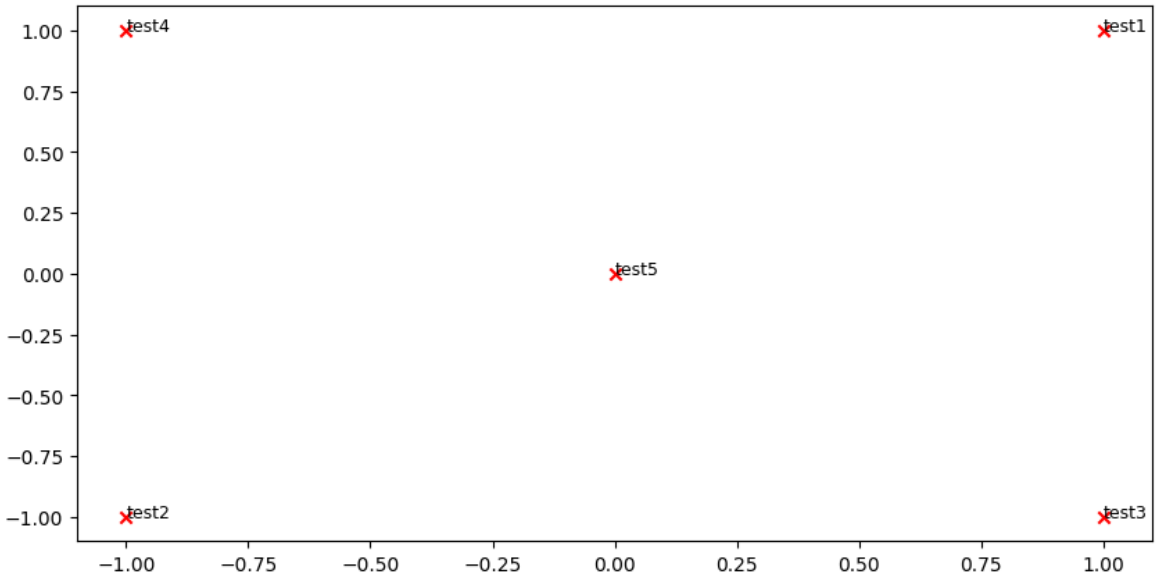
For this example, you may find it useful to adapt this code. In the future, a good way to make a plot is to look at the Matplotlib gallery, find a plot that looks somewhat like what you want, and adapt the code they give.

In [ ]:
```
def plot_embeddings(M_reduced, word2ind, words):
    """ Plot in a scatterplot the embeddings of the words specified in the list
        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 corp
            word2ind (dict): dictionary that maps word to indices for matrix M
            words (list of strings): words whose embeddings we want to visualize
    """

    # ------------------
    # Write your implementation here.

    for word in words:
        index = word2ind[word]
        x = M_reduced[index][0]
        y = M_reduced[index][1]
        plt.scatter(x,y,marker='x',color='red')
        plt.text(x,y,word,fontsize=9)
    plt.show()

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

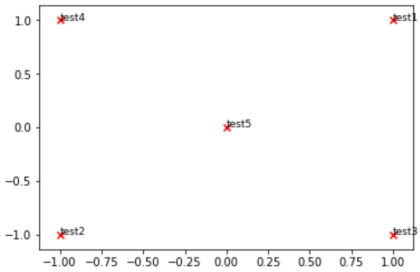
In [ ]:
```
# ---------------------
# Run this sanity check
# Note that this is not an exhaustive check for correctness.
```

```
# 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:



Test Plot Solution



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

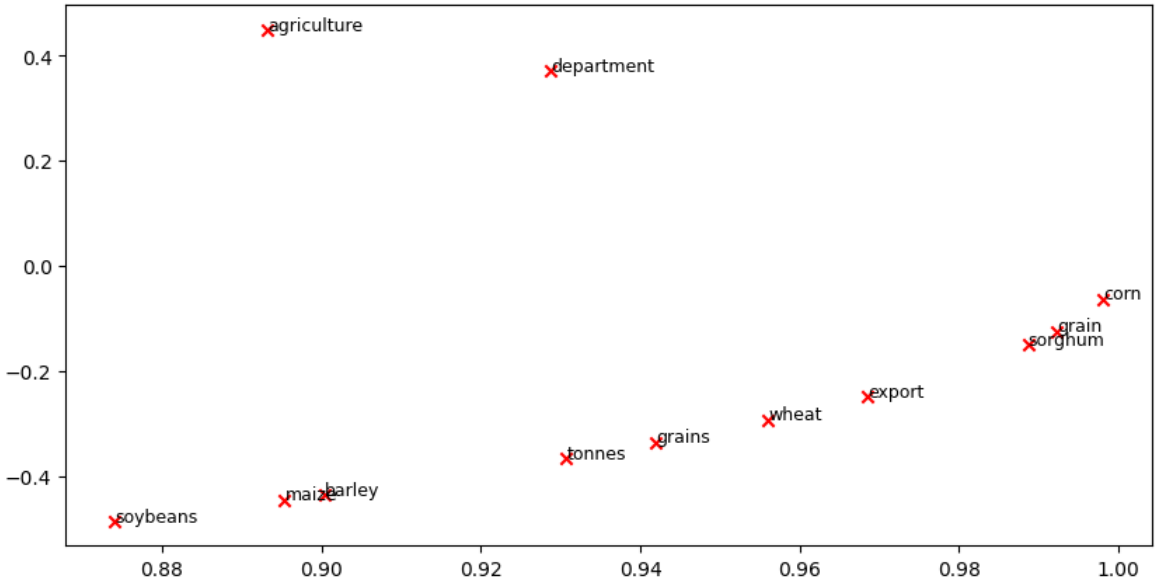
Now we will put together all the parts you have written! We will compute the co-occurrence matrix with fixed window of 4 (the default window size), over the Reuters "grain" corpus. Then we will use TruncatedSVD to compute 2-dimensional embeddings of each word. TruncatedSVD returns U*S, so we need to 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. What clusters together in 2-dimensional embedding space? What doesn't cluster together that you might think should have?

In [ ]:
```
# ----------------------------
# Run This Cell to Produce Your Plot
# ----------------------------
reuters_corpus = read_corpus()
M_co_occurrence, word2ind_co_occurrence = compute_co_occurrence_matrix(reuters_c
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',

plot_embeddings(M_normalized, word2ind_co_occurrence, words)
```

```
Running Truncated SVD over 7146 words...
Done.
```



Write your answer here.

the words "sorghum","grain","corn" cluster together, because they are all agricultural products. The word "agriculture" is far from others.

Submission Instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of all cells).
3. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.

4. Once you've rerun everything, select File -> Download as -> PDF via LaTeX (If you have trouble using "PDF via LaTex", you can also save the webpage as pdf. Make sure all your solutions especially the coding parts are displayed in the pdf, it's okay if the provided codes get cut off because lines are not wrapped in code cells).
5. Look at the PDF file and make sure all your solutions are there, displayed correctly. The PDF is the only thing your graders will see!
6. Submit your PDF on course.pku.edu.cn.