homework1-student

June 2, 2022

1 50.040 Natural Language Processing (Summer 2021) Homework 1

Due 3rd June 2022, 5pm

1.0.1 STUDENT ID: 1004365

1.0.2 Name: Lee Jet Xuen

1.0.3 Students with whom you have discussed (if any):

Tay Sze Chang 1004301
Brandon Chong 1004104

[1]:
```
import numpy as np
import math
from sklearn.decomposition import PCA
from sklearn.decomposition import TruncatedSVD
from matplotlib import pyplot as plt
from gensim.models import Word2Vec
```

[2]:
```
# !pip install gensim
np.seterr(divide='ignore', invalid='ignore')
```

[2]:
```
{'divide': 'warn', 'over': 'warn', 'under': 'ignore', 'invalid': 'warn'}
```

1.1 Introduction

Word embeddings are dense vectors that represent words, and capable of capturing semantic and syntactic similarity, relation with other words, etc. We have introduced two approaches in the class to learn word embeddings: **Count-based** and **Prediction-based**. Here we will explore both approaches. Note that we use "word embeddings" and "word vectors" interchangeably.

Before we start, you need to download the WikiText-2 dataset. Unzip the file and then put it under the "data" folder. The WikiText-2 dataset consists of multiple lines of long text. Please do not change the data unless you are requested to do so.

Environment: - Python 3.5 or above - gensim - sklearn - numpy

1.2 1. Count-Based Word Embeddings

1.2.1 Co-Occurrence

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

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

Document 1: "learn and live"

Document 2: "learn not and know not"

*	and	know	learn	live	not
and	0	1	1	1	1
know	1	0	0	0	1
learn	1	0	0	0	1
live	1	0	0	0	0
not	1	1	1	0	0

1.2.2 Normalized Pointwise Mutual Information

Pointwise Mutual Information (Prelude) Pointwise mutual information (PMI) is one of the most important concepts in NLP. The pointwise mutual information between a target word w and a context word c is defined as:

$$\text{PMI}(w, c) = \log_2 \frac{P(w, c)}{P(w)P(c)}$$

Given co-occurrence matrix $\mathbf{M} \in \mathbb{Z}^{N \times N}$ of N words, m_{ij} is the element of i th row and j th column. The PMI matrix can be calculated as

$$\text{PMI}_{ij} = \log_2 \frac{p_{ij}}{p_{i*}p_{*j}}$$

where

$$p_{ij} = \frac{m_{ij}}{\sum_{i=1}^{N}\sum_{j=1}^{N} m_{ij}} \quad p_{i*} = \frac{\sum_{j=1}^{N} m_{ij}}{\sum_{i=1}^{N}\sum_{j=1}^{N} m_{ij}} \quad p_{*j} = \frac{\sum_{i=1}^{N} m_{ij}}{\sum_{i=1}^{N}\sum_{j=1}^{N} m_{ij}}$$

For the details of PMI, please refer to https://web.stanford.edu/~jurafsky/slp3/6.pdf.

Normalized Pointwise Mutual Information In addition to PMI, pointwise mutual information can be normalized between [-1, +1]. The normalized mutual information between a target word w and a context word c is defined as:

$$\text{NPMI}(w, c) = \frac{pmi(w, c)}{h(w, c)} \tag{1}$$

where $h(x, y)$ is a joint self-information, which can estimated as $-\log_2 P(w, c)$

1.2.3 Principal Components Analysis (PCA) and Truncated Singular Value Decomposition (Truncated SVD)

The rows (or columns) of co-occurrence matrix or PPMI matrix can be utilized as word vectors, 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 first run PCA (Principal Components Analysis) to reduce the dimension. In practice, it is challenging to apply PCA to large corpora because of the memory needed to perform PCA. However, if you only want the top k vector components for relatively small k — known as Truncated SVD — then there are reasonably scalable techniques to compute those iteratively.

1.2.4 Read Corpus

Before you start, please make sure you have downloaded the dataset "WikiText-2" in the introduction.

[3]:
```
import string

def read_corpus(file_path, size=50000):
    '''
    params:
        file_path --- str: path to your data file.
        size --- int or str: the size of the corpus
    return:
        corpus --- list[str]: list of word strings.
    '''
    with open(file_path, 'r') as f:
        text = f.read()
        if size=='all':
            corpus = text.split()
        else:
            corpus = text.split()[:size]

        # ignores the <unk> tokens and punctuations
        corpus = [x.lower() for x in corpus if x != '<unk>' and x not in string.
 ↪punctuation]
        return corpus
```

Let's have a look at the corpus

[4]:
```
corpus = read_corpus(r'./data/wikitext-2/wiki.train.tokens')
print(corpus[0:100])
```

```
['valkyria', 'chronicles', 'iii', 'senjō', 'no', 'valkyria', '3', 'chronicles',
'japanese', '     3', 'lit', 'valkyria', 'of', 'the', 'battlefield', '3',
'commonly', 'referred', 'to', 'as', 'valkyria', 'chronicles', 'iii', 'outside',
'japan', 'is', 'a', 'tactical', 'role', '@-@', 'playing', 'video', 'game',
'developed', 'by', 'sega', 'and', 'media.vision', 'for', 'the', 'playstation',
'portable', 'released', 'in', 'january', '2011', 'in', 'japan', 'it', 'is',
'the', 'third', 'game', 'in', 'the', 'valkyria', 'series', 'the', 'same',
'fusion', 'of', 'tactical', 'and', 'real', '@-@', 'time', 'gameplay', 'as',
'its', 'predecessors', 'the', 'story', 'runs', 'parallel', 'to', 'the', 'first',
'game', 'and', 'follows', 'the', 'nameless', 'a', 'penal', 'military', 'unit',
'serving', 'the', 'nation', 'of', 'gallia', 'during', 'the', 'second',
'europan', 'war', 'who', 'perform', 'secret', 'black']
```

1.2.5 Question 1.1 [code]:

Implement the function "distinct_words" that reads in "corpus" and returns distinct words that appeared in the corpus and the number of distinct words.

Then, run the sanity check cell to check your implementation.

[5]:
```
def distinct_words(corpus):
    """
    Determine a list of distinct words for the corpus.
    Params:
        corpus --- list[str]: list of words in the corpus
    Return:
        corpus_words --- list[str]: list of distinct words in the corpus; sort␣
 ↪this list with built-in python function "sorted"
        num_corpus_words --- int: number of distinct words in the corpus
    """
    ### YOUR CODE HERE

    corpus_words = None
    num_corpus_words = None

    corpus_words = sorted(list(set(corpus)))
    num_corpus_words = len(corpus_words)

    ### END OF YOUR CODE

    return corpus_words, num_corpus_words
```

[6]:
```
# ---------------------------------------------------
# Run this sanity check to check your implementation
# ---------------------------------------------------
```

```
# Define toy corpus
test_corpus = "learn and live".split() + "learn not and know not".split()
test_corpus_words, num_corpus_words = distinct_words(test_corpus)

# Correct answers
# ans_test_corpus_words = sorted(list(set(['learn','and','live','not','know'])))
# ans_num_corpus_words = len(ans_test_corpus_words)

ans_test_corpus_words = ['and', 'know', 'learn', 'live', 'not']
ans_num_corpus_words = 5


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

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

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

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

1.2.6 Question 1.2 [code]:

Implement "compute_word_matrix" that reads in "corpus" and "window_size", and returns a co-occurrence matrix, NPMI matrix and a word-to-index dictionary.

Then, run the sanity check cell to check your implementation.

[7]:
```
def compute_word_matrix(corpus, window_size=1):
    """
    Compute co-occurrence matrix and PPMI matrix for the given corpus and␣
 ↪window_size (default of 1).

    Params:
        corpus --- list[str]: list of words
        window_size --- int: size of context window
    Return:
        CoM --- numpy array of shape (num_words, num_words):
                Co-occurrence 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.
        PPMI--- numpy array of shape (num_words, num_words):
                PPMI 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.


        word2index --- dict: dictionary that maps word to index (i.e. row/
↪column number) for matrix CoM which is the same as PPMI.
    """
    words, num_words = distinct_words(corpus)
    CoM, NPMI = None, None
    word2index = {}

    ### YOUR CODE HERE

    # implementation of word2index
    # for index, word in enumerate(words):
    #   word2index[word] = index
    word2index = {word: index for index, word in enumerate(words)}

    # implementation of CoM
    CoM = np.zeros((num_words, num_words))

    for i in range(len(corpus)):
        word_index = word2index[corpus[i]]
        left_window = max(i - window_size, 0)

        for j in range(left_window, i):
            window_word = corpus[j]

            CoM[word_index][word2index[window_word]] += 1
            CoM[word2index[window_word]][word_index] += 1

    # implementation of NPMI
    NPMI = np.zeros((num_words, num_words))

    matrix_sum = np.sum(CoM)
    mIj = np.reshape(np.sum(CoM, axis = 0), (num_words, -1))
    miJ = np.reshape(np.sum(CoM, axis = 1), (-1, num_words))

    P_ij = CoM/matrix_sum
    P_j = miJ/matrix_sum
    P_i = mIj/matrix_sum
    PiPj = np.matmul(P_i, P_j)
```

```
    PMI = np.log2(np.divide(P_ij, PiPj, where = P_ij!=0))
    NPMI = np.nan_to_num(np.divide(PMI, -np.log2(P_ij)))

    ### END OF YOUR CODE
    NPMI = np.round(NPMI, 7)

    return CoM, NPMI, word2index
```

[8]:
```
# ---------------------
# Run this sanity check
# ---------------------

# Define toy corpus and get co-occurrence matrix
test_corpus = "learn not and know not".split()
CoM_test, PPMI_test, word2Ind_test = compute_word_matrix(test_corpus,␣
 ↪window_size=1)
# Correct M and word2Ind
CoM_test_ans = np.array(
    [[0., 1., 0., 1.],
     [1., 0., 0., 1.],
     [0., 0., 0., 1.],
     [1., 1., 1., 0.]])

PPMI_test_ans = np.array(
[[0.       , 0.3333333, 0.       , 0.1383458],
 [0.3333333, 0.       , 0.       , 0.1383458],
 [0.       , 0.       , 0.       , 0.4716792],
 [0.1383458, 0.1383458, 0.4716792, 0.       ]]
)

word2Ind_ans = {'and':0, 'know':1, 'learn':2,  'not':3}

# check correct word2Ind
assert (word2Ind_ans == word2Ind_test), "Your word2Ind is incorrect:\nCorrect:␣
 ↪{}\nYours: {}".format(word2Ind_ans, word2Ind_test)

# check correct CoM shape
assert (CoM_test.shape == CoM_test_ans.shape), "CoM matrix has incorrect shape.
 ↪\nCorrect: {}\nYours: {}".format(CoM_test.shape, CoM_test_ans.shape)

# check correct PPMI shape
assert (PPMI_test.shape == PPMI_test_ans.shape), "PPMI matrix has incorrect␣
 ↪shape.\nCorrect: {}\nYours: {}".format(PPMI_test.shape, PPMI_test_ans.shape)
# Test correct CoM and PPMI values
for w1 in word2Ind_ans.keys():
    idx1 = word2Ind_ans[w1]
    for w2 in word2Ind_ans.keys():
```

```
        idx2 = word2Ind_ans[w2]
        student1 = CoM_test[idx1, idx2]
        correct1 = CoM_test_ans[idx1, idx2]
        student2 = PPMI_test[idx1, idx2]
        correct2 = PPMI_test_ans[idx1, idx2]
        if student1 != correct1:
            print("Correct CoM:")
            print(CoM_test_ans)
            print("Your CoM: ")
            print(CoM_test)
            raise AssertionError("Incorrect count at index ({}, {})=({}, {}) in␣
 ↪matrix CoM. Yours has {} but should have {}.".format(idx1, idx2, w1, w2,␣
 ↪student1, correct1))
        if student2 != correct2:
            print("Correct PPMI:")
            print(PPMI_test_ans)
            print("Your PPMI: ")
            print(PPMI_test)
            raise AssertionError("Incorrect count at index ({}, {})=({}, {}) in␣
 ↪matrix PPMI. Yours has {} but should have {}.".format(idx1, idx2, w1, w2,␣
 ↪student2, correct2))

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

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

1.2.7 Question 1.3 [code]:

Implement "dimension_reduction" function below with python package sklearn.decomposition. For the use of PCA function and TruncatedSVD function, please refer to https://scikit-learn.org/stable/modules/classes.html#module-sklearn.decomposition

Then, run the sanity check cell to check your implementation.

[9]:
```
def dimension_reduction (X, k=2):
    '''
    params:
        X --- numpy array of shape (num_words, word_embedding_size)
        k --- int: the number of principal components that we keep
    return:
        X_reduced --- numpy array of shape (num_words, k)
                      Using TruncatedSVD algorithm when k <=␣
 ↪floor(word_embedding_size/10)
```

```
                    Using PCA algorithm when k > floor(word_embedding_size/10)
    '''
    X_reduced = None
    n_iters = 10     # Use this parameter in your call to `TruncatedSVD`
    ### YOUR CODE HERE
    _, word_embedding_size = X.shape

    if k<= np.floor(word_embedding_size/10):
        svd = TruncatedSVD(n_components = k, n_iter = n_iters)
        svd.fit(X)
        X_reduced = svd.transform(X)
    else:
        pca = PCA(k)
        X_reduced = pca.fit_transform(X)

    ### END OF YOUR CODE

    return X_reduced
```

[10]:
```
# ---------------------
# Run this sanity check
# only check that your M_reduced has the right dimensions.
# ---------------------

# Define toy corpus and run student code
test_corpus = "learn not and know not".split()
CoM_test, PPMI_test, word2Ind_test = compute_word_matrix(test_corpus,␣
 ↪window_size=1)
CoM_test_reduced = dimension_reduction(CoM_test, k=2)

# Test proper dimensions
assert (CoM_test_reduced.shape[0] == 4), "CoM_reduced has {} rows; should have␣
 ↪{}".format(CoM_test_reduced.shape[0], 4)
assert (CoM_test_reduced.shape[1] == 2), "CoM_reduced has {} columns; should␣
 ↪have {}".format(CoM_test_reduced.shape[1], 2)

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

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

1.2.8 Question 1.4 [code]:

Implement "plot_embeddings" function to visualize the word embeddings on a 2-D plane.

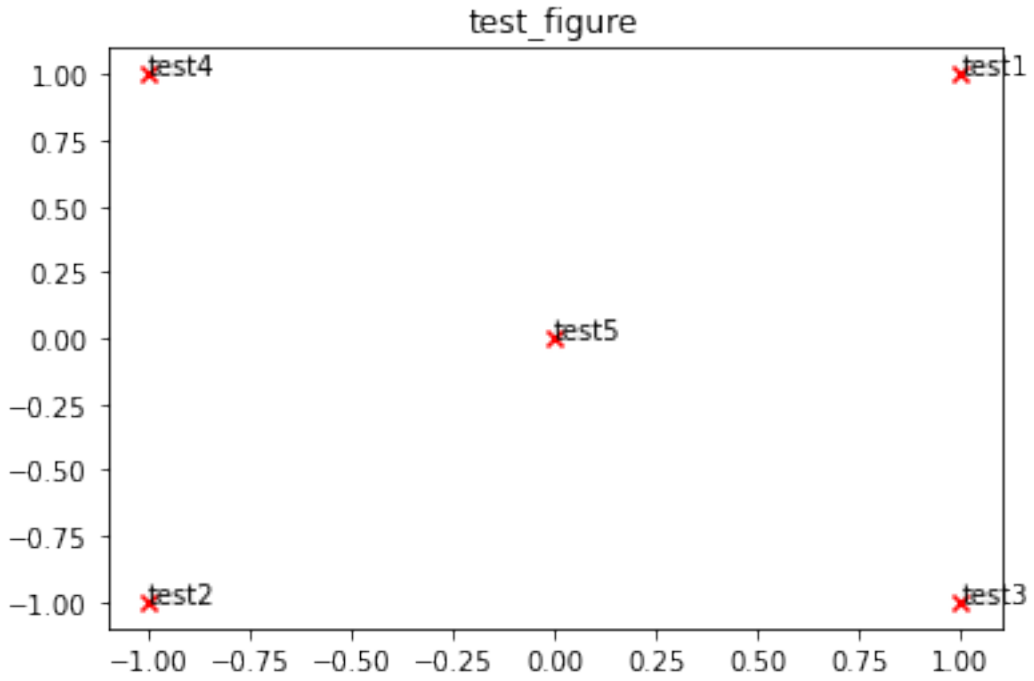
Then, run the sanity check cell to check your implementation.

[11]:
```
def plot_embeddings(X_reduced, word2Ind, words, fig_size, fig_title):
    """
    Plot in a scatterplot the embeddings of the words specified in the list␣
 ↪"words".

    params:
        X_reduced --- numpy array of shape (num_words , 2): numpy array of 2-d␣
 ↪word embeddings
        word2Ind --- dict: dictionary that maps words to indices
        words --- list[str]: a list of words of which the embeddings we want to␣
 ↪visualize
        fig_size --- tuple (a,b) : the size of figure
        fig_title --- str: title of the figure
    return:
        None
    """
    plt.figure(figsize = fig_size)
    ### YOUR CODE HERE

    index_list = [word2Ind[word] for word in words]

    x = [X_reduced[i][0] for i in index_list]
    y = [X_reduced[i][1] for i in index_list]

    for index, word in enumerate(words):
        x_coord = x[index]
        y_coord = y[index]
        plt.scatter(x_coord, y_coord, c='red', marker='x')
        plt.title(fig_title)
        plt.text(x_coord, y_coord, word)
    plt.show()

    ### END OF YOUR CODE
```

[12]:
```
# ---------------------
# 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:")

X_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(X_test, word2Ind_plot_test, words, (6,4), 'test_figure')

print ("-" * 80)
```

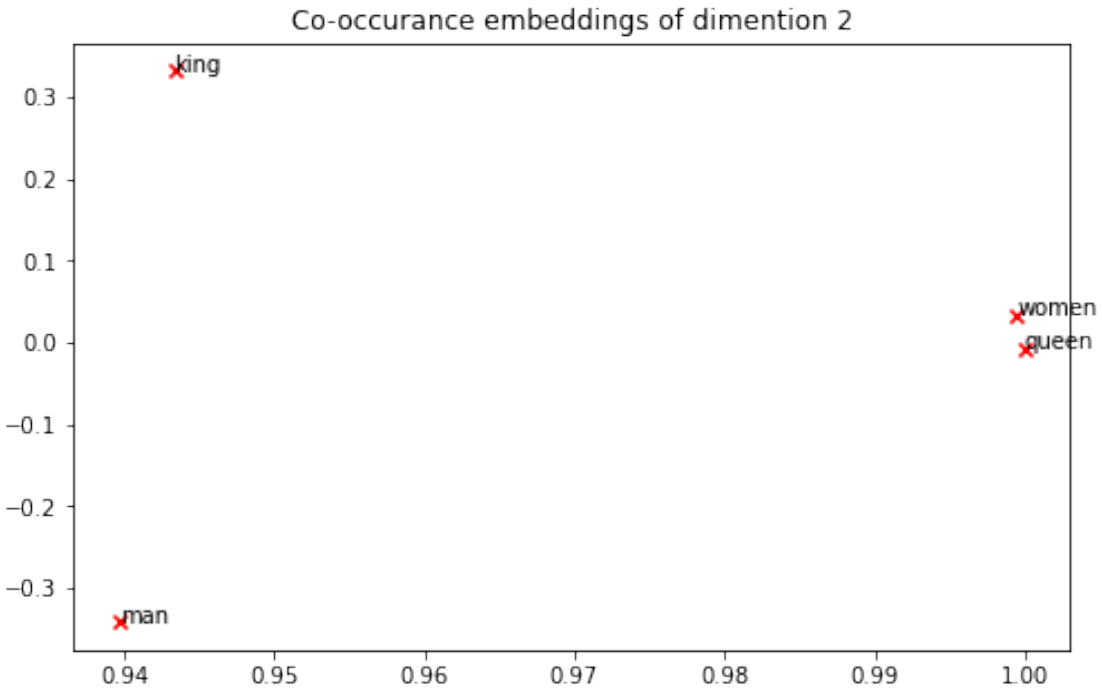
Outputted Plot:

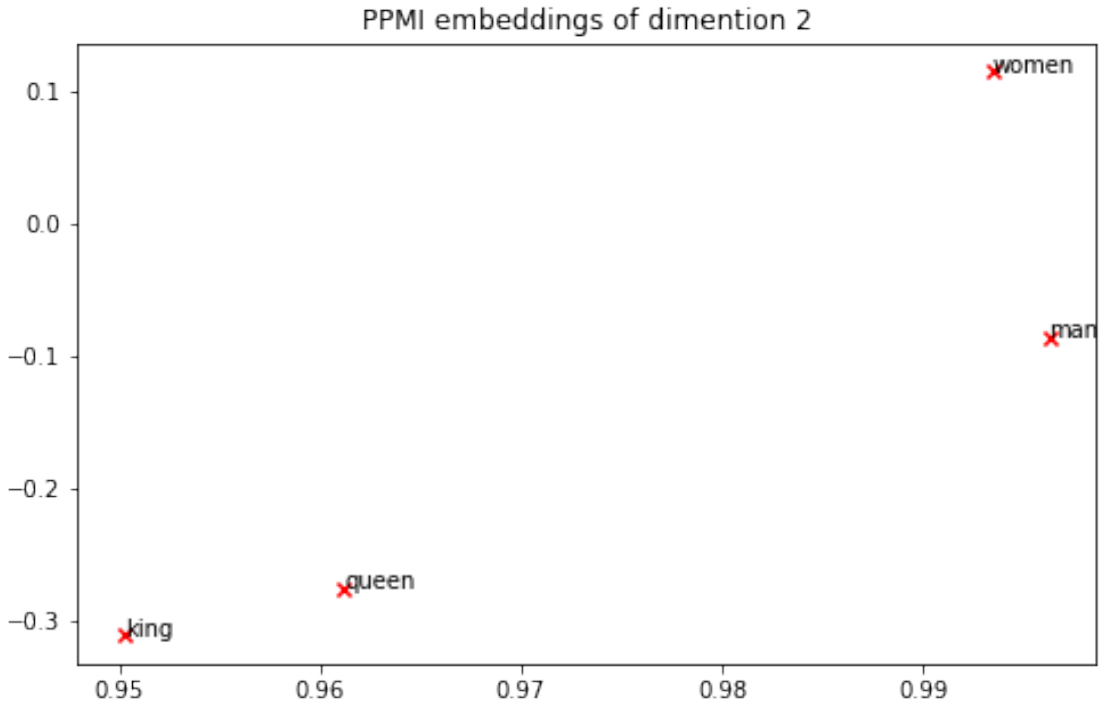


Test Plot Solution

[13]:
```
# ----------------------------
# Run This Cell to Produce Your Plot
# window_size is 3
# ----------------------------

# corpus = read_corpus(r'./data/ptb.train.txt', 50000)
corpus = read_corpus(r'./data/wikitext-2/wiki.train.tokens', 50000)

CoM, PPMI, word2Ind = compute_word_matrix(corpus, window_size=3)
CoM_reduced = dimension_reduction(CoM, k=2)
PPMI_reduced = dimension_reduction(PPMI, k=2)
```

```
# Rescale (normalize) the rows to make them each of unit-length
CoM_lengths = np.linalg.norm(CoM_reduced, axis=1)
CoM_normalized = CoM_reduced / CoM_lengths[:, np.newaxis] # broadcasting
PPMI_lengths = np.linalg.norm(PPMI_reduced, axis=1)
PPMI_normalized = PPMI_reduced / PPMI_lengths[:, np.newaxis] # broadcasting


words = ['king','women','queen','man']
plot_embeddings(CoM_normalized, word2Ind, words, (8,5), 'Co-occurance␣
 ↪embeddings of dimention 2')
plot_embeddings(PPMI_normalized, word2Ind, words, (8,5), 'PPMI embeddings of␣
 ↪dimention 2')
```





1.3 2. Prediction-Based Word Embeddings

1.3.1 Word2vec

Word2vec is a software package that contains two algorithms named CBOW and skip-gram (Mikolov 2013). In the CBOW architecture, the model predicts the current word from a window of surrounding context words. In the continuous skip-gram architecture, the model uses the current word to predict the surrounding window of context words. The architectures are shown as follows:

1.3.2 Question 2.1 [code]:

Complete the code in the function *create_word_batch*, which can be used to divide a single sequence of words into batches of words.

For example, the word sequence ["I", "like", "NLP", "So", "does", "he"] can be divided into two batches, ["I", "like", "NLP"], ["So", "does", "he"], each with batch_size=3 words. It is more efficient to train word embedding on batches of word sequences rather than on a long single sequence.

Then, run the sanity check cell to check your implementation.

[14]:
```
def create_word_batch(words, batch_size=100):
    '''
    Split the words into batches
    params:
        words --- list[str]: a list of words
```

```
        batch_size --- int: the number of words in a batch
    return:
        batch_words: list[list[str]]batches of words, list
    '''
    batch_words = []

    ### YOUR CODE HERE

#     for i in range(0,len(words), batch_size):
#         word_batch = words[i:i + batch_size]
#         batch_words.append(word_batch)
    batch_words = [words[i : i+batch_size] for i in range(0, len(words),␣
 ↪batch_size)]

    ### END OF YOUR CODE
    return batch_words
```

[15]:
```
# ------------------------------------------------
# Run this sanity check to check your implementation
# ------------------------------------------------
words_test = ["I", "like", "NLP", "So", "does", "he"]
batch_size_test = 3

ans = [["I", "like", "NLP"],["So", "does", "he"]]

batch_words_test = create_word_batch(words_test,batch_size_test)

assert ans == batch_words_test, 'your output does not match "ans"'
print('passed!')
```

```
passed!
```

1.3.3 Question 2.2 [code]:

Use "Word2Vec" function to build a word2vec model. For the use of "Word2Vec" function, please ,refer to https://radimrehurek.com/gensim/models/word2vec.html. Please use the parameters we have set for you.

It may take a few minutes to train the model.

[16]:
```
# whole_corpus = corpus = read_corpus(r'./data/ptb.train.txt', 'all')
whole_corpus = corpus = read_corpus(r'./data/wikitext-2/wiki.train.tokens',␣
 ↪'all')
batch_words = create_word_batch(whole_corpus)

vector_size = 100
min_count = 2
window = 3
```

```
sg = 1  #skip-gram algorithm

### YOUR CODE HERE
model = Word2Vec(sentences = batch_words, vector_size = vector_size, window =␣
 ↪window, min_count = min_count, sg = sg)
### END OF YOUR CODE

print('Done!')
```

```
Done!
```

1.3.4 Question 2.3 [code]:

Implement "get_word2Ind" function below first. Then, run the sanity check cell to check your implementation.

Use "get_word2Ind", "dimension_reduction", and "plot_embeddings" functions to visualize the word embeddings of the first 300 words in the vocabulary.

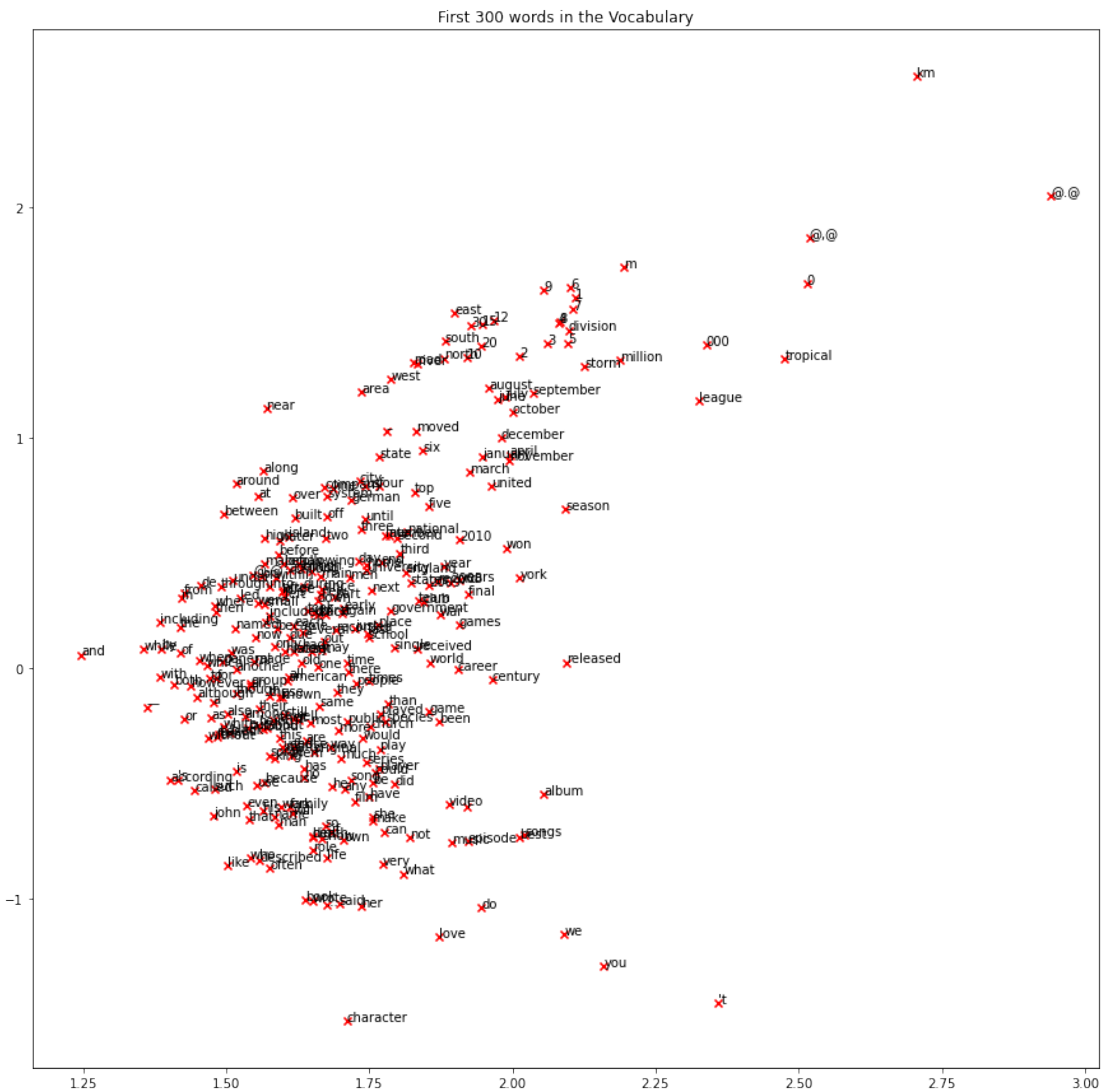
[17]:
```
def get_word2Ind(index2word):
    '''
    construct a dictionary that maps words to its index

    params:
        index2word --- list[str]: list of words
    return
        word2index --- dict: keys are words, values are the corresponding␣
 ↪indices
    '''
    word2index = dict()
    ### YOUR CODE HERE

    word2index = {word: index for index, word in enumerate(index2word)}

    ### END OF YOUR CODE
    return word2index
```

[18]:
```
# -------------------------------------------------
# Run this sanity check to check your implementation
# -------------------------------------------------
i2w_test = ['I','love','it']
ans_test = get_word2Ind(i2w_test)

ans = {'I':0, 'love':1, 'it':2}
assert ans == ans_test, 'your output did not match the correct answer.'
print('passed!')
```

```
passed!
```

[19]:

```
index2word = model.wv.index_to_key
words_to_visualize = index2word[:300]

### YOUR CODE HERE
index2word = get_word2Ind(index2word)
word_pca = dimension_reduction(model.wv.vectors, 2)

plot_embeddings(word_pca, index2word , words_to_visualize, (15, 15), "First 300␣
↪words in the Vocabulary")

### END OF YOUR CODE
```



1.3.5 Question 2.4 [code]:

1. Find the most similar words for the given words “man”,“woman”,“king”. You need to use “model.wv.most_similar” function.
2. Find out which word will it be for x in the pairs author : singer :: book : x? You need to use “model.wv.most_similar” function.

[20]:
```
words =  ['man', 'woman', 'king']
### 1
### YOUR CODE HERE

# Given each of the word:
for word in words:
    similar_word = [sw[0] for sw in model.wv.most_similar(word, topn = 5)]
    print(f"The 5 most similar words of {word} is {similar_word}")

### END OF YOUR CODE
```

```
The 5 most similar words of man is ['woman', 'dog', 'girl', 'boss', 'bearded']
The 5 most similar words of woman is ['girl', 'child', 'mother', 'grace',
'victim']
The 5 most similar words of king is ['henry', 'odaenathus', 'edward', 'lord',
'emperor']
```

[21]:
```
# ### 2
# ### YOUR CODE HERE

paired_word = model.wv.most_similar(positive=["book", "singer"],␣
 ↪negative=["author"], topn=1)
print(f"the word x would be {paired_word[0][0]}.")
# ### END OF YOUR CODE
```

```
the word x would be clarkson.
```

1.3.6 Question 2.5 [code+written]:

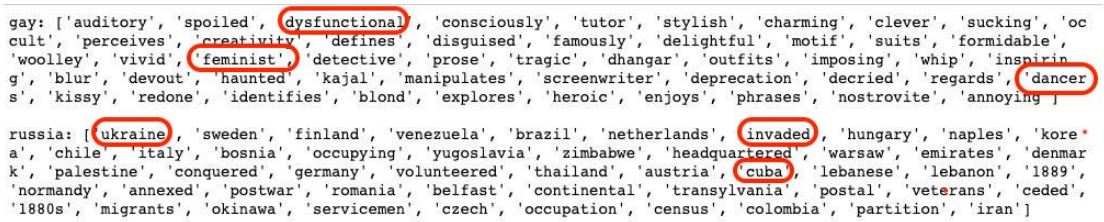
It’s important to be cognizant of the biases (gender, race, sexual orientation etc.) implicit in our word embeddings. Bias can be dangerous because it can reinforce stereotypes through applications that employ these models.

Use the `most_similar` function to find two cases where some bias is exhibited by the vectors. Please briefly explain the example of bias that you discover.

[22]:
```
### YOUR CODE HERE

gay = model.wv.most_similar(['gay'], topn = 50)
gay = [word[0] for word in gay]
# print(f"gay: {gay}\n")
```

```
russia = model.wv.most_similar(['russia'], topn = 50)
russia = [word[0] for word in russia]
# print(f"russia: {russia}\n")

### END OF YOUR CODE
```

Write your explanation:



As the weight of the embedding changes everytime I train, I did not print out the output of the word **gay** and **russia** and instead adding the screenshot of the result I got.

The first word I identified is the word **gay**, from the screenshot, we can see similar words returned such as *dysfunctional*, *feminist*, *dancers*. This is some of the common bias the society has when it comes to the discussion of **gay**, some people might think that if a guy is a ballet dancer, then he most likely is a gay. Also, gay in certain context is regarded as dysfunctional in terms of reproduction as they would not be able to give birth to the next generation.

Another word that I identified is **russia**, as we know that Russia started a war with Ukraine, and this has been captured by the word embedding, as the similar words of *ukraine* and *invaded* has been returned. On top of this, the word *cuba* has been returned, this might because of the historical event, Cuban Missile Crisis. The similar words returned by the word embedding has reflected some of the events that happened on the country. This might lead to some bias of the context of application is not on history. For example, if we use this embedding on testing scientific research context, then the words that associated with history would not be useful.