50.040 Natural Language Processing (Summer 2020) Homework 1

Due 5 June 2020, 5pm

STUDENT ID: 1003014

Name: Antonio Miguel Canlas Quizon

Students with whom you have discussed (if any):

In [1]:

```
from google.colab import drive
drive.mount('/content/gdrive')
```

```
Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6
qk8qdgf4n4g3pfee6491hc0brc4i.apps.googleusercontent.com&redirect_uri=urn%3aietf%3awg%3aoauth%3a2.0%
b&response_type=code&scope=email%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdocs.test%20https%3a%2
www.googleapis.com%2fauth%2fdrive%20https%3a%2f%2fwww.googleapis.com%2fauth%2fdrive.photos.readonly
ttps%3a%2f%2fwww.googleapis.com%2fauth%2fpeopleapi.readonly

Enter your authorization code:
··········
Mounted at /content/gdrive
```

In [2]:

```
% pwd
% ls
% cd /content/gdrive/'My Drive'/'NLP'/'HW1'
% ls
```

```
gdrive/  sample_data/
/content/gdrive/My Drive/NLP/HW1
data/  Description.pdf  homework1.ipynb  __MACOSX/
```

In [0]:

```
import numpy as np
from sklearn.decomposition import PCA
from matplotlib import pyplot as plt
from gensim.models import Word2Vec
```

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 and learn *co-occurence matrices* word embeddings and *Word2Vec* word embeddings. Note that we use "word embeddings" and "word vectors" interchangeably.

Before we start, you need to download the text8 dataset. Unzip the file and then put it under the "data" folder. The text8 dataset consists of one single line 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. Count-based word embeddings

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

The rows or columns can be used as word vectors but they are usually too large (linear in the size of the vocabulary). Thus in the next step we need to run "dimensionality reduction" algorithms like PCA, SVD.

Construct co-occurence matrix

Before you start, please make sure you have downloaded the dataset "text8" in the introduction.

In [0]:

```
def read_corpus(file_path, size=500000):
    '''
    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]
        return corpus
```

Let's have a look at the corpus

In [5]:

```
corpus = read_corpus(r'data/text8')
print(corpus[0:100])
```

```
['anarchism', 'originated', 'as', 'a', 'term', 'of', 'abuse', 'first', 'used', 'against', 'early',
'working', 'class', 'radicals', 'including', 'the', 'diggers', 'of', 'the', 'english',
'revolution', 'and', 'the', 'sans', 'culottes', 'of', 'the', 'french', 'revolution', 'whilst', 'th
e', 'term', 'is', 'still', 'used', 'in', 'a', 'pejorative', 'way', 'to', 'describe', 'any', 'act',
'that', 'used', 'violent', 'means', 'to', 'destroy', 'the', 'organization', 'of', 'society', 'it',
'has', 'also', 'been', 'taken', 'up', 'as', 'a', 'positive', 'label', 'by', 'self', 'defined',
'anarchists', 'the', 'word', 'anarchism', 'is', 'derived', 'from', 'the', 'greek', 'without',
'archons', 'ruler', 'chief', 'king', 'anarchism', 'as', 'a', 'political', 'philosophy', 'is',
'the', 'belief', 'that', 'rulers', 'are', 'unnecessary', 'and', 'should', 'be', 'abolished',
'although', 'there', 'are', 'differing',
```

```
although', 'there', 'are', 'differing']
```

Question 1 [code]:

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

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

In [0]:

```
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 in the corpus
    """
    corpus_words = None
    num_corpus_words = None
    ### You may need to use "set()" to remove duplicate words.
    ### YOUR CODE HERE (~2 lines)
    corpus_words = sorted(set(corpus))
    num_corpus_words = len(corpus_words)

    ### END OF YOUR CODE

    return corpus_words, num_corpus_words
```

In [7]:

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

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!
--------------------------------------------------------------------------------
```

Question 2 [code]:

Implement "compute_co_occurrence_matrix" that reads in "corpus" and "window_size", and returns a co-occurence matrix and a word-to-index dictionary.

Then, run the sanity check cell to check your implementation

In [0]:

```
from nltk import ngrams
```

```
def compute_co_occurrence_matrix(corpus, window_size=1):
    """
    Compute co-occurrence 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:
        M --- numpy array of shape (num_words, num_words)):
              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, num_words = distinct_words(corpus)
    M = None
    word2Ind = {}
    ###    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 sentence "learn and live" with window size of 2,
    ###    "learn" will co-occur with "and", "live".
    ###
    ### YOUR CODE HERE

    M = np.zeros((num_words, num_words))
    # grams = ngrams(corpus, window_size)
    word2Ind = {word: i for i, word in enumerate(words)}
    # print(word2Ind)

    # for i,word in enumerate(corpus):
    #     for j in range(max(i-window_size,0),min(i+window_size,num_words)):
    #       # print(i,j)
    #       if(word==corpus[j]):
    #         continue
    #       # print(word,corpus[j])
    #       # print(word2Ind[word],word2Ind[corpus[j]])
    #       M[word2Ind[word],word2Ind[corpus[j]]]+=1
    #       M[word2Ind[corpus[j]],word2Ind[word]]+=1

    for w,word in enumerate(corpus):
      target_index = word2Ind[word]
      for j in range(max(w - window_size, 0), w):
          # print(word,corpus[j])
          M[target_index][word2Ind[corpus[j]]] += 1
          M[word2Ind[corpus[j]]][target_index] += 1


    # for w,word in enumerate(corpus):
    #     curr = corpus[w]
    #     neighbors = corpus[max(0, w-window_size) : min(len(corpus), w+window_size+1)]

    #     for n in neighbors:
    #         M[word2Ind[curr]][word2Ind[n]] += 1
    #     M[word2Ind[curr]][word2Ind[curr]] -= 1


    ### END OF YOUR CODE
    return M, word2Ind
```

In [9]:

```
# ---------------------
# Run this sanity check
# ---------------------

# Define toy corpus and get co-occurrence matrix
test_corpus = "learn not and know not".split()
M_test, word2Ind_test = compute_co_occurrence_matrix(test_corpus, window_size=1)
# Correct M and word2Ind
M_test_ans = np.array(
```

```
M_test_ans = np.array(
    [[0., 1., 0., 1.],
     [1., 0., 0., 1.],
     [0., 0., 0., 1.],
     [1., 1., 1., 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 M shape
assert (M_test.shape == M_test_ans.shape), "M matrix has incorrect shape.\nCorrect: {}\nYours: {}"
.format(M_test.shape, M_test_ans.shape)

# Test correct M values
for w1 in word2Ind_ans.keys():
    idx1 = word2Ind_ans[w1]
    for w2 in word2Ind_ans.keys():
        idx2 = word2Ind_ans[w2]
        student = M_test[idx1, idx2]
        correct = M_test_ans[idx1, idx2]
        if student != correct:
            print("Correct M:")
            print(M_test_ans)
            print("Your M: ")
            print(M_test)
            raise AssertionError("Incorrect count at index ({}, {})=({}, {}) in matrix M. Yours has
{} but should have {}.".format(idx1, idx2, w1, w2, student, correct))

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

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

Question 3 [code]:

Implement "pca" function below with python package sklearn.decomposition.PCA. For the use of PCA function, please refer to https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

Then, run the sanity check cell to check your implementation

In [0]:

```
from sklearn.decomposition import PCA
def pca(X, k=2):
    '''
    A wrapper of the sklearn.decomposition.PCA function.
    params:
        X --- numpy array of shape (num_words, word_embedding_size)
        k --- int: the number of principal components that we keep
    return:
        X_pca --- numpy array of shape (num_words, k)
    '''
    X_pca = None

    ### YOUR CODE HERE (~2 line)
    pca_fx = PCA(n_components=k)
    X_pca = pca_fx.fit_transform(X)


    ### END OF YOUR CODE
    return X_pca
```

In [11]:

```
# ---------------------
# Run this sanity check
# only check that your M_reduced has the right dimensions
```

```
# 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()
M_test, word2Ind_test = compute_co_occurrence_matrix(test_corpus, window_size=1)
M_test_reduced = pca(M_test, k=2)

# Test proper dimensions
assert (M_test_reduced.shape[0] == 4), "M_reduced has {} rows; should have {}".format(M_test_reduce
d.shape[0], 4)
assert (M_test_reduced.shape[1] == 2), "M_reduced has {} columns; should have {}".format(M_test_red
uced.shape[1], 2)

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

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

Question 4 [code]:

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

In [0]:

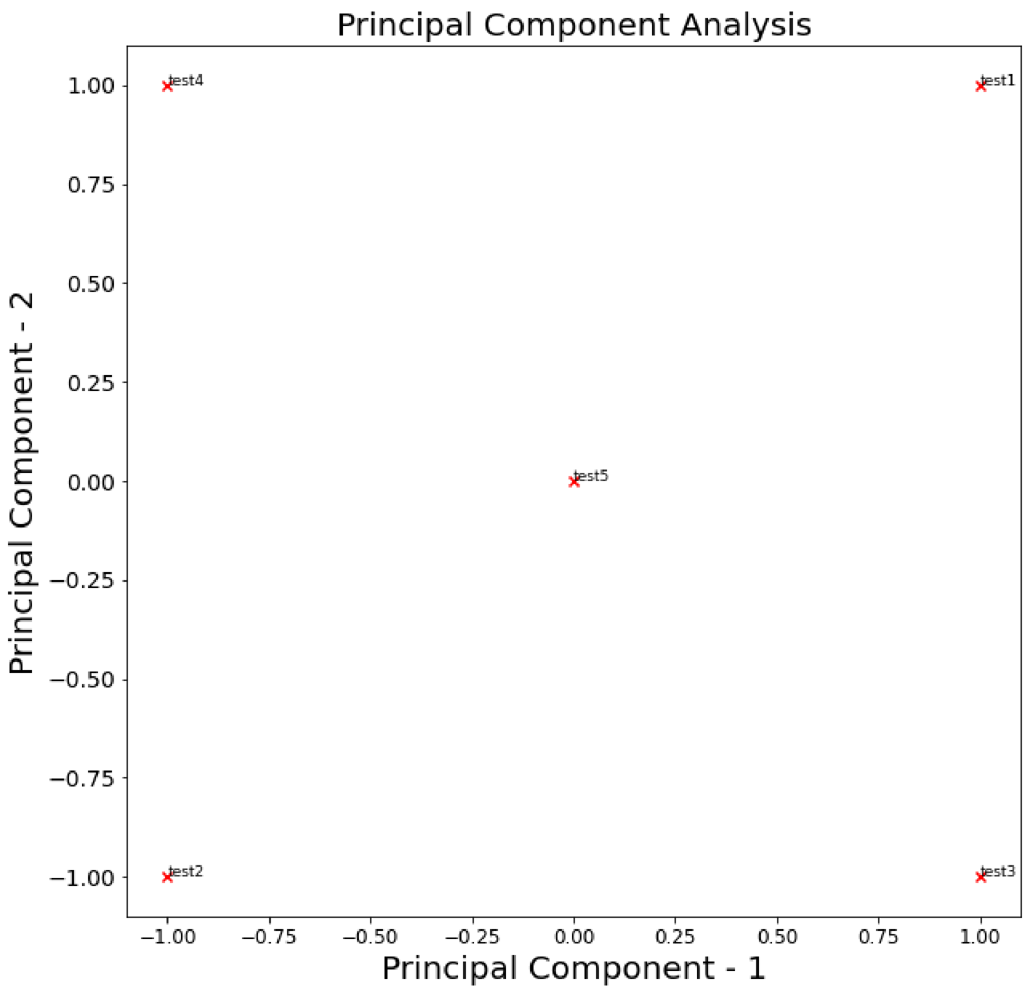
```
def plot_embeddings(X_pca, word2Ind, words):
    """
    Plot in a scatterplot the embeddings of the words specified in the list "words".

    params:
        X_pca --- 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
    return:
        None
    """
    ### You may need to use "plt.scatter", "plt.text" and a for loop here
    ### YOUR CODE HERE (~ 7 lines)
    plt.figure()
    plt.figure(figsize=(10,10))
    plt.xticks(fontsize=12)
    plt.yticks(fontsize=14)
    plt.xlabel('Principal Component - 1',fontsize=20)
    plt.ylabel('Principal Component - 2',fontsize=20)
    plt.title("Principal Component Analysis",fontsize=20)

    words_index = [word2Ind[word] for word in words]
    # print(words_index)
    xi = [X_pca[word_index][0] for word_index in words_index]
    yi = [X_pca[word_index][1] for word_index in words_index]

    for i, word in enumerate(words):
        x , y = xi[i] , yi[i]
        plt.scatter(x, y, marker = 'x', color = 'red')
        plt.text(x , y, word, fontsize = 9)
    plt.show()
    ### END OF YOUR CODE
```

In [13]:

```
# ---------------------
# 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]])
```

```python
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)

print ("-" * 80)
```

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

<Figure size 432x288 with 0 Axes>
```



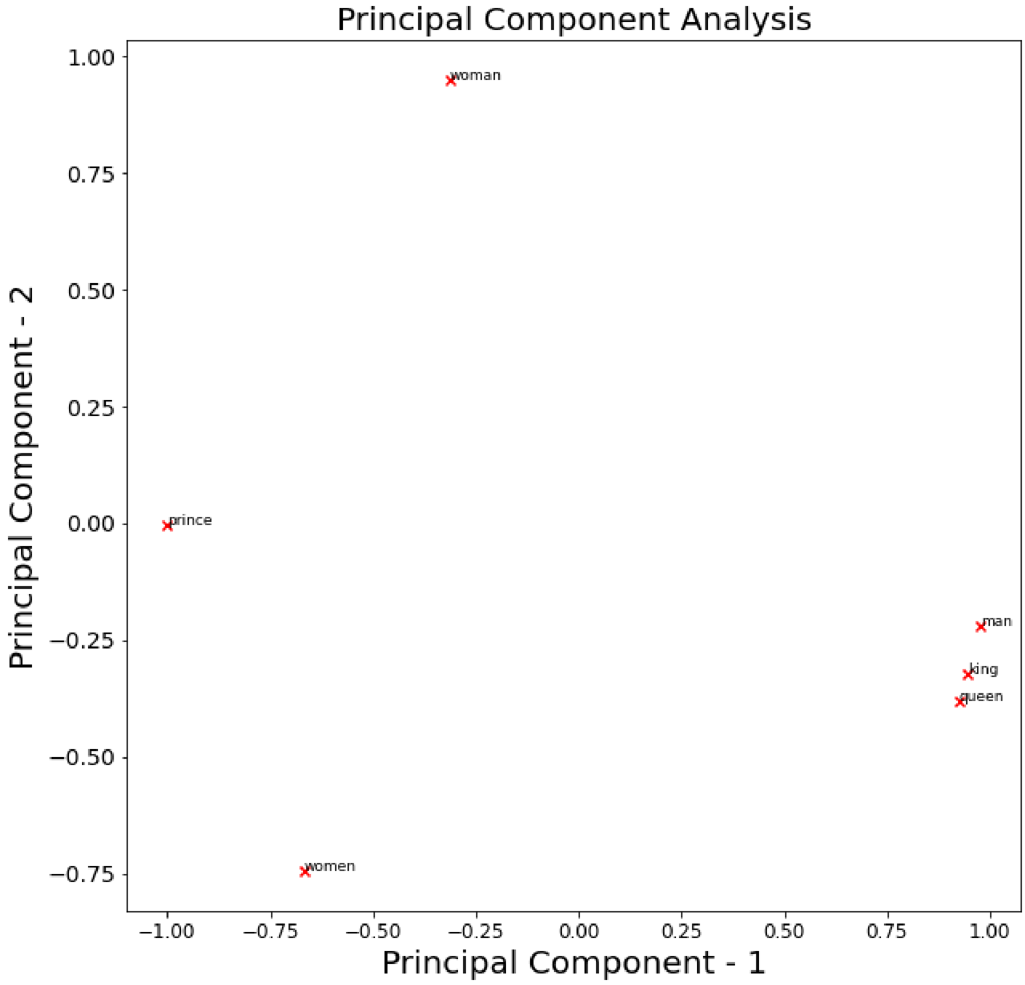

```
--------------------------------------------------------------------------------
```

Test Plot Solution

In [14]:

```python
# ------------------------------
# Run This Cell to Produce Your Plot
# ------------------------------
corpus = read_corpus(r'./data/text8',100000)
M_co_occurrence, word2Ind_co_occurrence = compute_co_occurrence_matrix(corpus, window_size=4)
M_reduced_co_occurrence = pca(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 = ['king','man','woman','women','queen','prince']
# words = ['king','man','women','prince']

plot_embeddings(M_normalized, word2Ind_co_occurrence, words)
```

```
<Figure size 432x288 with 0 Axes>
```



2. Prediction-based word embeddings

Question 5 [written]:

Given a sentence "I am interested in NLP", what will be the context and target pairs in a CBOW/Skip-gram model if the window size is 1? Write your answer in the cell below

Question 5 Answer

Given a CBOW/Skip-gram model, and window size = 1, the **target** and context pairs for the sentence are. (Target is in **bold**):

- (**I**,am)
- (**am**,I) , (**am**,interested)
- (**interested**,am) , (**interested**,in)
- (**in**,interested) , (**in**,NLP)
- (**NLP**,in)

The only difference in CBOW and Skip-gram is that CBOW uses context words to predict the target word, while Skip-gram uses the target word to predict the context word. However, the pair should be similar for both models.

Question 6 [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.

Then, run the sanity check cell to check your implementation

In [0]:

```
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
    temp = []
    for i,word in enumerate(words):
      if i%batch_size==0 and i!=0:
        batch_words.append(temp)
        temp = []
      temp.append(word)
    batch_words.append(temp)
    # print(batch_words)


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

In [16]:

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

Question 7 [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.

If you encounter "UserWarning: C extension not loaded, training will be slow", try to uninstall gensim first and then run "pip install gensim==3.6.0"

In [0]:

```
whole_corpus = corpus = read_corpus(r'./data/text8', 'all')
batch_words = create_word_batch(whole_corpus)

size = 100
min_count = 2
window = 3
sg = 1
### YOUR CODE HERE (1 line)
model = Word2Vec(sentences=batch_words, min_count=2,window=3,sg=1)
### END OF YOUR CODE
```

Question 8 [code]:

Implement "get_word2Ind" function below.

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

In [0]:

```
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: i for i, word in enumerate(index2word)}
    # print(word2index)

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

In [19]:

```
# ------------------------------------------------------
# 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!')
```

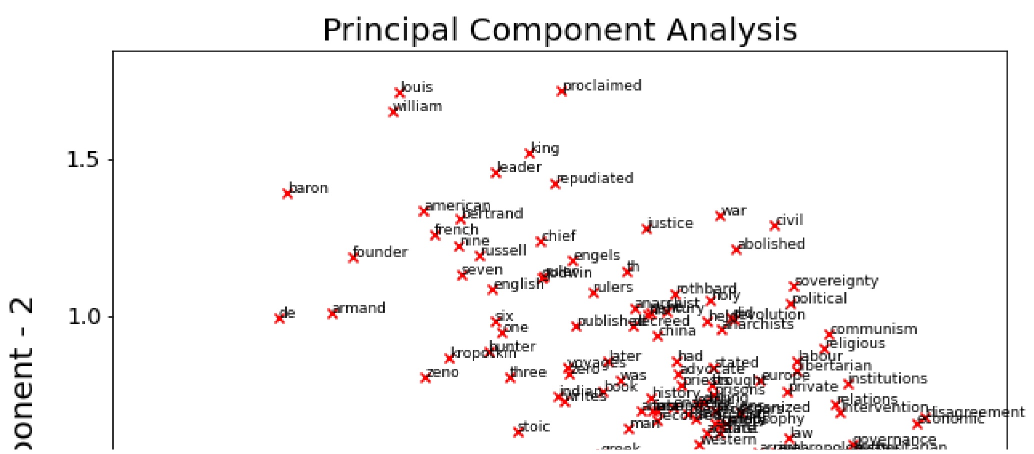
```
passed!
```

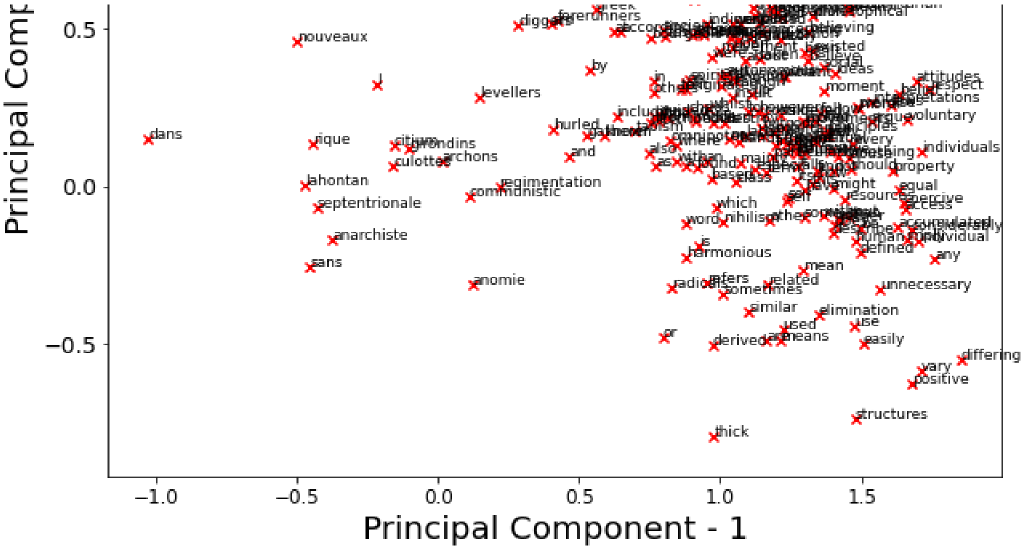
Run the cell below to visualize the word embeddings of the first 300 words in the vocabulary

In [20]:

```
word2Ind = get_word2Ind(model.wv.index2word)

vocab = model.wv.vocab
words_to_visualize = list(vocab.keys())[:300]

vec_pca = pca(model.wv.vectors, 2)

plt.figure(figsize=(15,15))
plot_embeddings(vec_pca, word2Ind, words_to_visualize)
```

```
<Figure size 1080x1080 with 0 Axes>

<Figure size 432x288 with 0 Axes>
```





Question 9:

Find the most similar words for the given words "dog","car","man". You need to use "model.wv.most_similar" function.

In [22]:

```python
words =  ['dog', 'car', 'man']

### YOUR CODE HERE (~ 2 lines)
print('---------')
print(words[0])
ret = model.wv.most_similar(words[0])
print(ret)
print('---------')
print(words[1])
ret = model.wv.most_similar(words[1])
print(ret)
print('---------')
print(words[2])
ret = model.wv.most_similar(words[2])
print(ret)
print('---------')
print(words)
ret = model.wv.most_similar(words)
print(ret)
### END OF YOUR CODE
```

```
---------
dog
[('hound', 0.7469233274459839), ('dogs', 0.7295308709144592), ('elk', 0.7271548509597778),
('bird', 0.7085689306259155), ('donkey', 0.7062799334526062), ('cat', 0.7043198943138123),
('goat', 0.7036310434341431), ('ass', 0.7024255990982056), ('leopard', 0.7015505433082581),
('winged', 0.7003122568130493)]
---------
car
[('cars', 0.7995898723602295), ('motorcycle', 0.7895153760910034), ('driver', 0.767907977104187),
('truck', 0.7624455094337463), ('automobile', 0.7545093894004822), ('vehicle',
0.7343831062316895), ('passenger', 0.7170274257659912), ('racing', 0.7120590209960938),
('airplane', 0.7079082131385803), ('cab', 0.7076858878135681)]
---------
man
[('woman', 0.7769400477409363), ('stranger', 0.7119983434677124), ('dumb', 0.7050443887710571), ('
girl', 0.6937760710716248), ('lover', 0.6895571947097778), ('person', 0.6862214803695679),
('thief', 0.6855819821357727), ('pygmalion', 0.6812896132469177), ('philia', 0.6770434379577637),
('baldrick', 0.6705169677734375)]
---------
['dog', 'car', 'man']
[('cub', 0.759698748588562), ('dumb', 0.7391194105148315), ('keeshond', 0.7349295020103455),
('pony', 0.7348888516426086), ('girl', 0.7338290810585022), ('totoro', 0.7322820425033569),
('kitten', 0.7308661937713623), ('stranger', 0.7283022403717041), ('blonde', 0.7267429828643799),
('waiter', 0.7245159149169922)]
```

```
/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of the se
cond argument of issubdtype from `int` to `np.signedinteger` is deprecated. In future, it will be
treated as `np.int64 == np.dtype(int).type`.
  if np.issubdtype(vec.dtype, np.int):
```

Question 9 Answer:

The top 10 most similar words are printed above for each word, as well as the whole sequence. The first word are the most similar to the target.

Hence, the most similar word for each entry:
(1) dog -> hound
(2) car -> cars
(3) man -> woman

Question 10 [written]:

Run the code below and explain the results in the empty cell.

In [23]:

```
model.wv.most_similar(positive=['london', 'japan'], negative=['england'])
```

```
/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of the se
cond argument of issubdtype from `int` to `np.signedinteger` is deprecated. In future, it will be
treated as `np.int64 == np.dtype(int).type`.
  if np.issubdtype(vec.dtype, np.int):
```

Out[23]:

```
[('tokyo', 0.7011764645576477),
 ('beijing', 0.6741393804550171),
 ('china', 0.6523181200027466),
 ('hong', 0.6265566349029541),
 ('kuala', 0.6204968094825745),
 ('mumbai', 0.6027482748031616),
 ('guangzhou', 0.5954842567443848),
 ('baku', 0.5950449705123901),
 ('shanghai', 0.5933279991149902),
 ('macau', 0.5920976996421814)]
```

Question 10 Answer:

Firstly, the code execution produces the top 10 most similar words given the positive and negative words. In this case, positive refers to words that contribute positively towards the similarity, and negative words negatively.

This method computes similarity is based on the cosine similarity, as euclidean distance tends to have higher similarity for higher dimensions. This results into higher similarity for words that may have different meanings. Hence, cosine similarity calculates the angle between two word vectors whereby no similarity of 0 is expressed as a 90-degree angle while the total similarity of 1 is at a 0-degree angle. Intuitively, it is the multiplication of two word vectors divided by the magnitude.

Hence as we indicate positive as 'japan' and negative as 'england', a plausible explanation is that we would likely see asian countries/cities/locations as the most similar words as they tend to coexist with very similar context. Therefore, we see asian places/locations such as korea, guangzhou, tokyo and china. Thus, tokyo and beijing are the top 2 most similar words.

In [0]: