Lab 3_Mini-project

The large number of English words can make language-based applications daunting. To cope with this, it is helpful to have a clustering or embedding of these words, so that words with similar meanings are clustered together, or have embeddings that are close to one another.

But how can we get at the meanings of words? John Firth (1957) put it thus:

You shall know a word by the company it keeps.

That is, words that tend to appear in similar contexts are likely to be related. In this assignment, you will investigate this idea by coming up with an embedding of words that is based on co-occurrence statistics.

The description here assumes you are using Python with NLTK.

In [1]:

```
%matplotlib inline
import numpy as np
import pandas as pd
import nltk
from nltk.corpus import brown
import re
from sklearn.feature_extraction.text import CountVectorizer
from nltk.tokenize import word_tokenize
from collections import Counter
import operator
import collections
import itertools
#from gensim.models import Word2Vec
from nltk.cluster import KMeansClusterer
from sklearn import cluster
from sklearn import metrics
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
from pylab import rcParams
import warnings
warnings.filterwarnings("ignore")
```

• First, download the Brown corpus (using nltk.corpus). This is a collection of text samples from a wide range of sources, with a total of over a million words. Calling brown.words() returns this text in one long list, which is useful.

In [2]:
```
nltk.download('brown')
```

```
[nltk_data] Downloading package brown to /Users/boyan/nltk_data...
[nltk_data]   Package brown is already up-to-date!
```

Out[2]:
```
True
```

In [3]:
```
nltk.corpus.brown.words()
```

Out[3]:
```
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]
```

In [4]:
```
text = []
text = nltk.corpus.brown.words()
text
```

Out[4]:
```
['The', 'Fulton', 'County', 'Grand', 'Jury', 'said', ...]
```

- **Remove stopwords and punctuation, make everything lowercase, and count how often each word occurs. Use this to come up with two lists:**

```
– A vocabulary V , consisting of a few thousand (e.g., 5000) of the
most commonly-occurring words.
– A shorter list C of at most 1000 of the most commonly-occurring
words, which we shall call context words.
```

In [5]:
```
# remove punctuations and digits
text_r = []
for i in text:
    text_r.append(re.sub("[^a-zA-Z]", " ", i))

text_r_1 = list(filter(lambda x: x.isalpha() and len(x) > 1, text_r))

df_text = pd.DataFrame(text_r_1, columns=['word'])
df_text
```

Out[5]:

	word
0	The
1	Fulton
2	County
3	Grand
4	Jury
...	...

	word
952520	the
952521	boucle
952522	dress
952523	was
952524	stupefying

952525 rows × 1 columns

In [6]:

```
# Convert upper case to lower case
text_r_l = []
for i in range(df_text.word.shape[0]):
    text_r_l.append(df_text.word[i].lower())
text_r_l = pd.DataFrame(text_r_l, columns=['word'])
text_r_l
```

Out[6]:

	word
0	the
1	fulton
2	county
3	grand
4	jury
...	...
952520	the
952521	boucle
952522	dress
952523	was
952524	stupefying

952525 rows × 1 columns

In [7]:

```
stopwords = []
nltk.download("stopwords")
stopwords = nltk.corpus.stopwords.words('english')
```

```
[nltk_data] Downloading package stopwords to /Users/boyan/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

In [8]:

```
filtered_text = [w for w in text_r_l.word if not w in stopwords]
len(filtered_text)
```

Out[8]: 508631

In [9]:

```
# count how often each word occurs.
word_count = dict(Counter(filtered_text))
sorted_words = sorted(word_count.items(), key = operator.itemgetter(1), reverse
# first 5000 most commonly-occuring words
V = [x[0] for x in sorted_words[:5000]]
C = V[:1000]
```

- For each word w ∈ V , and each occurrence of it in the text stream, look at the surrounding window of four words (two before, two after). Keep count of how often context words from C appear in these positions around word w. That is, for w ∈ V,c ∈ C, define

n(w, c) = # of times c occurs in a window around w.

Using these counts, construct the probability distribution Pr(c|w) of context words around w (for each w ∈ V), as well as the overall distribution Pr(c) of context words. These are distributions over C.

In [10]:

```
def ls_uniq(seq):
   checked = []
   for e in seq:
       if e not in checked:
           checked.append(e)
   return checked

c_words = []
for v_word in V:
    four_words = []
    positions = [x for x, n in enumerate(filtered_text) if n == v_word] # locate

    for i in positions:
        if i ==0:
            four_word = filtered_text[1:3]
        elif i == 1:
            four_word = ([filtered_text[0]] + filtered_text[2:4])
        else:
            four_word = (filtered_text[(i-2):i] + filtered_text[(i+1):(i+3)])
        four_word_uniq = ls_uniq(four_word)
        four_words = four_words + four_word_uniq

    four_words_count = dict(collections.Counter(four_words))
    window_count = len(positions)

    for c_word in four_words_count:
        if c_word in C:
            cword_fre = four_words_count[c_word]
            Pr_cw = cword_fre/window_count
            c_words.append((v_word, c_word, cword_fre, window_count, Pr_cw))

cwords = pd.DataFrame(c_words)
```

```
cwords.columns = ['V_Word','C_Word','Cword_Count','Window_Count','Pr_cw']
cwords.head()
```

Out[10]:

	V_Word	C_Word	Cword_Count	Window_Count	Pr_cw
0	one	major	12	3292	0.003645
1	one	wanted	8	3292	0.002430
2	one	wait	1	3292	0.000304
3	one	make	33	3292	0.010024
4	one	first	38	3292	0.011543

In [11]:

```
cwords_uniq = list(cwords['C_Word'].unique())
cwords_pro = {}
for cword in cwords_uniq:
    cwords_pro[cword] = filtered_text.count(cword) / len(filtered_text)
def cword_pro(x):
    return cwords_pro[x]
cwords['Pr_c'] = cwords['C_Word'].apply(cword_pro)
cwords.head()
```

Out[11]:

	V_Word	C_Word	Cword_Count	Window_Count	Pr_cw	Pr_c
0	one	major	12	3292	0.003645	0.000486
1	one	wanted	8	3292	0.002430	0.000444
2	one	wait	1	3292	0.000304	0.000185
3	one	make	33	3292	0.010024	0.001561
4	one	first	38	3292	0.011543	0.002676

- Represent each vocabulary item w by a |C|-dimensional vector Φ(w). This is known as the (positive) pointwise mutual information, and has been quite successful in work on word embeddings.

In [12]:

```
def max_log(row):
    f = row['Pr_cw']
    g = row['Pr_c']
    l = np.math.log(f/g)
    return max(0, l)
cwords['f_w'] = cwords.apply(max_log, axis = 1)
cwords.head()
```

Out[12]:

	V_Word	C_Word	Cword_Count	Window_Count	Pr_cw	Pr_c	f_w
0	one	major	12	3292	0.003645	0.000486	2.015746
1	one	wanted	8	3292	0.002430	0.000444	1.699134
2	one	wait	1	3292	0.000304	0.000185	0.496933

	V_Word	C_Word	Cword_Count	Window_Count	Pr_cw	Pr_c	f_w
3	one	make	33	3292	0.010024	0.001561	1.859652
4	one	first	38	3292	0.011543	0.002676	1.461839

In [13]:

```python
mutal_words = pd.pivot_table(cwords, index = 'V_Word', columns = 'C_Word', value
mutal_words.head()
```

Out[13]:

C_Word	able	accepted	according	account	across	act	action	activities	ac
V_Word									
abandoned	NaN	NaN	NaN	NaN	NaN	4.275155	NaN	NaN	
abel	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
ability	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
able	3.082068	NaN	NaN	3.002026	NaN	2.118753	NaN	NaN	3.01
aboard	NaN	NaN	NaN	NaN	4.278695	NaN	NaN	NaN	

5 rows × 1000 columns

In [14]:

```python
mutal_words = mutal_words.fillna(0)
mutal_words.head()
```

Out[14]:

C_Word	able	accepted	according	account	across	act	action	activities	a
V_Word									
abandoned	0.000000	0.0	0.0	0.000000	0.000000	4.275155	0.0	0.0	0.0
abel	0.000000	0.0	0.0	0.000000	0.000000	0.000000	0.0	0.0	0.0
ability	0.000000	0.0	0.0	0.000000	0.000000	0.000000	0.0	0.0	0.0
able	3.082068	0.0	0.0	3.002026	0.000000	2.118753	0.0	0.0	3.0
aboard	0.000000	0.0	0.0	0.000000	4.278695	0.000000	0.0	0.0	0.0

5 rows × 1000 columns

In [15]:

```python
from sklearn.decomposition import TruncatedSVD
from sklearn.random_projection import sparse_random_matrix
X = np.asarray(mutal_words)
svd = TruncatedSVD(n_components = 100, n_iter = 7, random_state = 42)
svd.fit(X)
X_reduce = svd.fit_transform(X)
X_reduce_df = pd.DataFrame(X_reduce, index = mutal_words.index)
X_reduce_df.head()
```

Out[15]:

	0	1	2	3	4	5	6	
V_Word								
abandoned	5.960956	-0.799199	-0.443532	0.477622	0.188357	-0.007875	-1.262637	-0.626

V_Word	0	1	2	3	4	5	6	
abel	5.741608	5.306603	1.364197	0.465327	0.085904	-0.433129	-1.344058	1.1268
ability	13.572285	-3.672069	-6.581054	1.495166	0.523935	-1.316302	0.390778	-2.596
able	22.168586	-0.288742	-4.411252	-0.904856	2.287316	-2.656532	2.975976	-2.467
aboard	6.606274	3.376979	1.393158	-0.672356	-0.731437	0.684725	-1.115608	0.326

5 rows × 100 columns

In [16]:

```
from sklearn.metrics.pairwise import cosine_similarity as cs
words_similarity = 1 - cs(X_reduce, X_reduce)
words_similarity_df = pd.DataFrame(words_similarity, index =
mutal_words.index,columns = mutal_words.index)
np.fill_diagonal(words_similarity_df.values, 1)
words_similarity_df.head()
```

Out[16]:

V_Word	abandoned	abel	ability	able	aboard	abroad	abrupt	absence
abandoned	1.000000	0.968543	0.730406	0.533748	0.774131	0.693942	0.911359	0.725871
abel	0.968543	1.000000	0.860558	0.759046	0.618207	0.790844	0.780160	0.872887
ability	0.730406	0.860558	1.000000	0.464315	0.869166	0.626037	0.544747	0.615985
able	0.533748	0.759046	0.464315	1.000000	0.603940	0.513003	0.571522	0.521385
aboard	0.774131	0.618207	0.869166	0.603940	1.000000	0.682407	0.842458	0.742596

5 rows × 5000 columns

(b) Nearest neighbor results.

In [17]:

```
# according to similarity matrix to find the cloest meaning word
words_list = ['communism', 'autumn', 'cigarette', 'pulmonary', 'mankind', 'afric
words_similarity_dict = {}
for word_list in words_list:
    words_similarity_dict[word_list] = words_similarity_df[word_list].idxmin()
for word_dict in words_similarity_dict:
    print ('{} ------> {}'.format(word_dict, words_similarity_dict[word_dict]))
```

```
communism ------> century
autumn ------> summer
cigarette ------> wet
pulmonary ------> artery
mankind ------> world
africa ------> asia
chicago ------> portland
revolution ------> world
september ------> december
chemical ------> feed
detergent ------> fabrics
dictionary ------> text
```

```
storm ------> weekend
worship ------> community
```

Yes, the results make sense!

(c) Clustering.

Using the vectorial representation Ψ(·), cluster the words in V into 100 groups. Clearly specify what algorithm and distance function you using for this, and the reasons for your choices. Look over the resulting 100 clusters. Do any of them seem even moderately coherent? Pick out a few of the best clusters and list the words in them.

Algorithm: KMeansCluster

Distance function: nltk.cluster.util.cosine_distance

Reasons: KMeans works iteractively, where initially each centroid is placed randomly in the vector space of the dataset and move themselves to the center of the points which are closer to them. In each new iteration the distance between each centroid and the points are recalculated and the centroids move again to the center of the closest points. The algorithm is finished when the position or the groups don't change anymore or when the distance in which the centroids change doesn't surpass a pre-defined threshold.

In [18]:

```
kclusterer = KMeansClusterer(100, distance=nltk.cluster.util.cosine_distance, re
assigned_clusters = kclusterer.cluster(X_reduce, assign_clusters=True)
```

Yes, some of them seem even moderately coherent.

In [19]:

```
# Pick out a few of the best clusters and list the words in them.
strings = []
for index, i in enumerate(X_reduce):
    strings.append(str(assigned_clusters[index]) + ":" + X_reduce_df.index[assig

print(strings[:10])
```

```
['42:across', '89:affairs', '34:accused', '97:afternoon', '96:african', '87:af',
'18:acceptance', '23:accompanied', '58:added', '41:acres']
```

In [ ]:

```
```