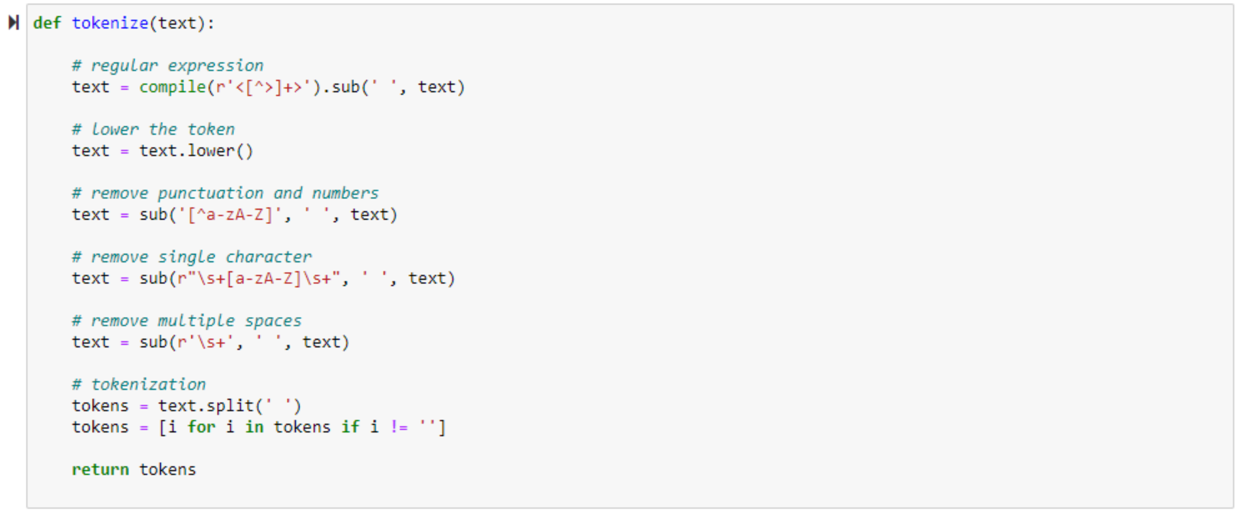
1. Getting used to handling language data.

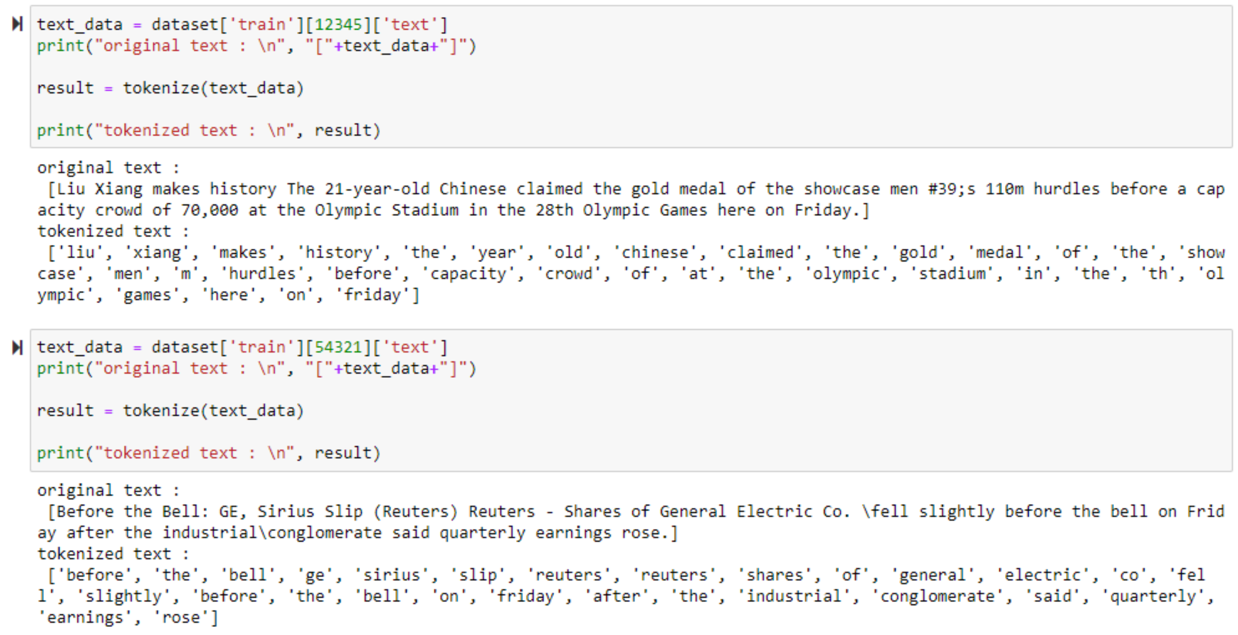
Task 1.1 – Implement the tokenize function in Vocabulary.py.



```python
def tokenize(text):

    # regular expression
    text = compile(r'<[^>]+>').sub(' ', text)

    # lower the token
    text = text.lower()

    # remove punctuation and numbers
    text = sub('[^a-zA-Z]', ' ', text)

    # remove single character
    text = sub(r"\s+[a-zA-Z]\s+", ' ', text)

    # remove multiple spaces
    text = sub(r'\s+', ' ', text)

    # tokenization
    tokens = text.split(' ')
    tokens = [i for i in tokens if i != '']

    return tokens
```

Figure 1. function: tokenize

Figure 1 is the tokenization function I implemented. First, I changed the text to regular expression, then change the upper case letters into lower case, then removed the punctuation and numbers then removed a single character. Since I replaced those punctuations, numbers into space, I removed the multiple spaces. Finally, I split the text with space and removed the unnecessary space.



```python
text_data = dataset['train'][12345]['text']
print("original text : \n", "["+text_data+"]")

result = tokenize(text_data)

print("tokenized text : \n", result)
```

```
original text :
 [Liu Xiang makes history The 21-year-old Chinese claimed the gold medal of the showcase men #39;s 110m hurdles before a cap
acity crowd of 70,000 at the Olympic Stadium in the 28th Olympic Games here on Friday.]
tokenized text :
 ['liu', 'xiang', 'makes', 'history', 'the', 'year', 'old', 'chinese', 'claimed', 'the', 'gold', 'medal', 'of', 'the', 'show
case', 'men', 'm', 'hurdles', 'before', 'capacity', 'crowd', 'of', 'at', 'the', 'olympic', 'stadium', 'in', 'the', 'th', 'ol
ympic', 'games', 'here', 'on', 'friday']
```

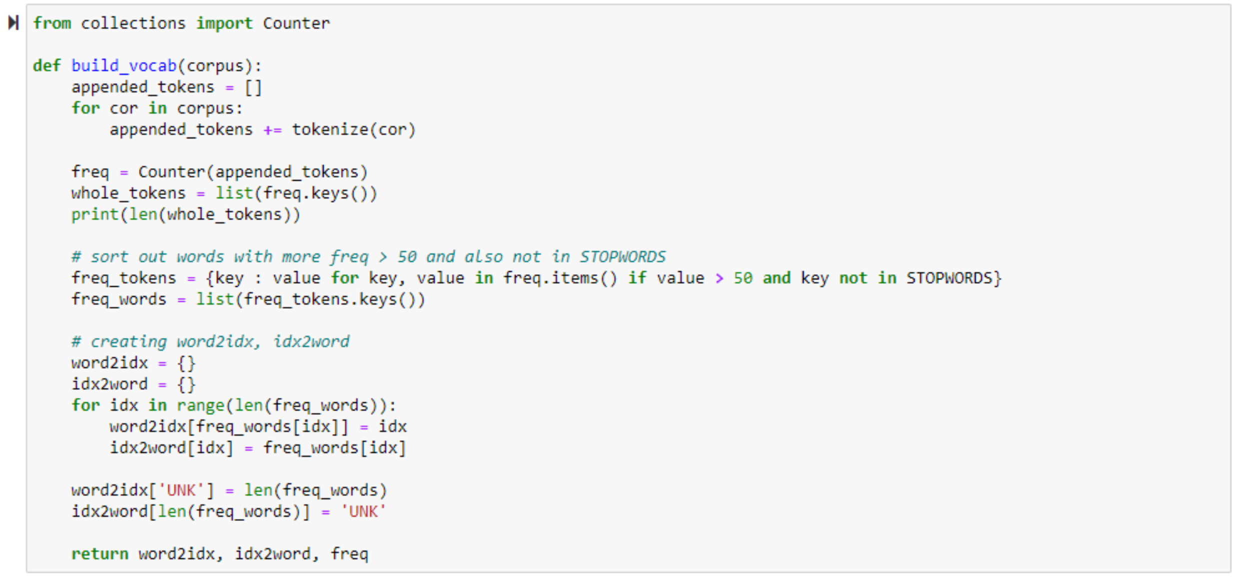
```python
text_data = dataset['train'][54321]['text']
print("original text : \n", "["+text_data+"]")

result = tokenize(text_data)

print("tokenized text : \n", result)
```

```
original text :
 [Before the Bell: GE, Sirius Slip (Reuters) Reuters - Shares of General Electric Co. \fell slightly before the bell on Frid
ay after the industrial\conglomerate said quarterly earnings rose.]
tokenized text :
 ['before', 'the', 'bell', 'ge', 'sirius', 'slip', 'reuters', 'reuters', 'shares', 'of', 'general', 'electric', 'co', 'fel
l', 'slightly', 'before', 'the', 'bell', 'on', 'friday', 'after', 'the', 'industrial', 'conglomerate', 'said', 'quarterly',
'earnings', 'rose']
```

Figure 2. tokenization examples

Figure 2 are examples of input and output of tokenize() function.

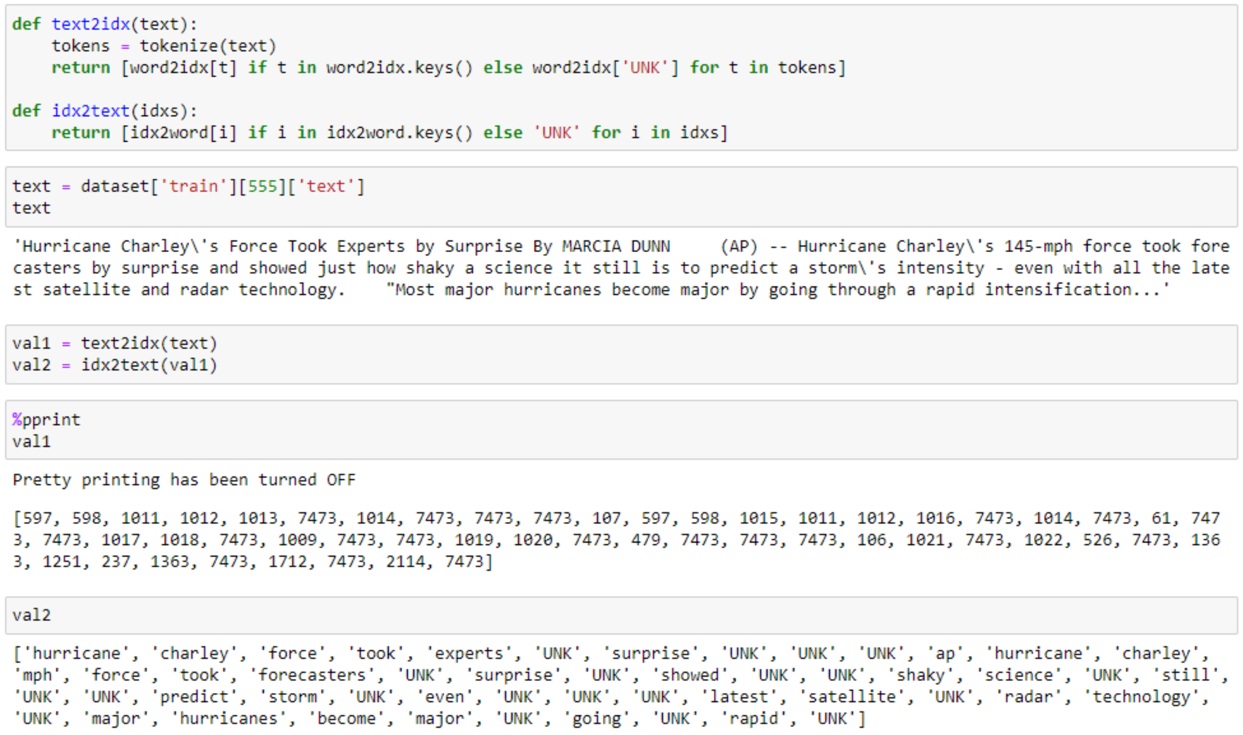
Task 1.2 – Implement the build_vocab function in Vocabulary.py.



```
from collections import Counter

def build_vocab(corpus):
    appended_tokens = []
    for cor in corpus:
        appended_tokens += tokenize(cor)

    freq = Counter(appended_tokens)
    whole_tokens = list(freq.keys())
    print(len(whole_tokens))

    # sort out words with more freq > 50 and also not in STOPWORDS
    freq_tokens = {key : value for key, value in freq.items() if value > 50 and key not in STOPWORDS}
    freq_words = list(freq_tokens.keys())

    # creating word2idx, idx2word
    word2idx = {}
    idx2word = {}
    for idx in range(len(freq_words)):
        word2idx[freq_words[idx]] = idx
        idx2word[idx] = freq_words[idx]

    word2idx['UNK'] = len(freq_words)
    idx2word[len(freq_words)] = 'UNK'

    return word2idx, idx2word, freq
```

Figure 3. build_vocab function

Figure 3 is the implementation of build_vocab. To create word2idx and idx2word, I chose words that appeared more than 50 times and also not in the stopwords; stopwords are from nltk.download('stopwords') and used set(stopwords.words('english')).



```
def text2idx(text):
    tokens = tokenize(text)
    return [word2idx[t] if t in word2idx.keys() else word2idx['UNK'] for t in tokens]

def idx2text(idxs):
    return [idx2word[i] if i in idx2word.keys() else 'UNK' for i in idxs]
```

```
text = dataset['train'][555]['text']
text
```

```
'Hurricane Charley\'s Force Took Experts by Surprise By MARCIA DUNN     (AP) -- Hurricane Charley\'s 145-mph force took fore
casters by surprise and showed just how shaky a science it still is to predict a storm\'s intensity - even with all the late
st satellite and radar technology.    "Most major hurricanes become major by going through a rapid intensification...'
```

```
val1 = text2idx(text)
val2 = idx2text(val1)
```

```
%pprint
val1
```

```
Pretty printing has been turned OFF

[597, 598, 1011, 1012, 1013, 7473, 1014, 7473, 7473, 7473, 107, 597, 598, 1015, 1011, 1012, 1016, 7473, 1014, 7473, 61, 747
3, 7473, 1017, 1018, 7473, 1009, 7473, 7473, 1019, 1020, 7473, 479, 7473, 7473, 7473, 106, 1021, 7473, 1022, 526, 7473, 136
3, 1251, 237, 1363, 7473, 1712, 7473, 2114, 7473]
```

```
val2
```

```
['hurricane', 'charley', 'force', 'took', 'experts', 'UNK', 'surprise', 'UNK', 'UNK', 'UNK', 'ap', 'hurricane', 'charley',
'mph', 'force', 'took', 'forecasters', 'UNK', 'surprise', 'UNK', 'showed', 'UNK', 'UNK', 'shaky', 'science', 'UNK', 'still',
'UNK', 'UNK', 'predict', 'storm', 'UNK', 'even', 'UNK', 'UNK', 'UNK', 'latest', 'satellite', 'UNK', 'radar', 'technology',
'UNK', 'major', 'hurricanes', 'become', 'major', 'UNK', 'going', 'UNK', 'rapid', 'UNK']
```

Figure 4. example result from build_vocab

Since the whole result of word2idx is too long to show, I chose text2idx and idx2text results from one example text.

Task 1.3 – Implement the make_vocab_charts function in Vocabulary.py.

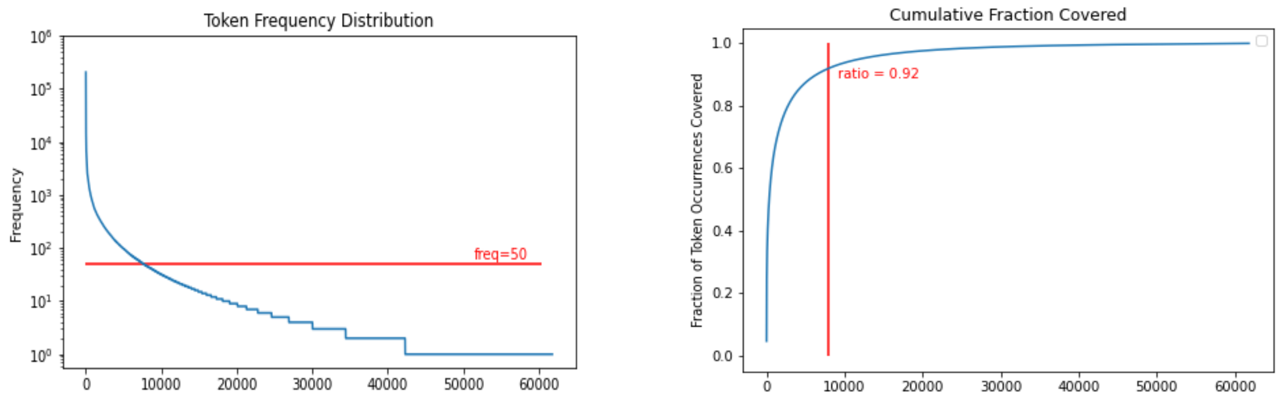
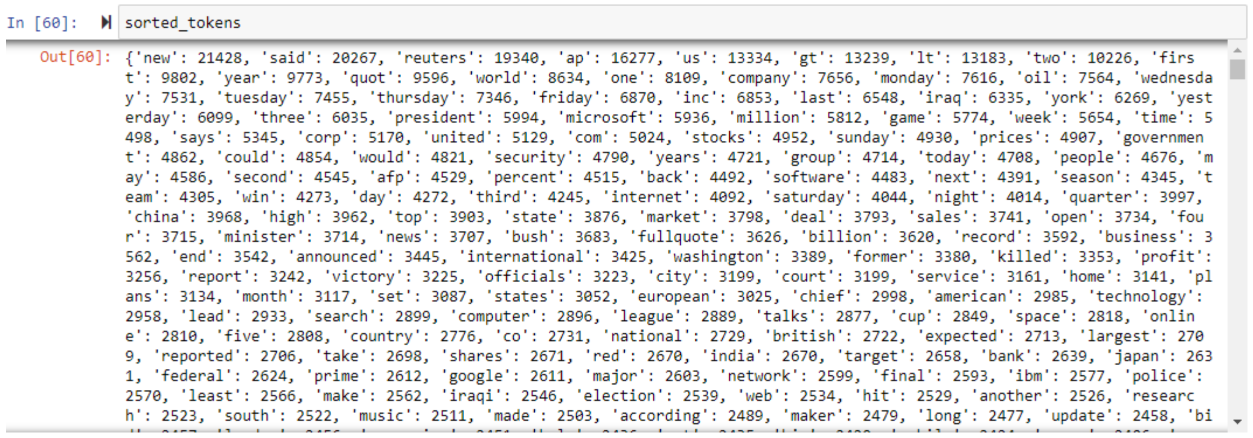


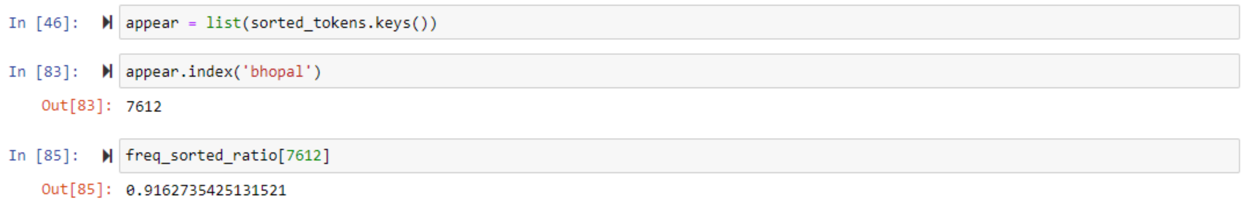
Figure 5. plot results of make_vocab_charts()

The above two plots are the results of the make_vocab_charts function.



```
In [60]:  sorted_tokens

Out[60]: {'new': 21428, 'said': 20267, 'reuters': 19340, 'ap': 16277, 'us': 13334, 'gt': 13239, 'lt': 13183, 'two': 10226, 'firs
         t': 9802, 'year': 9773, 'quot': 9596, 'world': 8634, 'one': 8109, 'company': 7656, 'monday': 7616, 'oil': 7564, 'wednesda
         y': 7531, 'tuesday': 7455, 'thursday': 7346, 'friday': 6870, 'inc': 6853, 'last': 6548, 'iraq': 6335, 'york': 6269, 'yest
         erday': 6099, 'three': 6035, 'president': 5994, 'microsoft': 5936, 'million': 5812, 'game': 5774, 'week': 5654, 'time': 5
         498, 'says': 5345, 'corp': 5170, 'united': 5129, 'com': 5024, 'stocks': 4952, 'sunday': 4930, 'prices': 4907, 'governmen
         t': 4862, 'could': 4854, 'would': 4821, 'security': 4790, 'years': 4721, 'group': 4714, 'today': 4708, 'people': 4676, 'm
         ay': 4586, 'second': 4545, 'afp': 4529, 'percent': 4515, 'back': 4492, 'software': 4483, 'next': 4391, 'season': 4345, 't
         eam': 4305, 'win': 4273, 'day': 4272, 'third': 4245, 'internet': 4092, 'saturday': 4044, 'night': 4014, 'quarter': 3997,
         'china': 3968, 'high': 3962, 'top': 3903, 'state': 3876, 'market': 3798, 'deal': 3793, 'sales': 3741, 'open': 3734, 'fou
         r': 3715, 'minister': 3714, 'news': 3707, 'bush': 3683, 'fullquote': 3626, 'billion': 3620, 'record': 3592, 'business': 3
         562, 'end': 3542, 'announced': 3445, 'international': 3425, 'washington': 3389, 'former': 3380, 'killed': 3353, 'profit':
         3256, 'report': 3242, 'victory': 3225, 'officials': 3223, 'city': 3199, 'court': 3199, 'service': 3161, 'home': 3141, 'pl
         ans': 3134, 'month': 3117, 'set': 3087, 'states': 3052, 'european': 3025, 'chief': 2998, 'american': 2985, 'technology':
         2958, 'lead': 2933, 'search': 2899, 'computer': 2896, 'league': 2889, 'talks': 2877, 'cup': 2849, 'space': 2818, 'onlin
         e': 2810, 'five': 2808, 'country': 2776, 'co': 2731, 'national': 2729, 'british': 2722, 'expected': 2713, 'largest': 270
         9, 'reported': 2706, 'take': 2698, 'shares': 2671, 'red': 2670, 'india': 2670, 'target': 2658, 'bank': 2639, 'japan': 263
         1, 'federal': 2624, 'prime': 2612, 'google': 2611, 'major': 2603, 'network': 2599, 'final': 2593, 'ibm': 2577, 'police':
         2570, 'least': 2566, 'make': 2562, 'iraqi': 2546, 'election': 2539, 'web': 2534, 'hit': 2529, 'another': 2526, 'researc
         h': 2523, 'south': 2522, 'music': 2511, 'made': 2503, 'according': 2489, 'maker': 2479, 'long': 2477, 'update': 2458, 'bi
```

This is the sorted frequency result without stopwords. I removed the stopwords because articles and other be-verbs seem not important features when doing NLP. So, if we see the sorted frequency result, if we ignore the top 25 ~ 30 words, their frequency is above 5000. So I chose 0.01 * 5000 = 50 as heuristic cutoff.



```
In [46]:  appear = list(sorted_tokens.keys())

In [83]:  appear.index('bhopal')

Out[83]:  7612

In [85]:  freq_sorted_ratio[7612]

Out[85]:  0.9162735425131521
```

For the cumulative fraction, the last word that appeared more than 50 is ‘Bhopal’, and the index of that word was 7612 and its cumulative ratio was 0.916 which is round up to 92. So I chose that 92 percent as my cutoff.

2. Frequency-Based Word Vectors – PPMI

Task 2.1

$$\text{PMI}(w_i, w_j) = \log \frac{p(w_i, w_j)}{p(w_i)p(w_j)}$$

1) What are the minimum and maximum values of PMI?

- minimum value is negative infinity, and the maximum value is $\min\left(-\log p(x) - logp(j)\right)$

2) If two tokens have a positive PMI, what does that imply about their relationship?

- positive PMI means two tokens co-occur more frequently than would be expected.

3) If they have a PMI of zero?

- zero PMI value means two tokens are statistically independent.

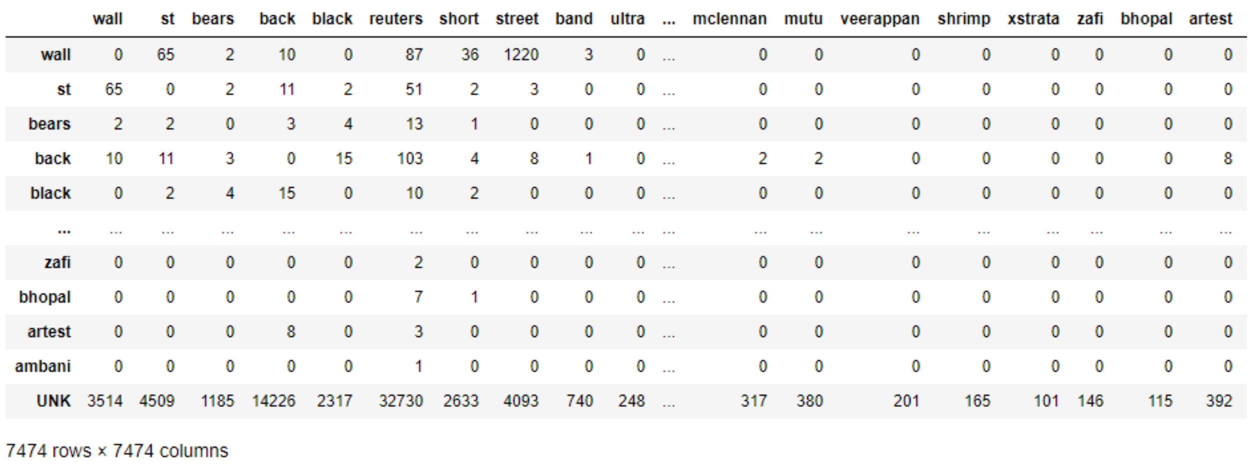
4) What about if the PMI is negative?

- negative PMI value means two tokens co-occur less frequently than would be expected.

5) What is an intuition of the use of PPMI?

- when co-occurrence of base word w and context word c is zero, $\log(0) = -\infty$, goes to negative infinite, it is hard to define, and co-occurrence matrix is a sparse matrix with lots of zero, so instead of using PMI, we use PPMI.

Task 2.2 – Implement the compute_cooccurrence_matrix function.



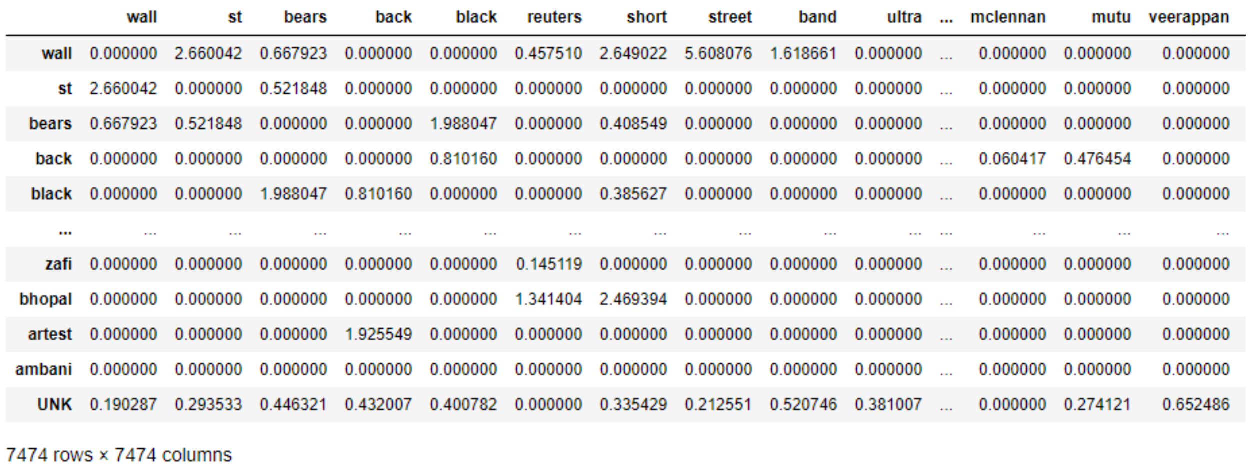
	wall	st	bears	back	black	reuters	short	street	band	ultra	...	mclennan	mutu	veerappan	shrimp	xstrata	zafi	bhopal	artest
wall	0	65	2	10	0	87	36	1220	3	0	...	0	0	0	0	0	0	0	0
st	65	0	2	11	2	51	2	3	0	0	...	0	0	0	0	0	0	0	0
bears	2	2	0	3	4	13	1	0	0	0	...	0	0	0	0	0	0	0	0
back	10	11	3	0	15	103	4	8	1	0	...	2	2	0	0	0	0	0	8
black	0	2	4	15	0	10	2	0	0	0	...	0	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
zafi	0	0	0	0	0	2	0	0	0	0	...	0	0	0	0	0	0	0	0
bhopal	0	0	0	0	0	7	1	0	0	0	...	0	0	0	0	0	0	0	0
artest	0	0	0	8	0	3	0	0	0	0	...	0	0	0	0	0	0	0	0
ambani	0	0	0	0	0	1	0	0	0	0	...	0	0	0	0	0	0	0	0
UNK	3514	4509	1185	14226	2317	32730	2633	4093	740	248	...	317	380	201	165	101	146	115	392

7474 rows × 7474 columns

Figure 6. Co-occurrence Matrix

Figure 6 is the result of the co-occurrence matrix with the whole dataset. I have chosen the window size into 4, and it is not including the index word, so the context size is 5. I chose this window size because the whole dataset and each text are long, and some words do come along with each other a lot but not next to each other. For example, if we look at the phrase 'throw out the trash', 'bank located in wall street', 'throw and trash', 'bank and wall street' are co-occur a lot but they are not located next to each other. Therefore I decided on a window size into the size of the chunk which is usually 4 or 5 words.

Task 2.3 – Implement the compute_ppmi_matrix function.



	wall	st	bears	back	black	reuters	short	street	band	ultra	...	mclennan	mutu	veerappan
wall	0.000000	2.660042	0.667923	0.000000	0.000000	0.457510	2.649022	5.608076	1.618661	0.000000	...	0.000000	0.000000	0.000000
st	2.660042	0.000000	0.521848	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000
bears	0.667923	0.521848	0.000000	0.000000	1.988047	0.000000	0.408549	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000
back	0.000000	0.000000	0.000000	0.000000	0.810160	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.060417	0.476454	0.000000
black	0.000000	0.000000	1.988047	0.810160	0.000000	0.000000	0.385627	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
zafi	0.000000	0.000000	0.000000	0.000000	0.000000	0.145119	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000
bhopal	0.000000	0.000000	0.000000	0.000000	0.000000	1.341404	2.469394	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000
artest	0.000000	0.000000	0.000000	1.925549	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000
ambani	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.000000	0.000000
UNK	0.190287	0.293533	0.446321	0.432007	0.400782	0.000000	0.335429	0.212551	0.520746	0.381007	...	0.000000	0.274121	0.652486

7474 rows × 7474 columns

Figure 7. PPMI matrix result

Task 2.4 – Run build_freq_vectors.py and examine the visualized word vectors.

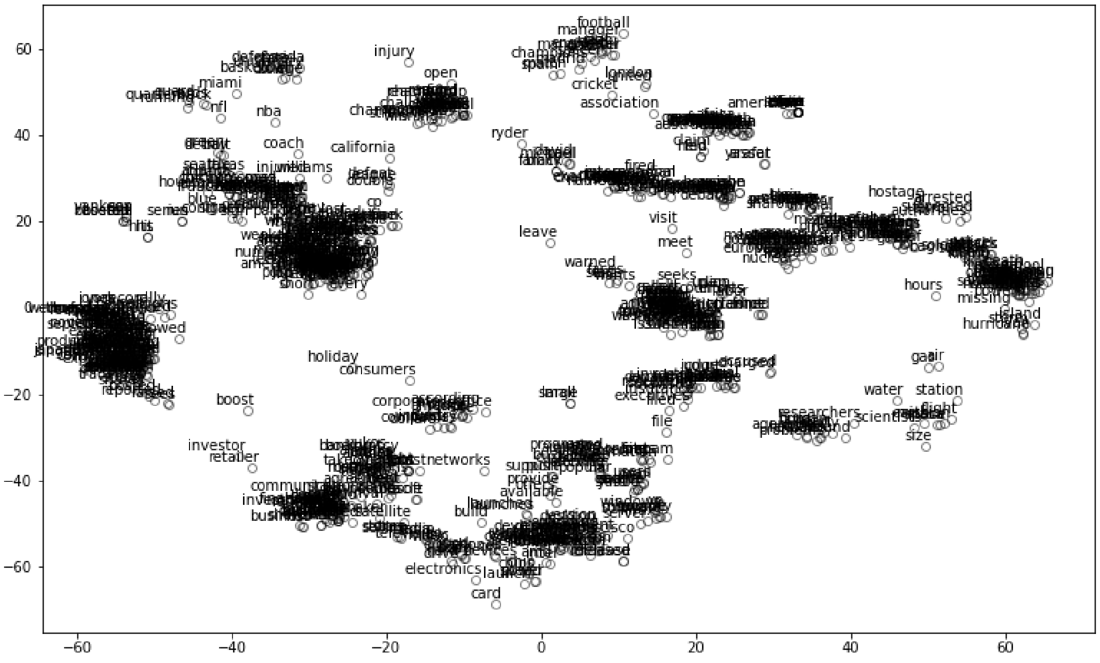
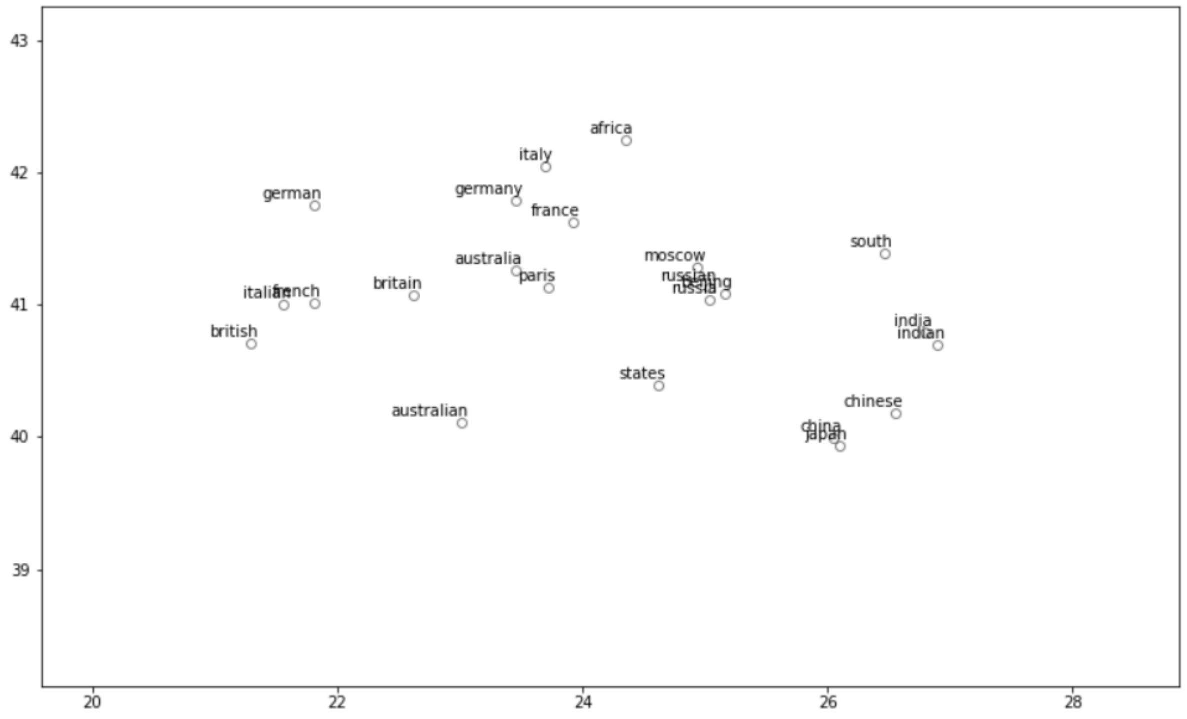
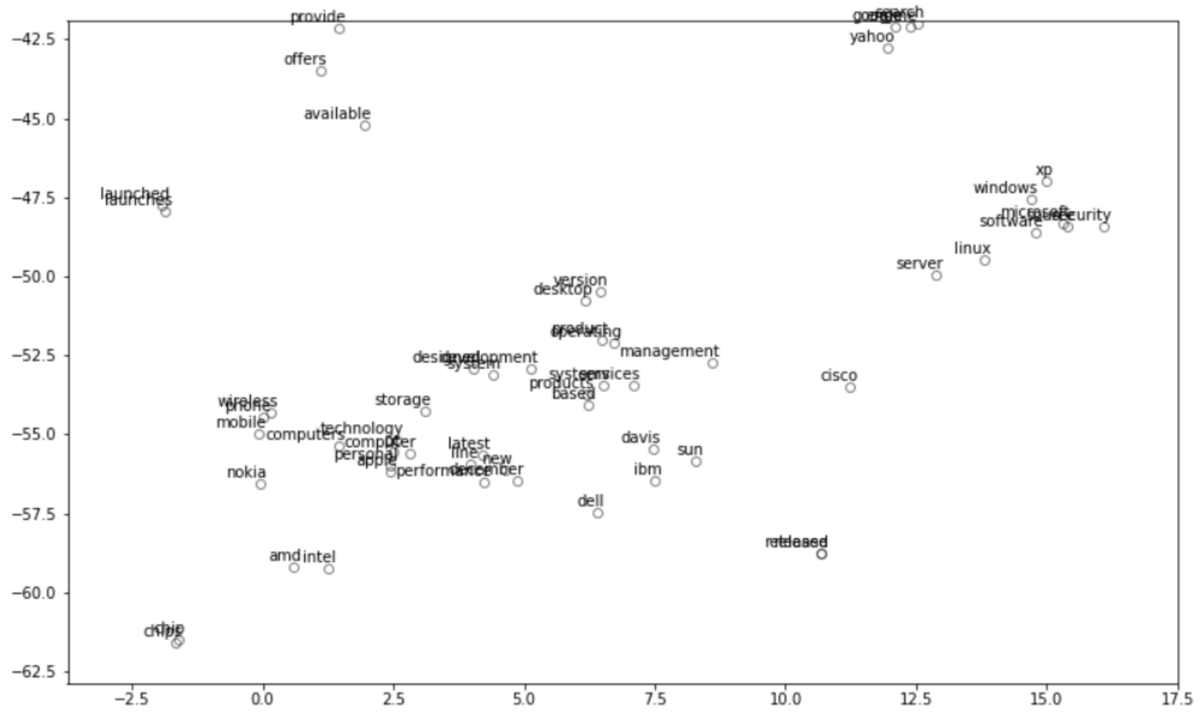


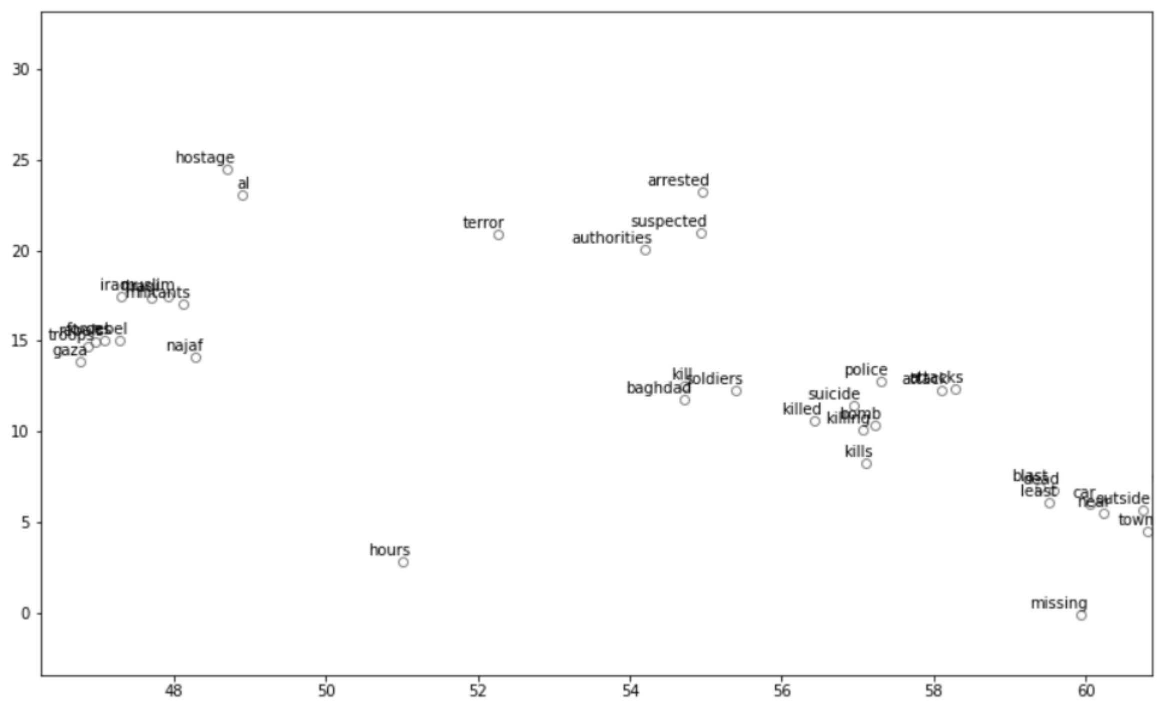
Figure 8. T-SNE result of frequent word vectors.



1. First cluster: German, Germany, France, Paris, India, Indian, British, Britain, Australian
- This cluster represents a country and its capital city or people in that country.



2. Second cluster: Intel, storage, technology, computer, Dell, AMD, desktop, mobile, server, Linux, IBM, wireless
- This cluster represents a category with technology. The elements are technological terminologies and some IT company’s name.



3. Third cluster: terror, arrested, soldiers, Baghdad, kills, killed, suicide, attack, hostage, police, suspected.
- This cluster represents the violent term category. The elements are terror, kill, soldiers, hostage, arrested which has to do with war.

3. Learning-Based Word Vectors - GloVe

Task 3.1 – Derive the gradient of the objective of the loss function and PMI.

$$J_B = \sum_{(i_m, j_m) \in B} f(C_{i_m j_m})(w_{i_m}^T \tilde{w}_{j_m} + b_{i_m} + \tilde{b}_{j_m} - \log C_{i_m j_m})^2$$

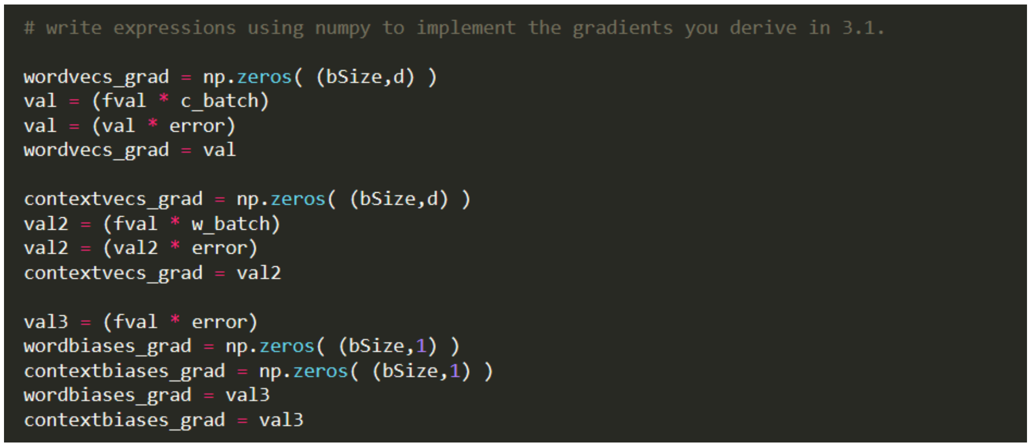
(1) $\frac{\partial J}{\partial w_i} = f(C_{ij})\widetilde{w_j}(w_i^t\widetilde{w_j} + b_i + \widetilde{b_j} - logC_{ij})^2$

(2) $\frac{\partial J}{\partial \widetilde{w_i}} = f(C_{ij})w_i(w_i^t\widetilde{w_j} + b_i + \widetilde{b_j} - logC_{ij})^2$

(3) $\frac{\partial J}{\partial b_i} = f(C_{ij})(w_i^t\widetilde{w_j} + b_i + \widetilde{b_j} - logC_{ij})^2$

(4) $\frac{\partial J}{\partial \widetilde{b_j}} = f(C_{ij})(w_i^t\widetilde{w_j} + b_i + \widetilde{b_j} - logC_{ij})^2$

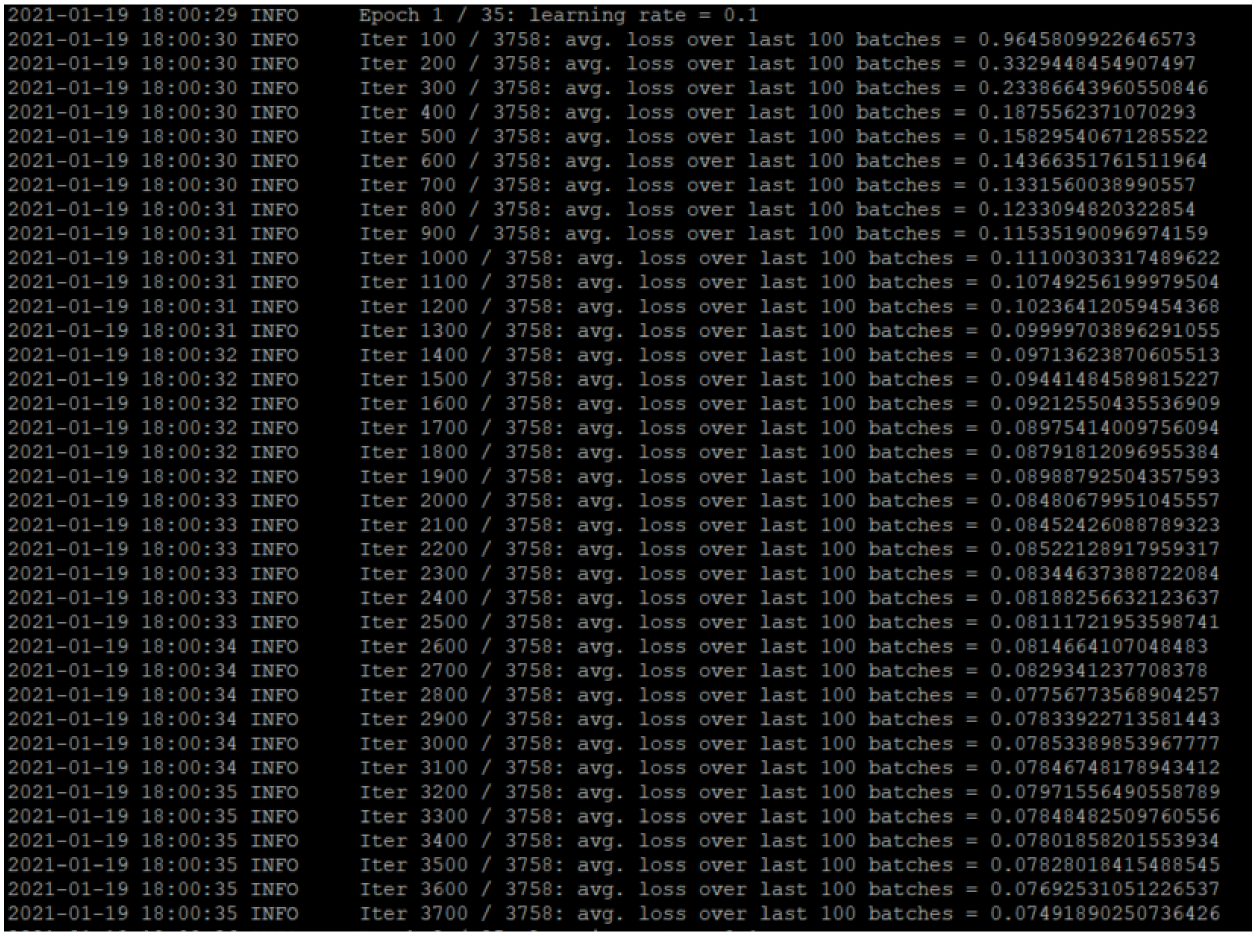
Task 3.2 – Implement the gradient computation for a batch in the build_glove_vectors.py



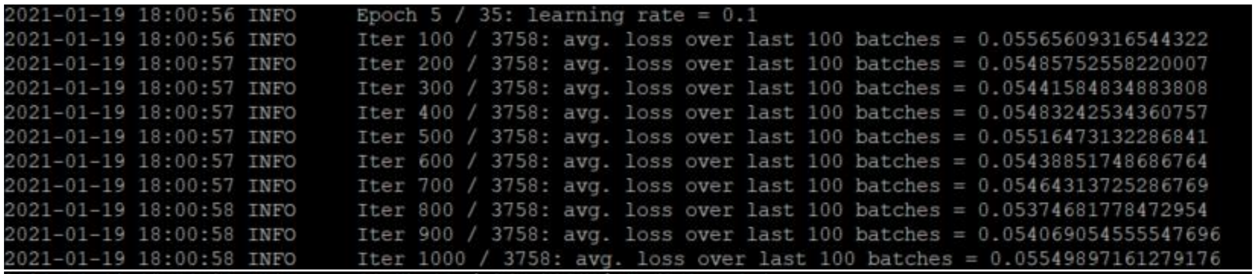
```
# write expressions using numpy to implement the gradients you derive in 3.1.

wordvecs_grad = np.zeros( (bSize,d) )
val = (fval * c_batch)
val = (val * error)
wordvecs_grad = val

contextvecs_grad = np.zeros( (bSize,d) )
val2 = (fval * w_batch)
val2 = (val2 * error)
contextvecs_grad = val2

val3 = (fval * error)
wordbiases_grad = np.zeros( (bSize,1) )
contextbiases_grad = np.zeros( (bSize,1) )
wordbiases_grad = val3
contextbiases_grad = val3
```

Figure 9. gradient computation

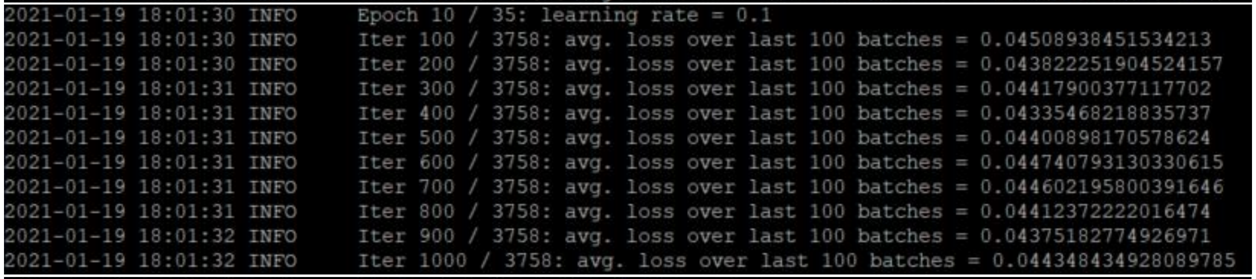
Task 3.3 – Run build_glove_vectors.py to learn GloVe vectors and visualize them with TSNE.



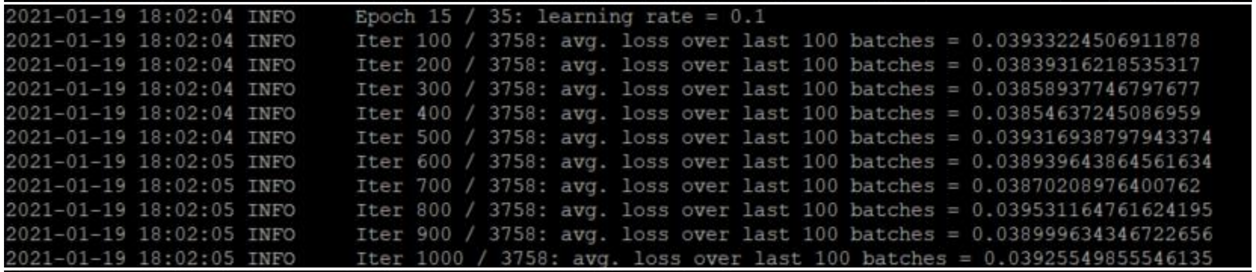
```
2021-01-19 18:00:29 INFO     Epoch 1 / 35: learning rate = 0.1
2021-01-19 18:00:30 INFO     Iter 100 / 3758: avg. loss over last 100 batches = 0.9645809922646573
2021-01-19 18:00:30 INFO     Iter 200 / 3758: avg. loss over last 100 batches = 0.3329448454907497
2021-01-19 18:00:30 INFO     Iter 300 / 3758: avg. loss over last 100 batches = 0.23386643960550846
2021-01-19 18:00:30 INFO     Iter 400 / 3758: avg. loss over last 100 batches = 0.1875562371070293
2021-01-19 18:00:30 INFO     Iter 500 / 3758: avg. loss over last 100 batches = 0.15829540671285522
2021-01-19 18:00:30 INFO     Iter 600 / 3758: avg. loss over last 100 batches = 0.14366351761511964
2021-01-19 18:00:30 INFO     Iter 700 / 3758: avg. loss over last 100 batches = 0.1331560038990557
2021-01-19 18:00:31 INFO     Iter 800 / 3758: avg. loss over last 100 batches = 0.1233094820322854
2021-01-19 18:00:31 INFO     Iter 900 / 3758: avg. loss over last 100 batches = 0.11535190096974159
2021-01-19 18:00:31 INFO     Iter 1000 / 3758: avg. loss over last 100 batches = 0.11100303317489622
2021-01-19 18:00:31 INFO     Iter 1100 / 3758: avg. loss over last 100 batches = 0.10749256199979504
2021-01-19 18:00:31 INFO     Iter 1200 / 3758: avg. loss over last 100 batches = 0.10236412059454368
2021-01-19 18:00:31 INFO     Iter 1300 / 3758: avg. loss over last 100 batches = 0.09999703896291055
2021-01-19 18:00:32 INFO     Iter 1400 / 3758: avg. loss over last 100 batches = 0.09713623870605513
2021-01-19 18:00:32 INFO     Iter 1500 / 3758: avg. loss over last 100 batches = 0.09441484589815227
2021-01-19 18:00:32 INFO     Iter 1600 / 3758: avg. loss over last 100 batches = 0.09212550435536909
2021-01-19 18:00:32 INFO     Iter 1700 / 3758: avg. loss over last 100 batches = 0.08975414009756094
2021-01-19 18:00:32 INFO     Iter 1800 / 3758: avg. loss over last 100 batches = 0.08791812096955384
2021-01-19 18:00:32 INFO     Iter 1900 / 3758: avg. loss over last 100 batches = 0.08988792504357593
2021-01-19 18:00:33 INFO     Iter 2000 / 3758: avg. loss over last 100 batches = 0.08480679951045557
2021-01-19 18:00:33 INFO     Iter 2100 / 3758: avg. loss over last 100 batches = 0.08452426088789323
2021-01-19 18:00:33 INFO     Iter 2200 / 3758: avg. loss over last 100 batches = 0.08522128917959317
2021-01-19 18:00:33 INFO     Iter 2300 / 3758: avg. loss over last 100 batches = 0.08344637388722084
2021-01-19 18:00:33 INFO     Iter 2400 / 3758: avg. loss over last 100 batches = 0.08188256632123637
2021-01-19 18:00:33 INFO     Iter 2500 / 3758: avg. loss over last 100 batches = 0.08111721953598741
2021-01-19 18:00:34 INFO     Iter 2600 / 3758: avg. loss over last 100 batches = 0.0814664107048483
2021-01-19 18:00:34 INFO     Iter 2700 / 3758: avg. loss over last 100 batches = 0.0829341237708378
2021-01-19 18:00:34 INFO     Iter 2800 / 3758: avg. loss over last 100 batches = 0.07756773568904257
2021-01-19 18:00:34 INFO     Iter 2900 / 3758: avg. loss over last 100 batches = 0.07833922713581443
2021-01-19 18:00:34 INFO     Iter 3000 / 3758: avg. loss over last 100 batches = 0.07853389853967777
2021-01-19 18:00:34 INFO     Iter 3100 / 3758: avg. loss over last 100 batches = 0.07846748178943412
2021-01-19 18:00:35 INFO     Iter 3200 / 3758: avg. loss over last 100 batches = 0.07971556490558789
2021-01-19 18:00:35 INFO     Iter 3300 / 3758: avg. loss over last 100 batches = 0.07848482509760556
2021-01-19 18:00:35 INFO     Iter 3400 / 3758: avg. loss over last 100 batches = 0.07801858201553934
2021-01-19 18:00:35 INFO     Iter 3500 / 3758: avg. loss over last 100 batches = 0.07828018415488545
2021-01-19 18:00:35 INFO     Iter 3600 / 3758: avg. loss over last 100 batches = 0.07692531051226537
2021-01-19 18:00:35 INFO     Iter 3700 / 3758: avg. loss over last 100 batches = 0.07491890250736426
```



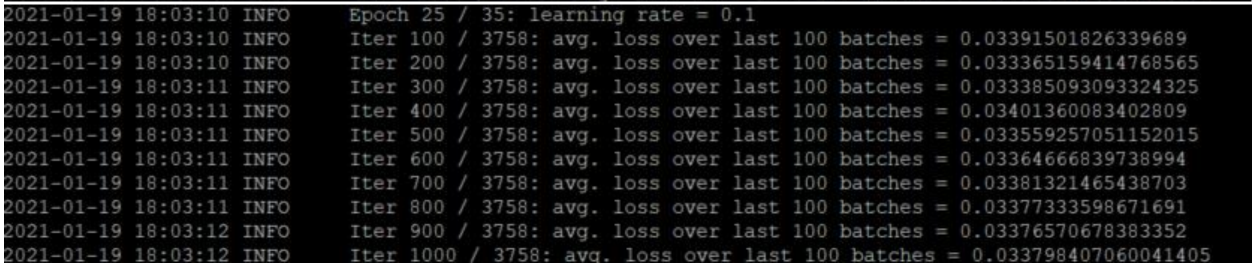
```
2021-01-19 18:00:56 INFO     Epoch 5 / 35: learning rate = 0.1
2021-01-19 18:00:56 INFO     Iter 100 / 3758: avg. loss over last 100 batches = 0.05565609316544322
2021-01-19 18:00:57 INFO     Iter 200 / 3758: avg. loss over last 100 batches = 0.05485752558220007
2021-01-19 18:00:57 INFO     Iter 300 / 3758: avg. loss over last 100 batches = 0.05441584834883808
2021-01-19 18:00:57 INFO     Iter 400 / 3758: avg. loss over last 100 batches = 0.05483242534360757
2021-01-19 18:00:57 INFO     Iter 500 / 3758: avg. loss over last 100 batches = 0.05516473132286841
2021-01-19 18:00:57 INFO     Iter 600 / 3758: avg. loss over last 100 batches = 0.05438851748686764
2021-01-19 18:00:57 INFO     Iter 700 / 3758: avg. loss over last 100 batches = 0.05464313725286769
2021-01-19 18:00:58 INFO     Iter 800 / 3758: avg. loss over last 100 batches = 0.05374681778472954
2021-01-19 18:00:58 INFO     Iter 900 / 3758: avg. loss over last 100 batches = 0.054069054555547696
2021-01-19 18:00:58 INFO     Iter 1000 / 3758: avg. loss over last 100 batches = 0.05549897161279176
```



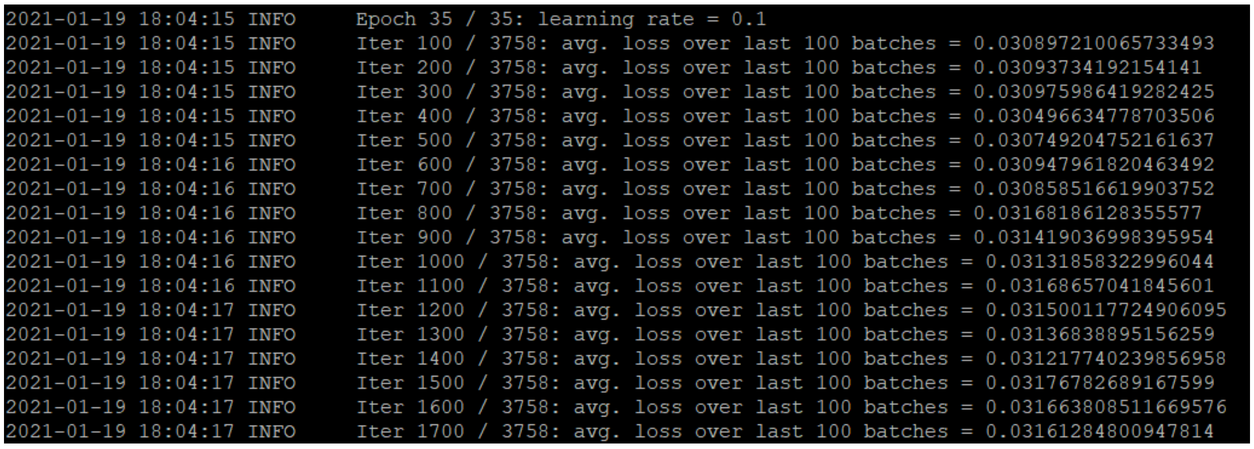
```
2021-01-19 18:01:30 INFO     Epoch 10 / 35: learning rate = 0.1
2021-01-19 18:01:30 INFO     Iter 100 / 3758: avg. loss over last 100 batches = 0.04508938451534213
2021-01-19 18:01:30 INFO     Iter 200 / 3758: avg. loss over last 100 batches = 0.043822251904524157
2021-01-19 18:01:31 INFO     Iter 300 / 3758: avg. loss over last 100 batches = 0.04417900377117702
2021-01-19 18:01:31 INFO     Iter 400 / 3758: avg. loss over last 100 batches = 0.04335468218835737
2021-01-19 18:01:31 INFO     Iter 500 / 3758: avg. loss over last 100 batches = 0.04400898170578624
2021-01-19 18:01:31 INFO     Iter 600 / 3758: avg. loss over last 100 batches = 0.044740793130330615
2021-01-19 18:01:31 INFO     Iter 700 / 3758: avg. loss over last 100 batches = 0.044602195800391646
2021-01-19 18:01:31 INFO     Iter 800 / 3758: avg. loss over last 100 batches = 0.04412372222016474
2021-01-19 18:01:32 INFO     Iter 900 / 3758: avg. loss over last 100 batches = 0.04375182774926971
2021-01-19 18:01:32 INFO     Iter 1000 / 3758: avg. loss over last 100 batches = 0.044348434928089785
```



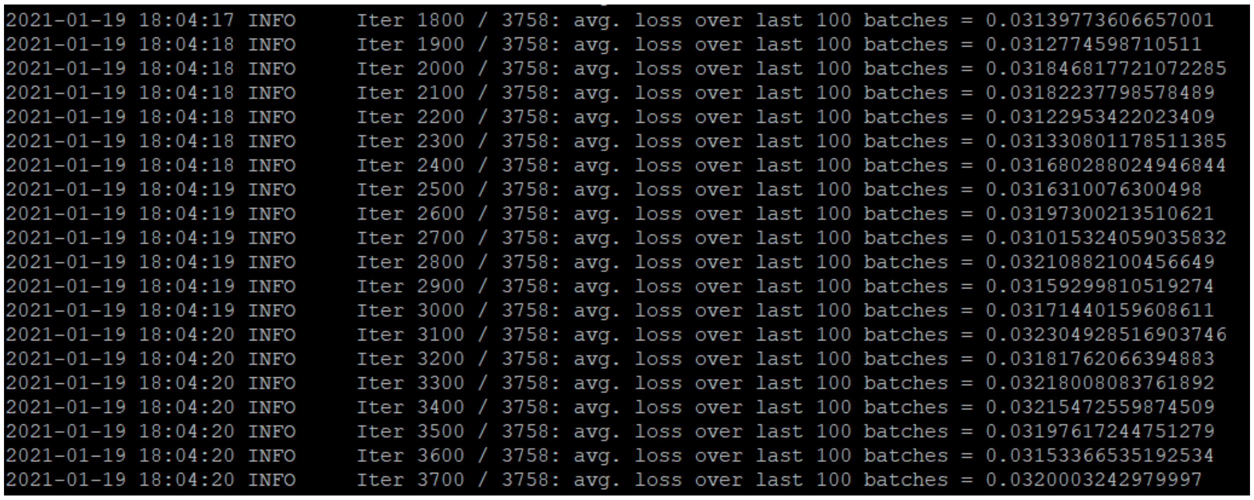
```
2021-01-19 18:02:04 INFO     Epoch 15 / 35: learning rate = 0.1
2021-01-19 18:02:04 INFO     Iter 100 / 3758: avg. loss over last 100 batches = 0.03933224506911878
2021-01-19 18:02:04 INFO     Iter 200 / 3758: avg. loss over last 100 batches = 0.03839316218535317
2021-01-19 18:02:04 INFO     Iter 300 / 3758: avg. loss over last 100 batches = 0.03858937746797677
2021-01-19 18:02:04 INFO     Iter 400 / 3758: avg. loss over last 100 batches = 0.03854637245086959
2021-01-19 18:02:04 INFO     Iter 500 / 3758: avg. loss over last 100 batches = 0.039316938797943374
2021-01-19 18:02:05 INFO     Iter 600 / 3758: avg. loss over last 100 batches = 0.038939643864561634
2021-01-19 18:02:05 INFO     Iter 700 / 3758: avg. loss over last 100 batches = 0.03870208976400762
2021-01-19 18:02:05 INFO     Iter 800 / 3758: avg. loss over last 100 batches = 0.039531164761624195
2021-01-19 18:02:05 INFO     Iter 900 / 3758: avg. loss over last 100 batches = 0.038999634346722656
2021-01-19 18:02:05 INFO     Iter 1000 / 3758: avg. loss over last 100 batches = 0.03925549855546135
```



```
2021-01-19 18:03:10 INFO     Epoch 25 / 35: learning rate = 0.1
2021-01-19 18:03:10 INFO     Iter 100 / 3758: avg. loss over last 100 batches = 0.03391501826339689
2021-01-19 18:03:10 INFO     Iter 200 / 3758: avg. loss over last 100 batches = 0.033365159414768565
2021-01-19 18:03:11 INFO     Iter 300 / 3758: avg. loss over last 100 batches = 0.033385093093324325
2021-01-19 18:03:11 INFO     Iter 400 / 3758: avg. loss over last 100 batches = 0.03401360083402809
2021-01-19 18:03:11 INFO     Iter 500 / 3758: avg. loss over last 100 batches = 0.033559257051152015
2021-01-19 18:03:11 INFO     Iter 600 / 3758: avg. loss over last 100 batches = 0.03364666839738994
2021-01-19 18:03:11 INFO     Iter 700 / 3758: avg. loss over last 100 batches = 0.03381321465438703
2021-01-19 18:03:11 INFO     Iter 800 / 3758: avg. loss over last 100 batches = 0.03377333598671691
2021-01-19 18:03:12 INFO     Iter 900 / 3758: avg. loss over last 100 batches = 0.03376570678383352
2021-01-19 18:03:12 INFO     Iter 1000 / 3758: avg. loss over last 100 batches = 0.033798407060041405
```



```
2021-01-19 18:04:15 INFO     Epoch 35 / 35: learning rate = 0.1
2021-01-19 18:04:15 INFO     Iter 100 / 3758: avg. loss over last 100 batches = 0.030897210065733493
2021-01-19 18:04:15 INFO     Iter 200 / 3758: avg. loss over last 100 batches = 0.03093734192154141
2021-01-19 18:04:15 INFO     Iter 300 / 3758: avg. loss over last 100 batches = 0.030975986419282425
2021-01-19 18:04:15 INFO     Iter 400 / 3758: avg. loss over last 100 batches = 0.030496634778703506
2021-01-19 18:04:15 INFO     Iter 500 / 3758: avg. loss over last 100 batches = 0.030749204752161637
2021-01-19 18:04:16 INFO     Iter 600 / 3758: avg. loss over last 100 batches = 0.030947961820463492
2021-01-19 18:04:16 INFO     Iter 700 / 3758: avg. loss over last 100 batches = 0.030858516619903752
2021-01-19 18:04:16 INFO     Iter 800 / 3758: avg. loss over last 100 batches = 0.03168186128355577
2021-01-19 18:04:16 INFO     Iter 900 / 3758: avg. loss over last 100 batches = 0.031419036998395954
2021-01-19 18:04:16 INFO     Iter 1000 / 3758: avg. loss over last 100 batches = 0.03131858322996044
2021-01-19 18:04:16 INFO     Iter 1100 / 3758: avg. loss over last 100 batches = 0.03168657041845601
2021-01-19 18:04:17 INFO     Iter 1200 / 3758: avg. loss over last 100 batches = 0.031500117724906095
2021-01-19 18:04:17 INFO     Iter 1300 / 3758: avg. loss over last 100 batches = 0.03136838895156259
2021-01-19 18:04:17 INFO     Iter 1400 / 3758: avg. loss over last 100 batches = 0.031217740239856958
2021-01-19 18:04:17 INFO     Iter 1500 / 3758: avg. loss over last 100 batches = 0.03176782689167599
2021-01-19 18:04:17 INFO     Iter 1600 / 3758: avg. loss over last 100 batches = 0.031663808511669576
2021-01-19 18:04:17 INFO     Iter 1700 / 3758: avg. loss over last 100 batches = 0.03161284800947814
```



```
2021-01-19 18:04:17 INFO     Iter 1800 / 3758: avg. loss over last 100 batches = 0.03139773606657001
2021-01-19 18:04:18 INFO     Iter 1900 / 3758: avg. loss over last 100 batches = 0.0312774598710511
2021-01-19 18:04:18 INFO     Iter 2000 / 3758: avg. loss over last 100 batches = 0.031846817721072285
2021-01-19 18:04:18 INFO     Iter 2100 / 3758: avg. loss over last 100 batches = 0.03182237798578489
2021-01-19 18:04:18 INFO     Iter 2200 / 3758: avg. loss over last 100 batches = 0.03122953422023409
2021-01-19 18:04:18 INFO     Iter 2300 / 3758: avg. loss over last 100 batches = 0.031330801178511385
2021-01-19 18:04:18 INFO     Iter 2400 / 3758: avg. loss over last 100 batches = 0.03168028802494684
2021-01-19 18:04:19 INFO     Iter 2500 / 3758: avg. loss over last 100 batches = 0.0316310076300498
2021-01-19 18:04:19 INFO     Iter 2600 / 3758: avg. loss over last 100 batches = 0.03197300213510621
2021-01-19 18:04:19 INFO     Iter 2700 / 3758: avg. loss over last 100 batches = 0.031015324059035832
2021-01-19 18:04:19 INFO     Iter 2800 / 3758: avg. loss over last 100 batches = 0.0321088210045664
2021-01-19 18:04:19 INFO     Iter 2900 / 3758: avg. loss over last 100 batches = 0.0315929981051927
2021-01-19 18:04:19 INFO     Iter 3000 / 3758: avg. loss over last 100 batches = 0.0317144015960861
2021-01-19 18:04:20 INFO     Iter 3100 / 3758: avg. loss over last 100 batches = 0.032304928516903746
2021-01-19 18:04:20 INFO     Iter 3200 / 3758: avg. loss over last 100 batches = 0.03181762066394883
2021-01-19 18:04:20 INFO     Iter 3300 / 3758: avg. loss over last 100 batches = 0.03218008083761892
2021-01-19 18:04:20 INFO     Iter 3400 / 3758: avg. loss over last 100 batches = 0.03215472559874509
2021-01-19 18:04:20 INFO     Iter 3500 / 3758: avg. loss over last 100 batches = 0.03197617244751279
2021-01-19 18:04:20 INFO     Iter 3600 / 3758: avg. loss over last 100 batches = 0.03153366535192534
2021-01-19 18:04:20 INFO     Iter 3700 / 3758: avg. loss over last 100 batches = 0.0320003242979997
```

These are a few loss values according to several epochs. In the first epoch, loss starts with 0.96 and gradually decreases as the epochs and iterations get higher. In each epoch, the loss value oscillates slightly, but if we see the loss values in a whole, we can see it is decreasing and converges to a value near 0.31.

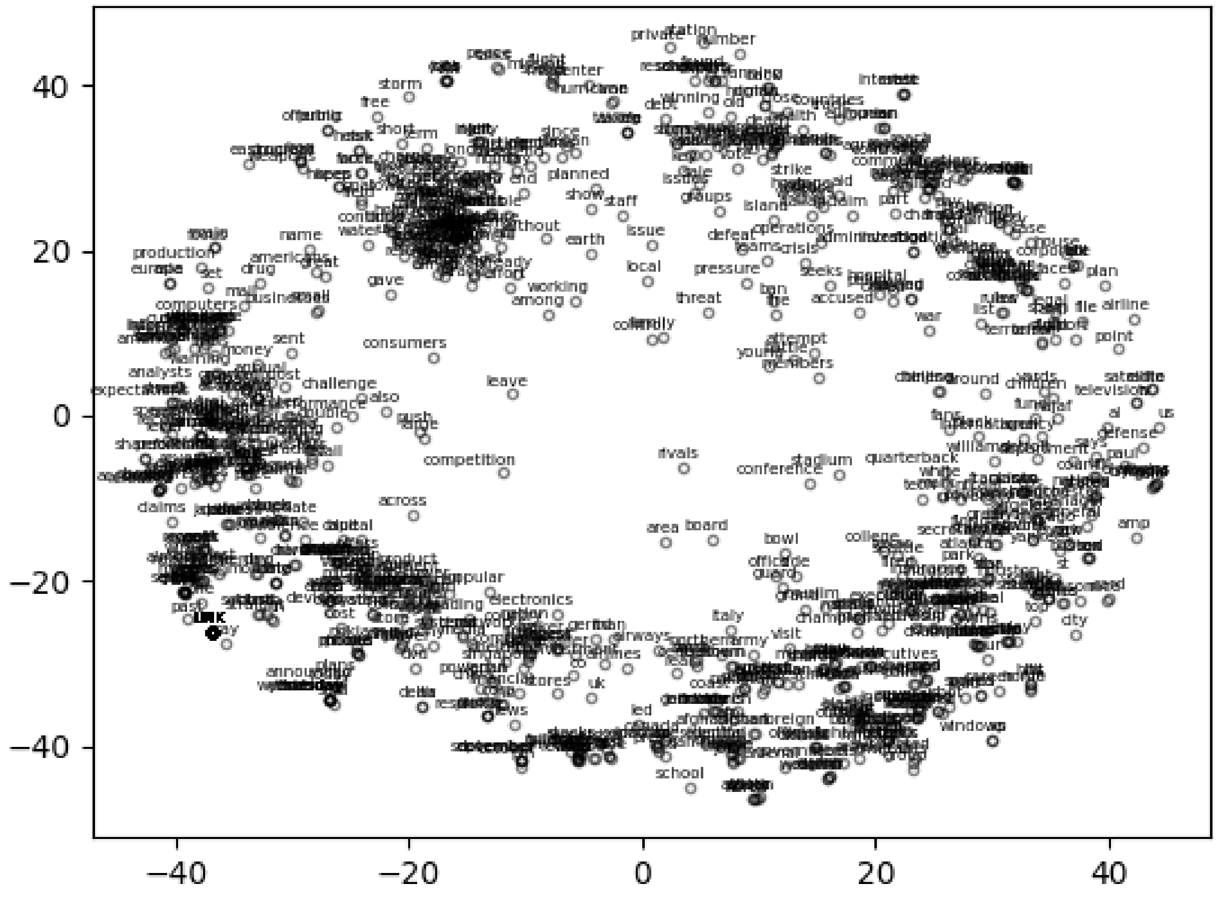
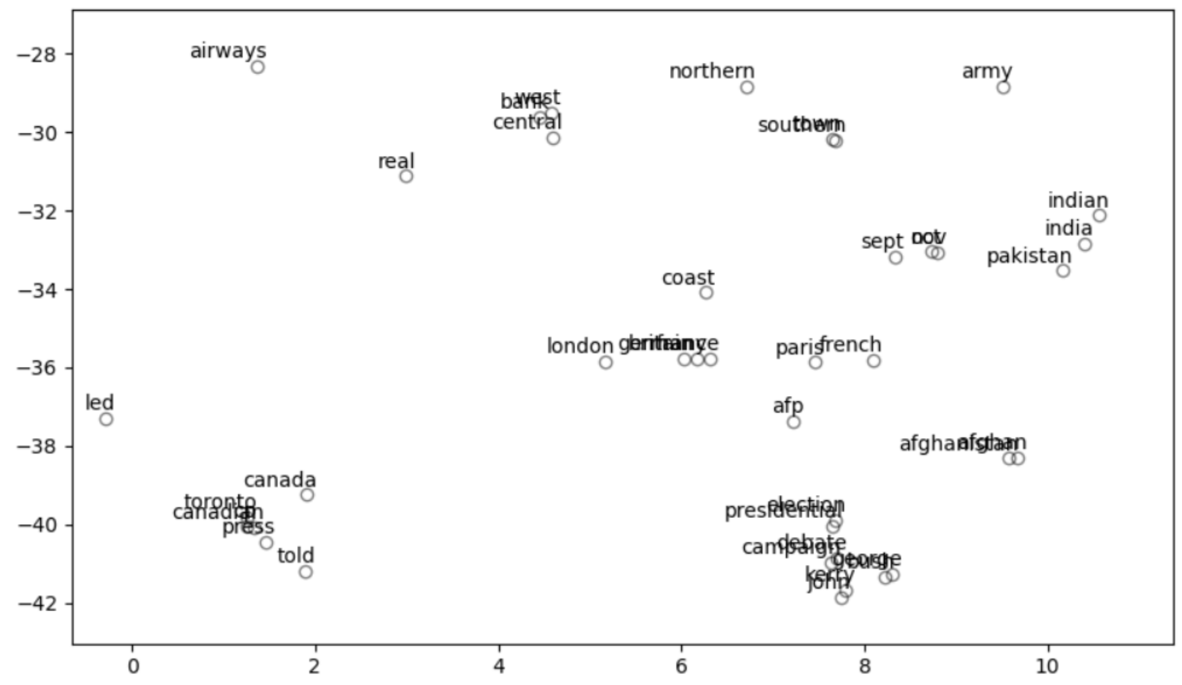
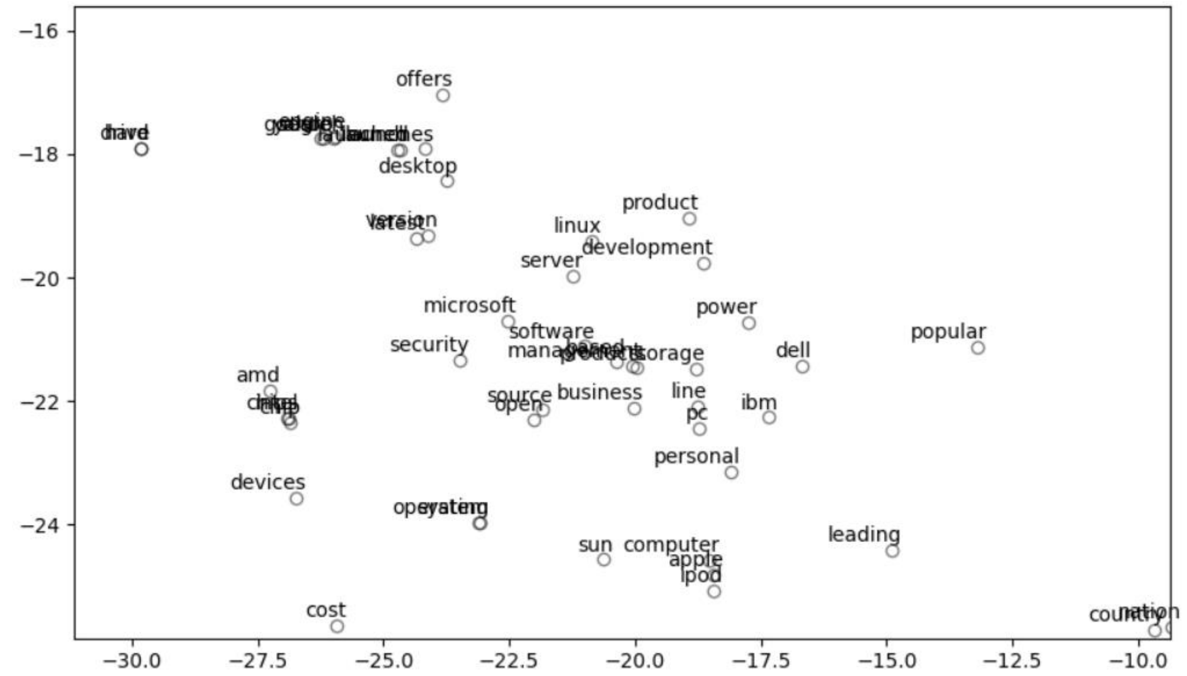


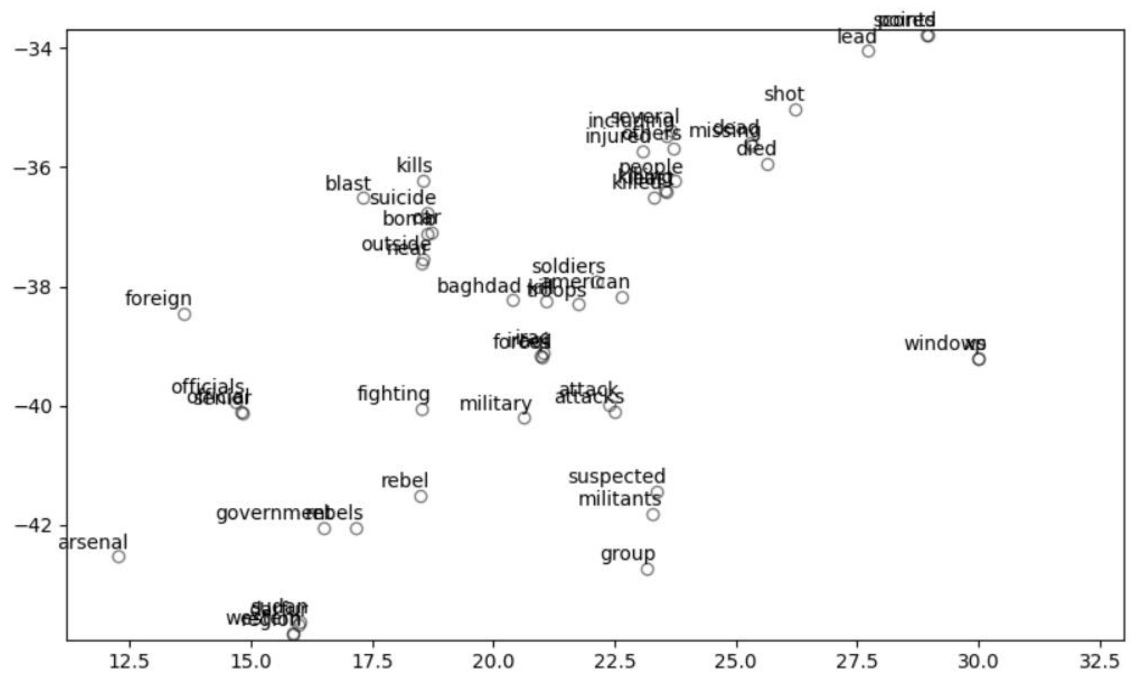
Figure 10. TSNE result of GloVe vector.



1. First cluster: Canada, Toronto, Paris French, India, Indian, Afghanistan, London, Germany.
This cluster represents the country category. The elements are the country name and its capital city.



2. Second cluster: Dell, IBM, pc, desktop, Linux, Server, Microsoft, AMD, computer, apple, iPod, software.
- This cluster represents the technology category. The elements of this category are IT company's name or technological terminologies.



3. Third cluster: kills, suicide, border, military, soldiers, Baghdad, injured, shot, died, suspected.
- This cluster represents the violent term category. The elements in the category are related to war and some violent terminologies.

If we compare the three clusters from the build_freq_vectors.py and build_glove_vectors.py, the whole cluster looks different, but sentiments clusters seem pretty similar.

4. Exploring learned biases in word2vec vectors

Task 4.1 Use the most_similar function to find three additional analogies that work.

Like, man : king :: woman : ?, this analogy function, a : b :: c : ?, return a result 'd' that proportional to 'c' like it does in 'a' and 'b'. These are the results that worked well.

```
analogy('usa', 'dollar', 'EU')
```

```
usa : dollar :: EU : ?
[('euro', 0.625), ('European_Union', 0.54), ('greenback', 0.54), ('currency', 0.532), ('Swiss_franc', 0.489), ('currencies',
0.476), ('safe_haven_Swiss_franc', 0.458), ('złoty', 0.457), ('renmimbi', 0.457), ('Swiss_Franc', 0.455)]
```

```
analogy('Tokyo', 'Japan', 'Oslo')
```

```
Tokyo : Japan :: Oslo : ?
[('Norway', 0.756), ('Norwegian', 0.605), ('Denmark', 0.587), ('coach_Age_Hareide', 0.587), ('Oslo_Norway', 0.584), ('Swede
n', 0.583), ('Norwegians', 0.567), ('Trond_Nymark', 0.545), ('Coffele_travels', 0.536), ('Jaysuma_Ndure', 0.534)]
```

```
analogy('brother', 'boy', 'sister')
```

```
brother : boy :: sister : ?
[('girl', 0.85), ('teenage_girl', 0.666), ('toddler', 0.666), ('mother', 0.633), ('woman', 0.632), ('teenager', 0.62), ('sch
oolgirl', 0.613), ('daughter', 0.608), ('grandmother', 0.584), ('teen', 0.579)]
```

```
analogy('Seoul', 'Korea', 'Pyeongyang')
```

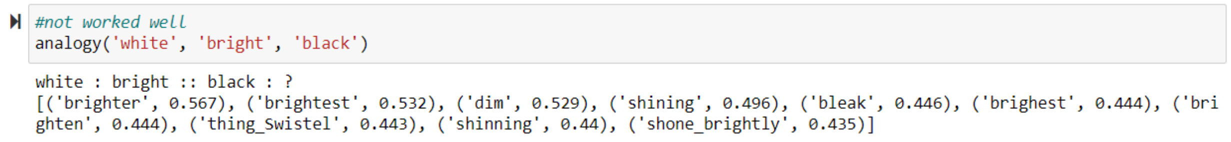
```
Seoul : Korea :: Pyeongyang : ?
[('DPRK', 0.653), ('North_Korea', 0.637), ('Korean', 0.593), ('South_Korea', 0.549), ('North_Koreans', 0.546), ('Korea_DPR
K', 0.541), ('◆_N.', 0.539), ('PDRK', 0.522), ('Pyonyang', 0.521), ('Korean_Peninsula', 0.517)]
```

All those examples worked well because their results satisfy the proportion. If we consider the original example, the male ruler is called king and female ruler is called queen, and those are the term that does not change by subjectivity. So, I selected analogy examples with objective terms.

The examples above have similar proportions.

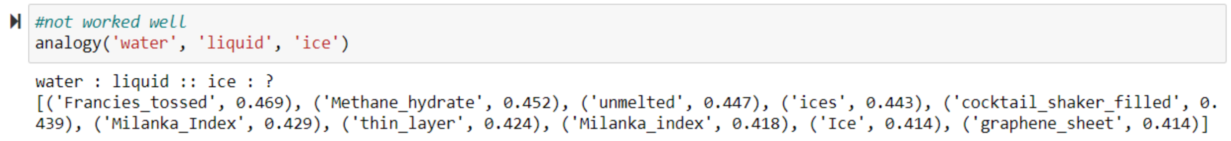
(1) the United States uses the dollar and the European Union use euro, and the results show the euro with the highest confidence score.
(2) Tokyo is the capital city of Japan and Oslo is the capital city of Norway and results show Norway, Norwegian which is the correct prediction.
(3) A brother is a male term and a sister is a female term so boy and girl can be the correct answer.
(4) Seoul is the capital city of Korea and Pyeongyang is the capital city of North Korea and the results show DPRK, which is the full name of North Korea with the highest confidence score.

Task 4.2 – Use the most_similar function to find three analogies that did not work.



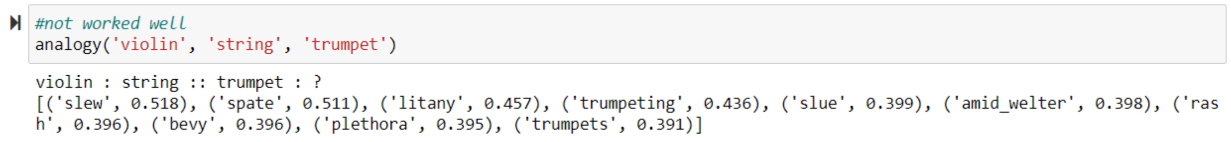
```
#not worked well
analogy('white', 'bright', 'black')
```

```
white : bright :: black : ?
[('brighter', 0.567), ('brightest', 0.532), ('dim', 0.529), ('shining', 0.496), ('bleak', 0.446), ('brighest', 0.444), ('bri
ghten', 0.444), ('thing_Swistel', 0.443), ('shinning', 0.44), ('shone_brightly', 0.435)]
```



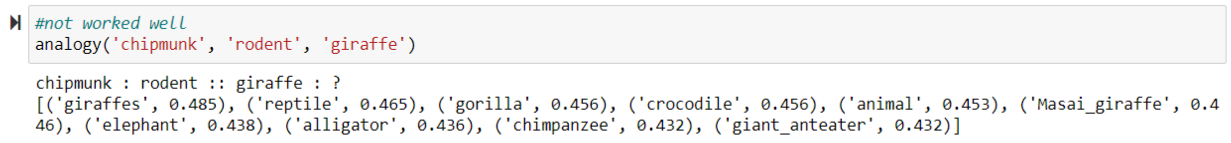
```
#not worked well
analogy('water', 'liquid', 'ice')
```

```
water : liquid :: ice : ?
[('Francies_tossed', 0.469), ('Methane_hydrate', 0.452), ('unmelted', 0.447), ('ices', 0.443), ('cocktail_shaker_filled', 0.
439), ('Milanka_Index', 0.429), ('thin_layer', 0.424), ('Milanka_index', 0.418), ('Ice', 0.414), ('graphene_sheet', 0.414)]
```



```
#not worked well
analogy('violin', 'string', 'trumpet')
```

```
violin : string :: trumpet : ?
[('slew', 0.518), ('spate', 0.511), ('litany', 0.457), ('trumpeting', 0.436), ('slue', 0.399), ('amid_welter', 0.398), ('ras
h', 0.396), ('bevy', 0.396), ('plethora', 0.395), ('trumpets', 0.391)]
```

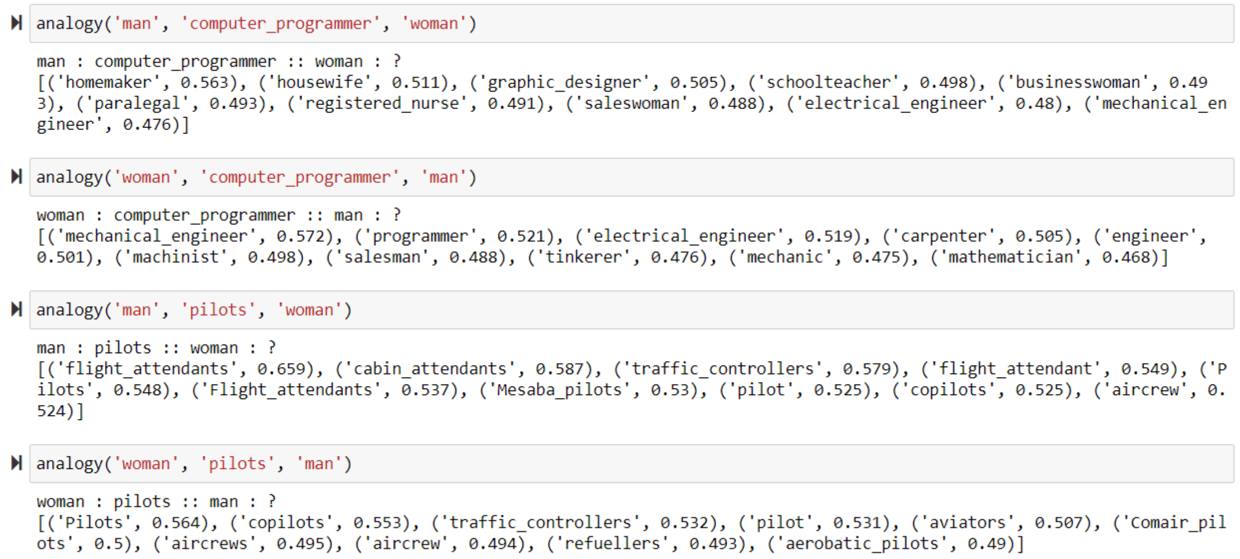


```
#not worked well
analogy('chipmunk', 'rodent', 'giraffe')
```

```
chipmunk : rodent :: giraffe : ?
[('giraffes', 0.485), ('reptile', 0.465), ('gorilla', 0.456), ('crocodile', 0.456), ('animal', 0.453), ('Masai_giraffe', 0.4
46), ('elephant', 0.438), ('alligator', 0.436), ('chimpanzee', 0.432), ('giant_anteater', 0.432)]
```

The above examples are analogies that did not work.
(1) White and black have opposite brightness and I thought it is obvious that white is bright and black is dark, the highest confidence results are brighter and brightest which are not related to dark. Even though 'dim' was the third result, other results were related to brightest, shinning so I chose this example as 'not work'.
(2) Water is a liquid object and ice is a solid object and they are both not objective terminologies but the results seemed to be far apart from solid, so I chose this example as 'not work'.
(3) A violin in a string instrument and a trumpet is a brass instrument, however, the result was slew, spate which was not related to trumpet, so I chose this example as 'not work'.
(4) A chipmunk's order is rodent and the giraffe's order is therapsid, but results such as reptile, a gorilla is not related to the giraffe, so I chose this example as 'not work'.

Task 4.3 – Use the most_similar function to find two additional cases of bias.



```
analogy('man', 'computer_programmer', 'woman')
```

```
man : computer_programmer :: woman : ?
[('homemaker', 0.563), ('housewife', 0.511), ('graphic_designer', 0.505), ('schoolteacher', 0.498), ('businesswoman', 0.49
3), ('paralegal', 0.493), ('registered_nurse', 0.491), ('saleswoman', 0.488), ('electrical_engineer', 0.48), ('mechanical_en
gineer', 0.476)]
```

```
analogy('woman', 'computer_programmer', 'man')
```

```
woman : computer_programmer :: man : ?
[('mechanical_engineer', 0.572), ('programmer', 0.521), ('electrical_engineer', 0.519), ('carpenter', 0.505), ('engineer',
0.501), ('machinist', 0.498), ('salesman', 0.488), ('tinkerer', 0.476), ('mechanic', 0.475), ('mathematician', 0.468)]
```

```
analogy('man', 'pilots', 'woman')
```

```
man : pilots :: woman : ?
[('flight_attendants', 0.659), ('cabin_attendants', 0.587), ('traffic_controllers', 0.579), ('flight_attendant', 0.549), ('P
ilots', 0.548), ('Flight_attendants', 0.537), ('Mesaba_pilots', 0.53), ('pilot', 0.525), ('copilots', 0.525), ('aircrew', 0.
524)]
```

```
analogy('woman', 'pilots', 'man')
```

```
woman : pilots :: man : ?
[('Pilots', 0.564), ('copilots', 0.553), ('traffic_controllers', 0.532), ('pilot', 0.531), ('aviators', 0.507), ('Comair_pil
ots', 0.5), ('aircrews', 0.495), ('aircrew', 0.494), ('refuellers', 0.493), ('aerobatic_pilots', 0.49)]
```

Above examples are what I consider biased analogies.

(1) If a man is a computer programmer then a woman's result should be anything corresponding to a computer programmer but the result shows homemaker, housewife which considers man as a worker and woman as a people who just stay at home and supporting man. However, if we put a woman as a computer programmer, man's results are mechanical engineer, programmer, electrical engineer which is equivalent to a computer programmer.
(2) If a man is a pilot then a woman is flight attendants, but if a woman is a pilot, then a man is also a pilot or copilot. This result is gender-biased since it considers woman as flight attendants that supports pilot and it is not an equivalent term.

I consider those two examples as biased cases since they are showing the result a woman does not have an equivalent job as a man and showing the roles that just support man.

Task 4.4 – Why might these biases exist in word2vec and what are some potential consequences that might result if word2vec were used in a live system?

It is because our society and our communication still have gender bias and it reflects into word embeddings. Even though the world and people's think is changing, there are still many biases toward not only gender but also to religion or ethnicity.
If word2vec were used in a live system, these biased data will result in biased outcomes. A machine learning program using this biased system will cause a biased result and people using that system could obtain biased understanding.