NLP Homework 3 Programming Assignment

Word Embeddings

Word embeddings or word vectors give us a way to use an efficient, dense representation in which similar words have a similar encoding. We have previously seen one-hot vectors used for representing words in a vocabulary. But, unlike these, word embeddings are capable of capturing the context of a word in a document, semantic and syntactic similarity and relation with other words.

There are several popular word embeddings that are used, some of them are-

- Word2Vec (by Google) (https://code.google.com/archive/p/word2vec/)
- GloVe (by Stanford) (https://nlp.stanford.edu/projects/glove/)
- fastText (by Facebook) (https://fasttext.cc/)

In this assignment, we will be exploring the **word2vec embeddings**, the embedding technique that was popularized by Mikolov et al. in 2013 (refer to the original paper here (https://papers.nips.cc/paper/5021-distributed-representations-of-words-and-phrases-and-their-compositionality.pdf)). For this, we will be using the GenSim package, find documentation here (https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html#sphx-glr-auto-examples-tutorials-run-word2vec-py). This model is provided by Google and is trained on Google News dataset. Word embeddings from this model have 300 dimensions and are trained on 3 million words and phrases.

Loading word vectors from GenSim

Fetch and load the `word2vec-google-news-300` pre-trained embeddings. Note that this may take a few minutes.

In [ ]:
```
import numpy as np
import gensim.downloader as api

def download_word2vec_embeddings():
    print("Downloading pre-trained word embeddings from: word2vec
-google-news-300.\n"
          + "Note: This can take a few minutes.\n")
    wv = api.load("word2vec-google-news-300")
    print("\nLoading complete!\n" +
          "Vocabulary size: {}".format(len(wv.vocab)))
    return wv

word_vectors = download_word2vec_embeddings()
```

```
Downloading pre-trained word embeddings from: word2vec-google-new
s-300.
Note: This can take a few minutes.


Loading complete!
Vocabulary size: 3000000
```

The loaded `word_vectors` in memory can be accessed like a dictionary to obtain the embedding of any word, like so-

In [ ]:
```
print(word_vectors['hello'])
print("\nThe embedding has a shape of: {}".format(word_vectors['h
ello'].shape))
```

```
[-0.05419922  0.01708984 -0.00527954  0.33203125 -0.25       -0.0
1397705
 -0.15039062 -0.265625    0.01647949  0.3828125  -0.03295898 -0.0
9716797
 -0.16308594 -0.04443359  0.00946045  0.18457031  0.03637695  0.1
6601562
  0.36328125 -0.25585938  0.375       0.171875    0.21386719 -0.1
9921875
  0.13085938 -0.07275391 -0.02819824  0.11621094  0.15332031  0.0
9082031
  0.06787109 -0.0300293  -0.16894531 -0.20800781 -0.03710938 -0.2
2753906
  0.26367188  0.012146    0.18359375  0.31054688 -0.10791016 -0.1
9140625
  0.21582031  0.13183594 -0.03515625  0.18554688 -0.30859375  0.0
4785156
 -0.10986328  0.14355469 -0.43554688 -0.0378418   0.10839844  0.1
40625
 -0.10595703  0.26171875 -0.17089844  0.39453125  0.12597656 -0.2
7734375
 -0.28125     0.14746094 -0.20996094  0.02355957  0.18457031  0.0
0445557
 -0.27929688 -0.03637695 -0.29296875  0.19628906  0.20703125  0.2
890625
 -0.20507812  0.06787109 -0.43164062 -0.10986328 -0.2578125  -0.0
2331543
  0.11328125  0.23144531 -0.04418945  0.10839844 -0.2890625  -0.0
9521484
 -0.10351562 -0.0324707   0.07763672 -0.13378906  0.22949219  0.0
6298828
  0.08349609  0.02929688 -0.11474609  0.00534058 -0.12988281  0.0
2514648
  0.08789062  0.24511719 -0.11474609 -0.296875   -0.59375    -0.2
9492188
 -0.13378906  0.27734375 -0.04174805  0.11621094  0.28320312  0.0
0241089
  0.13867188 -0.00683594 -0.30078125  0.16210938  0.01171875 -0.1
3867188
  0.48828125  0.02880859  0.02416992  0.04736328  0.05859375 -0.2
3828125
  0.02758789  0.05981445 -0.03857422  0.06933594  0.14941406 -0.1
0888672
 -0.07324219  0.08789062  0.27148438  0.06591797 -0.37890625 -0.2
6171875
 -0.13183594  0.09570312 -0.3125      0.10205078  0.03063965  0.2
3632812
  0.00582886  0.27734375  0.20507812 -0.17871094 -0.31445312 -0.0
1586914
  0.13964844  0.13574219  0.0390625  -0.29296875  0.234375   -0.3
3984375
 -0.11816406  0.10644531 -0.18457031 -0.02099609  0.02563477  0.2
5390625
  0.07275391  0.13574219 -0.00138092 -0.2578125  -0.2890625   0.1
0107422
  0.19238281 -0.04882812  0.27929688 -0.3359375  -0.07373047  0.0
1879883
 -0.10986328 -0.04614258  0.15722656  0.06689453 -0.03417969  0.1
6308594
  0.08642578  0.44726562  0.02026367 -0.01977539  0.07958984  0.1
7773438
```

```
 -0.04370117 -0.00952148  0.16503906  0.17285156  0.23144531 -0.0
4272461
  0.02355957  0.18359375 -0.41601562 -0.01745605  0.16796875  0.0
4736328
  0.14257812  0.08496094  0.33984375  0.1484375  -0.34375    -0.1
4160156
 -0.06835938 -0.14648438 -0.02844238  0.07421875 -0.07666016  0.1
2695312
  0.05859375 -0.07568359 -0.03344727  0.23632812 -0.16308594  0.1
6503906
  0.1484375  -0.2421875  -0.3515625  -0.30664062  0.00491333  0.1
7675781
  0.46289062  0.14257812 -0.25       -0.25976562  0.04370117  0.3
4960938
  0.05957031  0.07617188 -0.02868652 -0.09667969 -0.01281738  0.0
5859375
 -0.22949219 -0.1953125  -0.12207031  0.20117188 -0.42382812  0.0
6005859
  0.50390625  0.20898438  0.11230469 -0.06054688  0.33203125  0.0
7421875
 -0.05786133  0.11083984 -0.06494141  0.05639648  0.01757812  0.0
8398438
  0.13769531  0.2578125   0.16796875 -0.16894531  0.01794434  0.1
6015625
  0.26171875  0.31640625 -0.24804688  0.05371094 -0.0859375   0.1
7089844
 -0.39453125 -0.00156403 -0.07324219 -0.04614258 -0.16210938 -0.1
5722656
  0.21289062 -0.15820312  0.04394531  0.28515625  0.01196289 -0.2
6953125
 -0.04370117  0.37109375  0.04663086 -0.19726562  0.3046875  -0.3
6523438
 -0.23632812  0.08056641 -0.04248047 -0.14648438 -0.06225586 -0.0
534668
 -0.05664062  0.18945312  0.37109375 -0.22070312  0.04638672  0.0
2612305
 -0.11474609  0.265625   -0.02453613  0.11083984 -0.02514648 -0.1
2060547
  0.05297852  0.07128906  0.00063705 -0.36523438 -0.13769531 -0.1
2890625]

The embedding has a shape of: (300,)
```

Finding similar words [1 pt]

GenSim provides a simple way out of the box to find the most similar words to a given word. Test this out below.

In [ ]:
```
print("Finding top 5 similar words to 'hello'")
print(word_vectors.most_similar(["hello"], topn=5))
print("\n")

print("Finding similarity between 'hello' and 'goodbye'")
print(word_vectors.similarity("hello", "goodbye"))
```

```
Finding top 5 similar words to 'hello'
[('hi', 0.654898464679718), ('goodbye', 0.639905571937561), ('how
dy', 0.6310957074165344), ('goodnight', 0.5920578241348267), ('gr
eeting', 0.5855878591537476)]


Finding similarity between 'hello' and 'goodbye'
0.6399056
```

For quantifying simiarity between words based on their respective word vectors, a common metric is cosine similarity (https://en.wikipedia.org/wiki/Cosine_similarity). Formally the cosine similarity s between two vectors a and b, is defined as:

$$s = \frac{a \cdot b}{||a||||b||}, \text{where } s \in [-1, 1]$$

Write your own implementation (using only numpy) of cosine similarity and confirm that it produces the same result as the similarity method available out of the box from GenSim. [0.5 pt]

In [ ]:
```
def cosine_similarity(vector1, vector2):
    ### YOUR CODE BELOW
    return np.dot(vector1,vector2)/(np.sqrt(np.sum(np.square(vect
or1)))*np.sqrt(np.sum(np.square(vector2))))




    ### YOUR CODE ABOVE
```

In [ ]:
```
gensim_similarity = word_vectors.similarity("hello", "goodbye")
custom_similarity = cosine_similarity(word_vectors['hello'], word
_vectors['goodbye'])
print("GenSim implementation: {}".format(gensim_similarity))
print("Your implementation: {}".format(custom_similarity))

assert np.isclose(gensim_similarity, custom_similarity), 'Compute
d similarity is off from the desired value.'
```

```
GenSim implementation: 0.639905571937561
Your implementation: 0.6399056315422058
```

Additionally, implement two other similarity metrics (using only numpy): L1 similarity (https://en.wikipedia.org/wiki/Taxicab_geometry) (Manhattan distance) and L2 similarity (https://en.wikipedia.org/wiki/Euclidean_distance) (Euclidean distance). [0.5 pt]

In [ ]:
```
def L1_similarity(vector1, vector2):
    sum = 0
    ### YOUR CODE BELOW
    for i in range(len(vector1)):
        sum += abs(vector1[i] - vector2[i])
    return sum
    ### YOUR CODE ABOVE

def L2_similarity(vector1, vector2):
    ### YOUR CODE BELOW
    sum = 0
    for i in range(len(vector1)):
        sum += pow(abs(vector1[i] - vector2[i]),2)
    return np.sqrt(sum)
    ### YOUR CODE ABOVE
```

In [ ]:
```
cosine_score = cosine_similarity(word_vectors['hello'], word_vectors['goodbye'])
L1_score = L1_similarity(word_vectors['hello'], word_vectors['goodbye'])
L2_score = L2_similarity(word_vectors['hello'], word_vectors['goodbye'])
print("Cosine similarity: {}".format(cosine_score))
print("L1 similarity: {}".format(L1_score))
print("L2 similarity: {}".format(L2_score))

assert np.isclose(cosine_score, 0.63990), 'Cosine similarity is off from the desired value.'
assert np.isclose(L1_score, 40.15768), 'L1 similarity is off from the desired value.'
assert np.isclose(L2_score, 2.88523), 'L2 similarity is off from the desired value.'
```

```
Cosine similarity: 0.6399056315422058
L1 similarity: 40.157687187194824
L2 similarity: 2.8852380361647056
```

Exploring synonymns and antonyms [2 pts]

In general, you would expect to have a high similarity between synonyms and a low similarity score between antonyms. For e.g. "pleasant" would have a higher similarity score to "enjoyable" as compared to "unpleasant".

In [ ]:
```
print("Similarity between synonyms- 'pleasant' and 'enjoyable': {}".format(word_vectors.similarity("pleasant", "enjoyable")))
print("Similarity between antonyms- 'pleasant' and 'unpleasant': {}".format(word_vectors.similarity("pleasant", "unpleasant")))
```

```
Similarity between synonyms- 'pleasant' and 'enjoyable': 0.6838439702987671
Similarity between antonyms- 'pleasant' and 'unpleasant': 0.6028146743774414
```

However, counter-intuitievely this is not always the case. Often, the similarity score between a word and its antonym is higher than the similarity score with its synonym. For e.g. "sharp" has a giher similarity score with "blunt" as compared to "pointed".

Find two sets of words {w, w_s, w_a} such that {w, w_s} are synonyms and {w, w_a} are antonyms, which have intuitive similarity scores with synonyms and antonyms (synonym_score > antonym_score). [0.5 pts]

Find two sets of words {w, w_s, w_a} such that {w, w_s} are synonyms and {w, w_a} are antonyms, which have counter-intuitive similarity scores with synonyms and antonyms (antonym_score > synonym_score). [0.5 pts]

In [ ]:
```
print("Similarity between synonyms- 'sharp' and 'pointed': {}".fo
rmat(word_vectors.similarity("sharp", "pointed")))
print("Similarity between antonyms- 'sharp' and 'blunt': {}".form
at(word_vectors.similarity("sharp", "blunt")))

### YOUR EXAMPLES BELOW
# words = list(word_vectors.index_to_key)
word_set_1 = ['begin','start','end']
word_set_2 = ['girl','woman','man']
word_set_3 = ['warm','hot','cold']
word_set_4 = ['dark','black','light']
### YOUR EXAMPLES ABOVE

print("For word set 1:")
syn_score, ant_score = word_vectors.similarity(word_set_1[0], wor
d_set_1[1]), word_vectors.similarity(word_set_1[0], word_set_1
[2])
print("Synonym similarity {} - {}: {}".format(word_set_1[0], word
_set_1[1], syn_score))
print("Antonym similarity {} - {}: {}".format(word_set_1[0], word
_set_1[2], ant_score))
assert syn_score > ant_score, 'word_set_1 is not a valid word set
'

print("For word set 2:")
syn_score, ant_score = word_vectors.similarity(word_set_2[0], wor
d_set_2[1]), word_vectors.similarity(word_set_2[0], word_set_2
[2])
print("Synonym similarity {} - {}: {}".format(word_set_2[0], word
_set_2[1], syn_score))
print("Antonym similarity {} - {}: {}".format(word_set_2[0], word
_set_2[2], ant_score))
assert syn_score > ant_score, 'word_set_2 is not a valid word set
'

print("For word set 3:")
syn_score, ant_score = word_vectors.similarity(word_set_3[0], wor
d_set_3[1]), word_vectors.similarity(word_set_3[0], word_set_3
[2])
print("Synonym similarity {} - {}: {}".format(word_set_3[0], word
_set_3[1], syn_score))
print("Antonym similarity {} - {}: {}".format(word_set_3[0], word
_set_3[2], ant_score))
assert ant_score > syn_score, 'word_set_3 is not a valid word set
'

print("For word set 4:")
syn_score, ant_score = word_vectors.similarity(word_set_4[0], wor
d_set_4[1]), word_vectors.similarity(word_set_4[0], word_set_4
[2])
print("Synonym similarity {} - {}: {}".format(word_set_4[0], word
_set_4[1], syn_score))
print("Antonym similarity {} - {}: {}".format(word_set_4[0], word
_set_4[2], ant_score))
assert ant_score > syn_score, 'word_set_4 is not a valid word set
'
```

```
Similarity between synonyms- 'sharp' and 'pointed': 0.19262400269
508362
Similarity between antonyms- 'sharp' and 'blunt': 0.4294208288192
749
For word set 1:
Synonym similarity begin - start: 0.685394823551178
Antonym similarity begin - end: 0.3477645814418793
For word set 2:
Synonym similarity girl - woman: 0.7494640946388245
Antonym similarity girl - man: 0.5921713709831238
For word set 3:
Synonym similarity warm - hot: 0.43215373158454895
Antonym similarity warm - cold: 0.5953035354614258
For word set 4:
Synonym similarity dark - black: 0.3987771272659302
Antonym similarity dark - light: 0.47133004665374756
```

What do you think is the reason behind this? Look at how the word2vec model is trained and explain your reasoning. [1 pts]

Space for answer

Exploring analogies [2 pts]

The Distributional Hypothesis which says that words that occur in the same contexts tend to have similar meanings, leads to an interesting property which allows us to find word analogies like "king" - "man" + "woman" = "queen".

We can exploit this in GenSim like so-

In [ ]:
```
word_vectors.most_similar(positive=['woman', 'king'], negative=['
man'], topn=1)
```

Out[ ]:
```
[('queen', 0.7118192911148071)]
```

In the above, the analogy `man:king::woman:queen` holds true even when looking at the word embeddings.

Find two more such analogies that hold true when looking at embeddings. Write your analogy in the form of `a:b::c:d`, and check that `word_vectors.most_similar(positive=[c, b], negative=[a], topn=1)` produces d. [0.5 pts]

Find two cases where the analogies do not hold true when looking at embeddings. Write your analogy in the form of `a:b::c:d`, and check that `word_vectors.most_similar(positive=[c, b], negative=[a], topn=10)` does not have d. [0.5 pts]

In [ ]:
```
### YOUR EXAMPLES BELOW
a1='bull'
b1='cow'
c1='man'
d1='woman'

a2='football'
b2='footballer'
c2='surf'
d2='surfer'

### YOUR EXAMPLES ABOVE

assert(word_vectors.most_similar(positive=[c1, b1], negative=[a
1], topn=1))[0][0] == d1, "example 1 invalid"
assert(word_vectors.most_similar(positive=[c2, b2], negative=[a
2], topn=1))[0][0] == d2, "example 2 invalid"

### YOUR EXAMPLES BELOW
a3='doctor'
b3='pill'
c3='teacher'
d3='blackboard'

a4='spain'
b4='barcelona'
c4='usa'
d4='chicago'
### YOUR EXAMPLES ABOVE

matches3 = [x for x,y in word_vectors.most_similar(positive=[c3,
b3], negative=[a3], topn=10)]
matches4 = [x for x,y in word_vectors.most_similar(positive=[c4,
b4], negative=[a4], topn=10)]

assert d3 not in matches3, "example 3 invalid"
assert d4 not in matches4, "example 4 invalid"
```

Why do you think some analogies work out while some do not? What might be the reason for this? [1 pts]

Analogies might not work beacause of the training process itself. Maybe in the training process the randomized negative word contexts affect the final word representations and we don't find exactly the relationship later. Also, the positive examples that we need for certain results to be produced might not happen in this given corpus. However, in some of the examples I tried before, the result I expected was not on the top option for the most similar but it was among the top 10.

Exploring Bias [2.5 pts]

Often, bias creeps into word embeddings. This may be gender, racial or ethnic bias. Let us look at an example-

`man:doctor::woman:?`

gives high scores for "nurse" and "gynecologist", revealing the underlying gender stereotypes within these job roles.

In [ ]:
```
word_vectors.most_similar(positive=["woman", "doctor"], negative=
["man"], topn=10)
```

Out[ ]:
```
[('gynecologist', 0.7093892097473145),
 ('nurse', 0.647728681564331),
 ('doctors', 0.6471461057662964),
 ('physician', 0.64389967918396),
 ('pediatrician', 0.6249487996101379),
 ('nurse_practitioner', 0.6218314170837402),
 ('obstetrician', 0.6072014570236206),
 ('ob_gyn', 0.5986712574958801),
 ('midwife', 0.5927063226699829),
 ('dermatologist', 0.5739566087722778)]
```

Provide two more examples that reveal some bias in the word embeddings. Look at the top-5 matches and justify your examples. [1.5 pts]

In [ ]:
```
### YOUR EXAMPLES BELOW
# Demonstrate bias that woman dedicates to chore related activiti
es
a1='he'
b1='works'
c1='she'
d1='works'
# Demonstrate that each gender has a color
a2='he'
b2='blue'
c2='she'
d2='pink'
### YOUR EXAMPLES ABOVE
matches1 = word_vectors.most_similar(positive=[c1, b1], negative=
[a1], topn=5)
matches2 = word_vectors.most_similar(positive=[c2, b2], negative=
[a2], topn=5)
print("{}:{}::{}:?".format(a1,b1,c1))
print(word_vectors.most_similar(positive=[c1, b1], negative=[a1],
topn=5))

print("\n{}:{}::{}:?".format(a2,b2,c2))
print(word_vectors.most_similar(positive=[c2, b2], negative=[a2],
topn=5))

assert d1 not in matches1, "example 3 invalid"
assert d2 not in matches2, "example 4 invalid"
```

```
he:works::she:?
[('Mae_Weems', 0.436980277299881), ('busies_herself', 0.433944880
9623718), ('Cosmetologist', 0.42960768938064575), ('accomplished_
seamstress', 0.4282073676586151), ('busying_herself', 0.427336990
83328247)]

he:blue::she:?
[('pink', 0.6348273158073425), ('purple', 0.5983327627182007), ('
periwinkle_blue', 0.5695484280586243), ('Elie_Saab_dress', 0.5683
978796005249), ('fuchsia_colored', 0.5631670951843262)]
```

Why do you think such bias exists? [1 pt]

I think it happens because the texts that are used to train the algorithms might be older and they contain those biases. As a result, the words like blue and man appear close to each other and they are reflected on the weights.

Visualizing Embeddings [2.5 pts]

Since the word embeddings have a dimension of 300, it is not possible to visualize them directly. However, we can apply a dimension reduction technique like tSNE to reduce the dimensionality of the embeddings to 2-D and then plot them.

Visualizing embeddings in this manner allows us to observe semantic and syntactic similarity of words graphically. Words that are similar to each other appear closer to each other on the tSNE plot.

Let us begin by loading a smaller dataset and applying the Word2Vec model on that corpus. GenSim has a list of datasets available along with a simple_preprocess utility. You can choose any dataset here for your purpose.

We define a `CustomCorpus` class that compiles and loads a dataset of Obama's transcripts (from here (https://github.com/nlp-compromise/nlp-corpus/tree/master/src/sotu)) and provides it to the Word2Vec model. We then use this model for our tSNE plot later.

In [ ]:
```
from gensim.models.word2vec import Word2Vec
from gensim.test.utils import datapath
from gensim import utils



class CustomCorpus(object):
    """An interator that yields sentences (lists of str)."""

    def __iter__(self):
        # Loading dataset
        import urllib.request
        commit = "6c87fd90508c544e340d88c2ca38d1126832f055"
        urls = [
            f"https://raw.githubusercontent.com/nlp-compromise/nl
p-corpus/{commit}/sotu/Obama_{year}.txt" for year in range(2009,
2016)
        ]

        compiled = []
        for url in urls:
            for line in urllib.request.urlopen(url):
                compiled.append(line)

        # For each line in dataset, yield the preprocessed line
        for line in compiled:
            yield utils.simple_preprocess(line)

model = Word2Vec(sentences=CustomCorpus(), size=100)
```

In the code below, complete the method to generate the tSNE plot, given the word vectors. You may use `sklearn.manifold.TSNE` for this purpose. The `generate_tSNE` method takes as input the original word embedding matrix with shape=(VOCAB_SIZE, 100) and reduces it into a 2-D word embedding matrix with shape=(VOCAB_SIZE, 2). [1.25 pts]

In [ ]:
```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
import random

def generate_tSNE(vectors):
    # for i in range(len(vectors)):
    #     vectors[i] = vectors[i].astype(float)
    vocab_size = vectors.shape[0]
    print("Vocab size: {}".format(vocab_size))
    assert vectors.shape[1] == 100


    ### YOUR CODE BELOW    print(vectors)
    tsne_transformed_vectors = TSNE(n_components=2).fit_transform
((vectors))
    ### YOUR CODE ABOVE

    assert tsne_transformed_vectors.shape[1] == 2
    assert tsne_transformed_vectors.shape[0] == vocab_size
    return tsne_transformed_vectors

tsne = generate_tSNE(model.wv[model.wv.vocab])
```

```
Vocab size: 1210
```

Let us plot the result and add labels for a few words on the plot. You can edit the `must_include` list to mandatorily include a few words you want to base your inferences on.

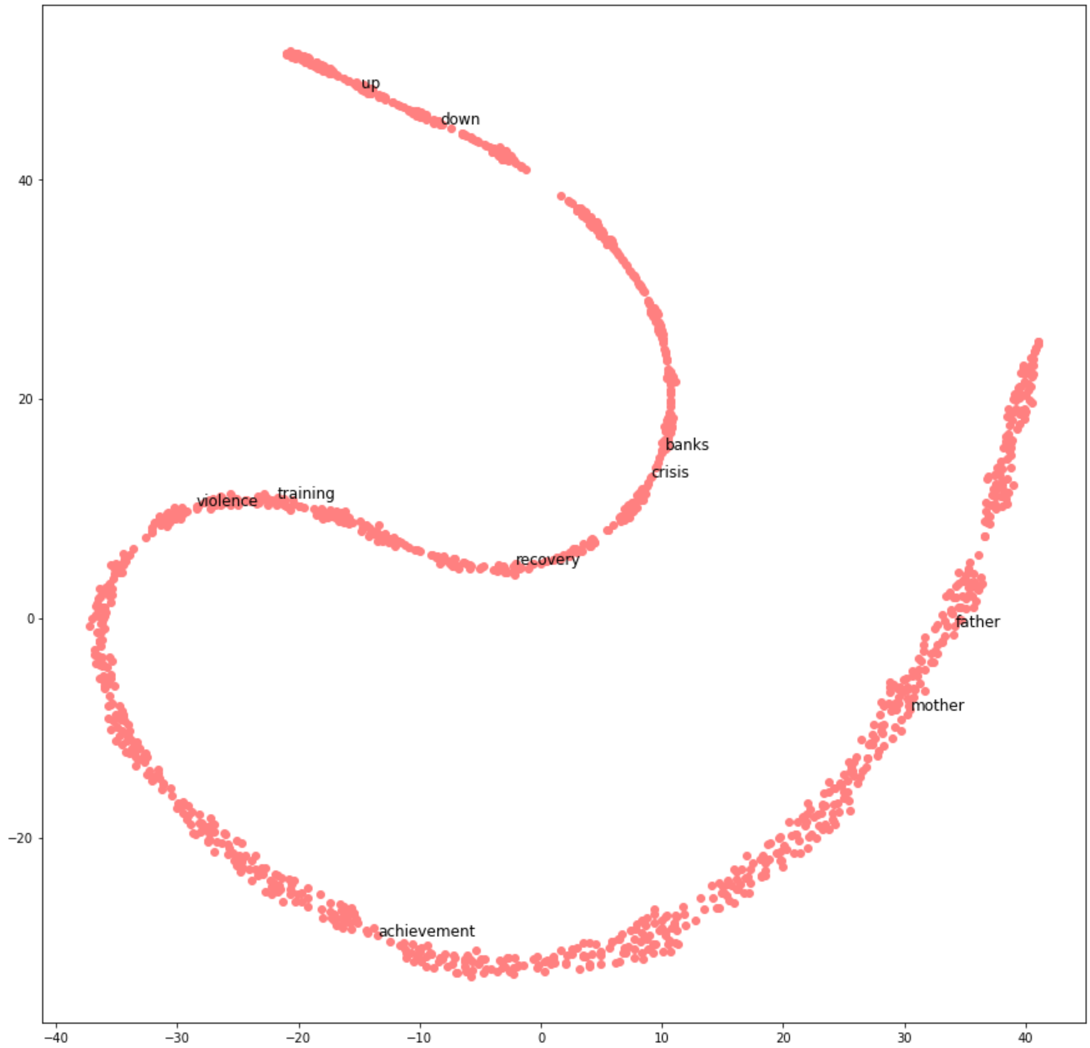
From the tSNE plot, draw inferences for 5 pairs of words, for why they appear close to each other or far apart. Explain your observations with reasoning. [1.25 pts]

In [ ]:
```
def plot_with_matplotlib(x_vals, y_vals, words, must_include, ran
dom_include):
    plt.figure(figsize=(15, 15))
    plt.scatter(x_vals, y_vals, color=[1., 0.5, 0.5])

    indices = list(range(len(words)))
    random.seed(1)
    selected_indices = []#random.sample(indices, random_include)

    selected_indices.extend([i for i in indices if words[i] in mu
st_include])
    for i in selected_indices:
        plt.annotate(words[i], (x_vals[i], y_vals[i]), fontsize=1
2)

must_include = ['training','achievement','crisis','recovery','fat
her','mother','down','up','banks','violence']
plot_with_matplotlib(tsne[:, 0], tsne[:, 1], list(model.wv.vocab.
keys()), must_include, random_include=100)
```



For this question I chose several pairs of words. Some masculine femenine relationships, which usually give high similarity like mother and father. Also I selected up and down to have some antonyms as well. All these words were pretty close on the plot. Also I chose crisis and recovery which a priori seemed to be related to me, but the embeddings don't reflect that. For words unrelated I chose violence and banks which are the furthest away pair from all the before mentioned but I found another pair really related: crisis and banks.