1 (a)

$tf_{t,d}$ (t: word, d: Document) $= \log_{10}(\text{count}(t,d) + 1)$

$idf_t = \log_{10} \frac{N}{df_t}$ (N: total # of documents; df_t: t occurs in # of documents)

$w_{t,d} = tf_{t,d} \times idf_t$

Document 1:

car $tf_{t,d} = \log_{10}(28+1)$ $idf_t = \log_{10}(\frac{20}{13})$

$w_{t,d} = \log_{10}(29) \times \log_{10}(\frac{20}{13})$

auto $w_{t,d} = \log_{10}(0+1) \times \log_{10}(\frac{20}{11})$

insurance $w_{t,d} = \log_{10}(4+1) \times \log_{10}(\frac{20}{7})$

best $w_{t,d} = \log_{10}(15+1) \times \log_{10}(\frac{20}{17})$

Document 2

car $w_{t,d} = \log_{10}(5+1) \times \log_{10}(\frac{20}{13})$

auto $w_{t,d} = \log_{10}(34+1) \times \log_{10}(\frac{20}{11})$

insurance $w_{t,d} = \log_{10}(18+1) \times \log_{10}(\frac{20}{7})$

best $w_{t,d} = \log_{10}(0+1) \times \log_{10}(\frac{20}{17})$

Document 3

car $w_{t,d} = \log_{10}(25+1) \times \log_{10}(\frac{20}{13})$

auto $w_{t,d} = \log_{10}(30+1) \times \log_{10}(\frac{20}{11})$

insurance $w_{t,d} = \log_{10}(0+1) \times \log_{10}(\frac{20}{7})$

best $w_{t,d} = \log_{10}(18+1) \times \log_{10}(\frac{20}{17})$

The exact result is as follows:

```
tf-idf weight of word : car  in Document 1
tf-idf is 0.2735951326791240 5
=====
tf-idf weight of word : car  in Document 2
tf-idf is 0.1455817054584407 6
=====
tf-idf weight of word : car  in Document 3
tf-idf is 0.2647226141116811
=====
tf-idf weight of word : insurance  in Document 1
tf-idf is 0.3186827610174174 4
=====
tf-idf weight of word : insurance  in Document 2
tf-idf is 0.5931811502820314
=====
tf-idf weight of word : insurance  in Document 3
tf-idf is 0.0
=====
tf-idf weight of word : auto  in Document 1
tf-idf is 0.0
=====
tf-idf weight of word : auto  in Document 2
tf-idf is 0.4008976742729881
=====
tf-idf weight of word : auto  in Document 3
tf-idf is 0.3872131391784394
=====
tf-idf weight of word : best  in Document 1
tf-idf is 0.0849880819447423 9
=====
tf-idf weight of word : best  in Document 2
tf-idf is 0.0
=====
tf-idf weight of word : best  in Document 3
tf-idf is 0.0902558029019673
=====
```

1 (b)

According to 1 (a), we can get vectors representing these 3 documents:

```
array([[0.27359513, 0.31868276, 0.        , 0.08498808],
       [0.14558171, 0.59318115, 0.40089767, 0.        ],
       [0.26472261, 0.        , 0.38721314, 0.0902558 ]])
```

document 1
2
3

$$\cos(\vec{v}, \vec{w}) = \frac{\vec{v} \cdot \vec{w}}{|\vec{v}||\vec{w}|}$$

Calculate cosine similarity by the following codes.

```
# calculate the distance
from numpy import linalg as LA
document_list = ['1', '2', '3']
for i, doc in enumerate(document_list):
    for j, doc in enumerate(document_list):
        if i < j:
            print('Similarity between Document', i, ' and Document', j, 'is')
            sim = document_vector[i].dot(document_vector[j]) / \
            (LA.norm(document_vector[i]) * LA.norm(document_vector[j]))
            print(sim)
```

```
Similarity between Document 0  and Document 1 is
0.7310120360347362
Similarity between Document 0  and Document 2 is
0.39131092911382215
Similarity between Document 1  and Document 2 is
0.5552547264235028
```

1 cd)

Firstly, TF-IDF will benefit from stemming a lot. For example, "happy" and "happiness" will be considered as one, "happi" After stemming, stem count will replace tf and idf will be higher as well. So TF-IDF is much easier to determine the weight of one word.

Second, Word2Vec cannot benefit from stemming. Because the goal of word2vec is to predict. We need to consider the word's part of speech, i.e. noun or adj.

2 (a)

$\because$ Cross-Entropy $(p, q) = -\sum_m p_m \log(q_m)$

$\therefore$ Cross-Entropy $(y, \hat{y}) = -\sum_m y_m \log(\hat{y}_m)$

$\because$ y is the ground truth distribution, and it is a one-hot vector, so we can assume for word w_c, jth entry is 1.

So Cross-Entropy $(y, \hat{y}) = -y_j \log(\hat{y}_j)$

Here $\hat{y}_j = P(C = w_c \mid T = w_t)$ and $y_j = 1$

So Cross-Entropy $(y, \hat{y}) = \boxed{-\log(P(C = w_c \mid T = w_t))}$

2(b) $\text{Cross-Entropy}(y, \hat{y}) = -\log\left(P(C = w_c \mid T = w_t)\right)$

$$= -\log\left(\frac{e^{(u_{w_c}^T \cdot v_{w_t})}}{\sum_{w' \in V} e^{u_{w'}^T \cdot v_{w_t}}}\right)$$

$$= -\log e^{u_{w_c}^T \cdot v_{w_t}} + \log \sum_{w' \in V} e^{u_{w'}^T \cdot v_{w_t}}$$

We define $L = \text{Cross-Entropy}(y, \hat{y})$

$$\frac{\partial L}{\partial v_{w_t}} = \frac{\partial\left(\log \sum_{w' \in V} e^{u_{w'}^T \cdot v_{w_t}}\right)}{\partial v_{w_t}} - \frac{\partial\left(\log e^{u_{w_c}^T \cdot v_{w_t}}\right)}{\partial v_{w_t}}$$

$$= \frac{\partial \log \sum_{w' \in V} e^{u_{w'}^T v_{w_t}}}{\partial \sum_{w' \in V} e^{u_{w'}^T v_{w_t}}} \cdot \frac{\partial \sum_{w' \in V} e^{u_{w'}^T v_{w_t}}}{\partial v_{w_t}} - \frac{\partial (u_{w_c}^T \cdot v_{w_t})}{\partial v_{w_t}}$$

Chain Rule

$$= \frac{\sum_{w' \in V} e^{u_{w'}^T \cdot v_{w_t}} \cdot u_{w'}^T}{\sum_{w' \in V} e^{u_{w'}^T \cdot v_{w_t}}} - u_{w_c}^T$$

$\hat{y}$

$$= \sum_{w' \in V} u_{w'}^T \cdot \hat{y} - u_{w_c}^T$$

$$= U^T(\hat{y} - y)$$

2 (c) ① $C = w_c$

From 2(b), we have $L = -\log e^{u_{w_c}^T \cdot v_{w_t}} + \log \sum_{w' \in V} e^{u_{w'}^T \cdot v_{w_t}}$

$$\frac{\partial L}{\partial u_{w_c}} = \frac{\partial \log\left(\sum_{w' \in V} e^{u_{w'}^T \cdot v_{w_t}}\right)}{\partial u_{w_c}} - \frac{\partial \log e^{u_{w_c}^T \cdot v_{w_t}}}{\partial u_{w_c}}$$

$$= \underbrace{\frac{e^{u_{w_c} v_{w_t}} \cdot v_{w_t}}{\sum_{w' \in V} e^{u_{w'}^T \cdot v_{w_t}}}}_{\hat{y}_c} - v_{w_t}$$

$$= \hat{y}_c \cdot v_{w_t} - v_{w_t}$$

$$= (\hat{y}_c - 1)\, v_{w_t}$$

↑ c is the index of Content C

② $C \neq w_c$

$$\frac{\partial L}{\partial u_{w_{c'}}} = \frac{\partial \log\left(\sum_{w' \in V} e^{u_{w'}^T \cdot v_{w_t}}\right)}{\partial u_{w_{c'}}} - \frac{\partial \log e^{u_{w_c}^T \cdot v_{w_t}}}{\partial u_{w_{c'}}}$$

constant for $u_{w_{c'}}$

$$= \underbrace{\frac{e^{u_{w_{c'}}^T \cdot v_{w_t}} \cdot v_{w_t}}{\sum_{w' \in V} e^{u_{w'}^T v_{w_t}}}}_{\hat{y}_{c'}}$$

$$= \hat{y}_{c'} \cdot v_{w_t}$$

↑ c' is the index of the word we are calculating now

We can summarize $\frac{\partial L}{\partial w_c} = (\hat{y}_c - y_w)\, v_{w_t}$ where $y_w = \begin{cases} 1, & c = w_c \\ 0, & c \neq w_c \end{cases}$

3 (a)

SPACE: Assume each entry in a skip-gram vector occupies k bytes

Originally: $(100{,}000 \times 300 \times 2)k$

There are $52+10 = 62$ more characters

Each one has a 300-long vector. to represent.

Hence need extra $62 \times 300 = 18{,}600$ space.

Updated: $(100{,}000 \times 300 \times 2 + (52+10) \times 300)k$

Percentage: $(52+10) \times 300 / 100{,}000 \times 300 \times 2 = 0.031\%$

TIME: Assuming each operation takes t amount of time

Originally; $(2 \times 10{,}000{,}000)/10 = 2{,}000{,}000t$

Average length is 5.5. Should need 5.5x time

Updated.: $2 \times (10{,}000{,}000 + 10{,}000{,}000 \times 5.5)/10 = 13{,}000000$

$\Rightarrow$ Percentage = 550%

CHANGE WINDOW SIZE.

Original: $2 \times 300 \times (5 \times 2)$ (epochs, hs., window)

$\parallel$

Now $1 \times 200 \times (? \times 2) \times (5.5+1)$

So we can know window size should change to $2.307 \approx 2$.

3(b) We can begin with negative algorithm.

Algorithm 1: Negative Sampling.

ts = the training set of word sequence, w =window size, k = number of negative samples we choose.

Intialize W_{word}, V_{word}

Intialize W_{affix}. V_{affix}

Initialize set of training examples te

generate negative samples.

for te[i]. $0 \le i \le |V|$ do

    for $i-w \le j \le i+w$, $j \ne i$. do

        Randomly select k words $\ne$ te[j]

        For each of these k words, te $\leftarrow$ (word, te[i], 0)

    end

end

gradient descent

for # epoch do

    for ((context_word, target_word), label) in te do

        Find the affix of the target word

        Find $V_{affix}[a_t]$ for the target word

        Find the affix of context word, $W_{affix}[a_c]$ for the context word.

        Perform the average.

        Gradient Desent.

        $V_{affix}[a_t]$;

    end

end

Heme, in order to do this calculation,

① Hidden dimensions of the embeddings

② Window size.

③ # of epochs

④ Time to take the average.

⑤ # of words has multiple suffix or preffix

3(c) When we do negative sampling, we will choose random words to make a sample (w_1, w_2)

But w_1 may have a prefix and a suffix. , totally 2

w_1 may have 2 suffixs

But sometimes it only has one prefix. or suffix , only 1

Because of this randomness, we cannot get the exact calculation.

YES.

Such random factors do NOT exists in these two models.

so changing the regime would remove the uncertainty

4(a) (i) Because ELMo is using character-level CNN, so it can capture the relationship between characters, not only word level. So when there comes a out-of-bag word, ELMo can still handle it, because ELMo see these characters before

(ii) Even though ELMo never sees the out-of-vocabulary word. But it may see the content before.

For example, "I play Fidditch at school today". "Fidditch" is a out-of-vocubulary word, but according to the context, ELMo can still induce it is a game.

ELMo is contexualized, word-sense inference can still be deduced from the context words.

4(b) $PE(x) = [\sin x, \cos x]^T$

$PE(x+k) = [\sin(x+k), \cos(x+k)]^T$

$= [\sin x \cos k + \cos x \sin k, \cos x \cos k - \sin x \sin k]^T$

$= \underbrace{\begin{bmatrix} \cos k & \sin k \\ -\sin k & \cos k \end{bmatrix}}_{A} \cdot [\sin x, \cos x]^T$

$\therefore A = \begin{bmatrix} \cos k & \sin k \\ -\sin k & \cos k \end{bmatrix}$

4cc) BERT: Masked Language Modeling

Its objective is to learn the context of word

BERT can be used to do classification and Question Answering.

GPT2: Autoregressive language model

Objective: Predict next token given the previous tokens

Advantage of Masked LM.

① Can do classification.

② Can extract the info based on context

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)
- GloVe (by Stanford)
- fastText (by Facebook)

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). For this, we will be using the GenSim package, find documentation here. 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-30
          + "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-news-300.
Note: This can take a few minutes.
```

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

In [2]:
```
print(word_vectors['hello'])
print("\nThe embedding has a shape of: {}".format(word_vectors['hello'].shape))
```

```
[-0.05419922  0.01708984 -0.00527954  0.33203125 -0.25       -0.01397705
```

```
-0.15039062 -0.265625    0.01647949  0.3828125  -0.03295898 -0.09716797
-0.16308594 -0.04443359  0.00946045  0.18457031  0.03637695  0.16601562
 0.36328125 -0.25585938  0.375       0.171875    0.21386719 -0.19921875
 0.13085938 -0.07275391 -0.02819824  0.11621094  0.15332031  0.09082031
 0.06787109 -0.0300293  -0.16894531 -0.20800781 -0.03710938 -0.22753906
 0.26367188  0.012146    0.18359375  0.31054688 -0.10791016 -0.19140625
 0.21582031  0.13183594 -0.03515625  0.18554688 -0.30859375  0.04785156
-0.10986328  0.14355469 -0.43554688 -0.0378418   0.10839844  0.140625
-0.10595703  0.26171875 -0.17089844  0.39453125  0.12597656 -0.27734375
-0.28125     0.14746094 -0.20996094  0.02355957  0.18457031  0.00445557
-0.27929688 -0.03637695 -0.29296875  0.19628906  0.20703125  0.2890625
-0.20507812  0.06787109 -0.43164062 -0.10986328 -0.2578125  -0.02331543
 0.11328125  0.23144531 -0.04418945  0.10839844 -0.2890625  -0.09521484
-0.10351562 -0.0324707   0.07763672 -0.13378906  0.22949219  0.06298828
 0.08349609  0.02929688 -0.11474609  0.00534058 -0.12988281  0.02514648
 0.08789062  0.24511719 -0.11474609 -0.296875   -0.59375    -0.29492188
-0.13378906  0.27734375 -0.04174805  0.11621094  0.28320312  0.00241089
 0.13867188 -0.00683594 -0.30078125  0.16210938  0.01171875 -0.13867188
 0.48828125  0.02880859  0.02416992  0.04736328  0.05859375 -0.23828125
 0.02758789  0.05981445 -0.03857422  0.06933594  0.14941406 -0.10888672
-0.07324219  0.08789062  0.27148438  0.06591797 -0.37890625 -0.26171875
-0.13183594  0.09570312 -0.3125      0.10205078  0.03063965  0.23632812
 0.00582886  0.27734375  0.20507812 -0.17871094 -0.31445312 -0.01586914
 0.13964844  0.13574219  0.0390625  -0.29296875  0.234375   -0.33984375
-0.11816406  0.10644531 -0.18457031 -0.02099609  0.02563477  0.25390625
 0.07275391  0.13574219 -0.00138092 -0.2578125  -0.2890625   0.10107422
 0.19238281 -0.04882812  0.27929688 -0.3359375  -0.07373047  0.01879883
-0.10986328 -0.04614258  0.15722656  0.06689453 -0.03417969  0.16308594
 0.08642578  0.44726562  0.02026367 -0.01977539  0.07958984  0.17773438
-0.04370117 -0.00952148  0.16503906  0.17285156  0.23144531 -0.04272461
 0.02355957  0.18359375 -0.41601562 -0.01745605  0.16796875  0.04736328
 0.14257812  0.08496094  0.33984375  0.1484375  -0.34375    -0.14160156
-0.06835938 -0.14648438 -0.02844238  0.07421875 -0.07666016  0.12695312
 0.05859375 -0.07568359 -0.03344727  0.23632812 -0.16308594  0.16503906
 0.1484375  -0.2421875  -0.3515625  -0.30664062  0.00491333  0.17675781
 0.46289062  0.14257812 -0.25        -0.25976562  0.04370117  0.34960938
 0.05957031  0.07617188 -0.02868652 -0.09667969 -0.01281738  0.05859375
-0.22949219 -0.1953125  -0.12207031  0.20117188 -0.42382812  0.06005859
 0.50390625  0.20898438  0.11230469 -0.06054688  0.33203125  0.07421875
-0.05786133  0.11083984 -0.06494141  0.05639648  0.01757812  0.08398438
 0.13769531  0.2578125   0.16796875 -0.16894531  0.01794434  0.16015625
 0.26171875  0.31640625 -0.24804688  0.05371094 -0.0859375   0.17089844
-0.39453125 -0.00156403 -0.07324219 -0.04614258 -0.16210938 -0.15722656
 0.21289062 -0.15820312  0.04394531  0.28515625  0.01196289 -0.26953125
-0.04370117  0.37109375  0.04663086 -0.19726562  0.3046875  -0.36523438
-0.23632812  0.08056641 -0.04248047 -0.14648438 -0.06225586 -0.0534668
-0.05664062  0.18945312  0.37109375 -0.22070312  0.04638672  0.02612305
-0.11474609  0.265625   -0.02453613  0.11083984 -0.02514648 -0.12060547
 0.05297852  0.07128906  0.00063705 -0.36523438 -0.13769531 -0.12890625]

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

Finding similar words [5 pts]

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

In [3]:
```
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), ('howdy', 0.63109570
74165344), ('goodnight', 0.5920578241348267), ('greeting', 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. 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. [3 pts]

In [4]:
```
def cosine_similarity(vector1, vector2):
    ### YOUR CODE BELOW
    return np.dot(vector1, vector2) / (np.linalg.norm(vector1) * np.linalg.norm(
    ### YOUR CODE ABOVE
```

In [5]:
```
gensim_similarity = word_vectors.similarity("hello", "goodbye")
custom_similarity = cosine_similarity(word_vectors['hello'], word_vectors['goodb
print("GenSim implementation: {}".format(gensim_similarity))
print("Your implementation: {}".format(custom_similarity))

assert np.isclose(gensim_similarity, custom_similarity), 'Computed similarity is
```

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

Additionally, implement two other similarity metrics (using only numpy): L1 similarity (Manhattan distance) and L2 similarity (Euclidean distance). [2 pts]

In [13]:
```
def L1_similarity(vector1, vector2):
    ### YOUR CODE BELOW
    return np.sum(np.abs(vector1 - vector2))
    ### YOUR CODE ABOVE

def L2_similarity(vector1, vector2):
    ### YOUR CODE BELOW
    return np.sqrt(np.sum(np.power(vector1 - vector2, 2)))
    ### YOUR CODE ABOVE
```

In [14]:
```
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 des
assert np.isclose(L1_score, 40.15768), 'L1 similarity is off from the desired va
assert np.isclose(L2_score, 2.88523), 'L2 similarity is off from the desired val
```

```
Cosine similarity: 0.6399056315422058
```

```
L1 similarity: 40.15768814086914
L2 similarity: 2.8852379322052
```

Exploring synonymns and antonyms [10 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 [15]:
```
print("Similarity between synonyms- 'pleasant' and 'enjoyable': {}".format(word_
print("Similarity between antonyms- 'pleasant' and 'unpleasant': {}".format(word
```

```
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). [4 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). [4 pts]

In [27]:
```
print("Similarity between synonyms- 'sharp' and 'pointed': {}".format(word_vecto
print("Similarity between antonyms- 'sharp' and 'blunt': {}".format(word_vectors

### YOUR EXAMPLES BELOW
word_set_1 = dict()
word_set_1[0] = 'big'
word_set_1[1] = 'huge'
word_set_1[2] = 'small'

word_set_2 = dict()
word_set_2[0] = 'small'
word_set_2[1] = 'tiny'
word_set_2[2] = 'big'

word_set_3 = dict()
word_set_3[0] = 'love'
word_set_3[1] = 'like'
word_set_3[2] = 'hate'

word_set_4 = dict()
word_set_4[0] = 'hot'
word_set_4[1] = 'scalding'
word_set_4[2] = 'cold'
### YOUR EXAMPLES ABOVE

print("For word set 1:")
syn_score, ant_score = word_vectors.similarity(word_set_1[0], word_set_1[1]), wo
print("Synonym similarity {} - {}: {}".format(word_set_1[0], word_set_1[1], syn_
print("Antonym similarity {} - {}: {}".format(word_set_1[0], word_set_1[2], ant_
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], word_set_2[1]), wo
print("Synonym similarity {} - {}: {}".format(word_set_2[0], word_set_2[1], syn_
print("Antonym similarity {} - {}: {}".format(word_set_2[0], word_set_2[2], ant_
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], word_set_3[1]), wo
print("Synonym similarity {} - {}: {}".format(word_set_3[0], word_set_3[1], syn_
print("Antonym similarity {} - {}: {}".format(word_set_3[0], word_set_3[2], ant_
assert ant_score > syn_score, 'word_set_1 is not a valid word set'

print("For word set 4:")
syn_score, ant_score = word_vectors.similarity(word_set_4[0], word_set_4[1]), wo
print("Synonym similarity {} - {}: {}".format(word_set_4[0], word_set_4[1], syn_
print("Antonym similarity {} - {}: {}".format(word_set_4[0], word_set_4[2], ant_
assert ant_score > syn_score, 'word_set_2 is not a valid word set'
```

```
Similarity between synonyms- 'sharp' and 'pointed': 0.19262400269508362
Similarity between antonyms- 'sharp' and 'blunt': 0.4294208288192749
For word set 1:
Synonym similarity big - huge: 0.7809855937957764
Antonym similarity big - small: 0.49586784839630127
For word set 2:
Synonym similarity small - tiny: 0.7187926769256592
Antonym similarity small - big: 0.49586784839630127
For word set 3:
Synonym similarity love - like: 0.3671387732028961
Antonym similarity love - hate: 0.6003956198692322
For word set 4:
Synonym similarity hot - scalding: 0.3972260057926178
Antonym similarity hot - cold: 0.460213840007782
```

What do you think is the reason behind this? Look at how the word2vec model is trained and explain your reasoning. [2 pts]

Answer

- Because the similarity here is more like given this word, the possibility of seeing the other word.
- We should call it relatedness instead of similarity.
- Even though hot has similar meaning with scalding, but they seldom happens together, hence, the similarity from word2vec is lower.

Exploring analogies [10 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 [28]:
```
word_vectors.most_similar(positive=['woman', 'king'], negative=['man'], topn=1)
```

Out[28]:
```
[('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. [4 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. [4 pts]

In [82]:
```
### YOUR EXAMPLES BELOW
c1, b1, a1, d1 = 'female', 'dad', 'male', 'mom'
c2, b2, a2, d2 = 'China', 'Tokyo', 'Japan', 'Beijing'
c3, b3, a3, d3 = 'cold', 'water', 'warm', 'ice'
c4, b4, a4, d4 = 'Georgia', 'Gainesville', 'Florida', 'Athens'
### YOUR EXAMPLES ABOVE

assert(word_vectors.most_similar(positive=[c1, b1], negative=[a1], topn=1))[0][0
assert(word_vectors.most_similar(positive=[c2, b2], negative=[a2], topn=1))[0][0

### YOUR EXAMPLES BELOW
### YOUR EXAMPLES ABOVE

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

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? [2 pts]

In [84]:
```
word_vectors.most_similar(positive=[c4, b4], negative=[a4], topn=10)
```

Out[84]:
```
[('Newnan', 0.724273681640625),
 ('Statesboro', 0.6935749053955078),
 ('Cartersville', 0.6866348385810852),
 ('Watkinsville', 0.6776729822158813),
 ('Macon', 0.6588613986968994),
 ('Flowery_Branch', 0.6550853848457336),
 ('Toccoa', 0.6507533192634583),
 ('Warner_Robins', 0.648144006729126),
 ('Loganville', 0.6371333599090576),
 ('Kennesaw', 0.6352677345275879)]
```

Answer:

- We take the 4th set of analogy as example. What word2vec did is to look for words which are close to Gainesville and close to Georgia, and which are not close to Florida. Newnan is obtained here bebause there's a Newnan in Georgia, and there's also a Newnan's Lake in

Gainesville, FL. However the fact that “Athens”, the right answer, is contextually separated from Georgia by the existence of Athens, Greece.

- So I don’t think the semantic relationship between ‘Gainesville’ and ‘Florida’ is something word2vec is really capturing. It is considering a word similar to the two in positive sets but not relevant with the word in negative set.
- When this word has many different meanings and occur in many different situations, word2vec is hard to do analogy.

Exploring Bias [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 [85]:
```
word_vectors.most_similar(positive=["woman", "doctor"], negative=["man"], topn=1
```

Out[85]:
```
[('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. [4 pts]

In [106…
```
### YOUR EXAMPLES BELOW
c1, b1, a1 = 'US' , 'black', 'Africa'
c2, b2, a2 =  'man', 'housewife', 'woman'
### YOUR EXAMPLES ABOVE

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

```
Africa:black::US:?
[('white', 0.5049769878387451), ('clad_commandos_stormed', 0.4150097370147705),
('gray', 0.37365710735321045), ('POLITICS_Summary', 0.3617469072341919), ('Extre
me_rightists', 0.35885876417160034)]

woman:housewife::man:?
[('schoolteacher', 0.5566020011901855), ('homemaker', 0.5082709193229675), ('sho
```

```
pkeeper', 0.5011745691299438), ('businessman', 0.4810183644294739), ('laborer',
0.4744962155818939)]
```

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

Anwser

- The model is trained on the corpora from Internet or old news, novels. The bias is rooted in the history, so when we train model with old or biased corpora, it is easy to make it exist in the model.

Visualizing Embeddings [10 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) and provides it to the Word2Vec model. We then use this model for our tSNE plot later.

In [1]:
```
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
        urls = ["https://raw.githubusercontent.com/nlp-compromise/nlp-corpus/mas
                "https://raw.githubusercontent.com/nlp-compromise/nlp-corpus/mas
                "https://raw.githubusercontent.com/nlp-compromise/nlp-corpus/mas
                "https://raw.githubusercontent.com/nlp-compromise/nlp-corpus/mas
                "https://raw.githubusercontent.com/nlp-compromise/nlp-corpus/mas
                "https://raw.githubusercontent.com/nlp-compromise/nlp-corpus/mas
                "https://raw.githubusercontent.com/nlp-compromise/nlp-corpus/mas

        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). [5 pts]

In [6]:
```
from sklearn.manifold import TSNE
import matplotlib.pyplot as plt
import random

def generate_tSNE(vectors):
    vocab_size = vectors.shape[0]
    print("Vocab size: {}".format(vocab_size))
    assert vectors.shape[1] == 100

    ### YOUR CODE BELOW
    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
```

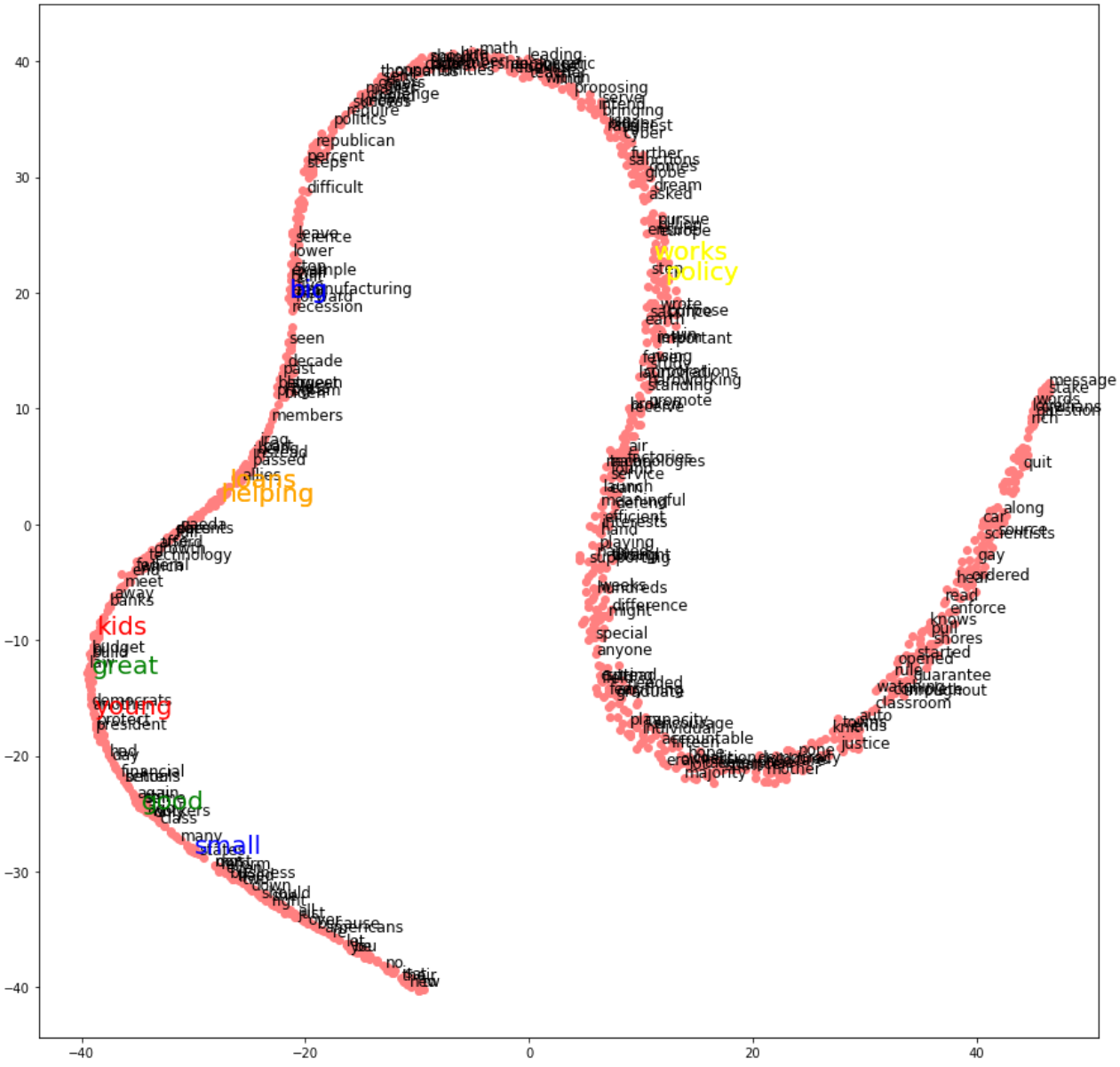
Let us plot the result and add labels for a few words on the plot. You can edit the `must_include` list to mandatorily include a few words you want to base your inferences on.

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. [5 pts]

In [53]:
```
def plot_with_matplotlib(x_vals, y_vals, words, must_include, random_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 must_include])
    extend_indices = [i for i in indices if words[i] in must_include]
    color_list = ['red', 'blue', 'green', 'yellow', 'orange']
    for i in selected_indices:
        if words[i] in must_include:
            plt.annotate(words[i], (x_vals[i], y_vals[i]), fontsize=20, color=co
        else:
            plt.annotate(words[i], (x_vals[i], y_vals[i]), fontsize=12)

must_include = ['kids', 'young', 'big', 'small', 'good', 'great', 'works', 'poli
plot_with_matplotlib(tsne[:, 0], tsne[:, 1], list(model.wv.vocab.keys()), must_i
# print('list(model.wv.vocab.keys())', list(model.wv.vocab.keys()))
```



Answers

- ('works', 'policy'), ('young', 'kids') and ('helping', 'loans') are very close to each other respectively. It is because these two words are often appearing together. So thier similarity is very hign in word2vec, so even though we lower the dimensionality to 2, these 2 words are still close to each other.
- 'small' and 'big' are antonym. So after loweing dimensionality, these 2 vectors are still far away.
- 'good' and 'great' are synonym, so they are close to wach other.

In [ ]: