NLP HW3

~~3~~.a) For example if we take the word
4 "Purge", and since ELMO takes token embedding
~~for~~ like word embedding and character embedding.
I Then we would also have an word embedding for
the word "urge", even though "urge" doesn't
appear in our training data, Hence ~~OOV~~. it
Can handle the OOV Problem.

II

3c. This is impossible because looking at prefix or suffix word, we cannot tell some negative sample will contain such prefix or suffix, so this is hard because we cannot tell in advance.

CBOW will remove this uncertainty because we don't have to worry about ~~the~~ negative sampling.

Also, hierarchical softmax reduces computation by the binary tree and it doesn't care about negative sampling. Because hierarchical softmax is used to approximate the true softmax function.

NLP
HW 3.

3a) new space = $\overset{D}{100{,}000 \times 300} + 62 \times \overset{D}{300}$
$= 30{,}018{,}600$ weights

old space $= 100{,}000 \times 300$
So, we increased ~~space~~ space by $\% \, 0.062$

time $= \dfrac{(62/5.5) + 100{,}000}{100{,}000} - 1 \cong \% \, 0.0112$
increased time

The model update $(c+1) \times D$ for each epoch

$|V| = 100{,}000$, $D = 300$, $C = 5$

$(5+1) \times 300 = 1800$ weight get update at each step
So, for 2 epoch, 3600 total weights get update

Let $D = 200$ then $(200)(c+1) = 3600$

$C = 17$, so, window size should be

17.

NLP
HW3

2c $$\frac{\partial \text{ crossentropy}}{\partial u_{w_t}} = \overbrace{\frac{v_{w_t} * \exp(u_{w_c}^T \cdot v_{w_t})}{\sum_{w' \in W} \exp(u_{w'}^T \cdot v_{w_t})}}^{\hat{y}_{w_c}} - v_{w_t}$$

$$= v_{w_t} \hat{y}_{w_c} - v_{w_t}$$

when $w_c = c$

$$= v_{w_t} (\hat{y}_{w_c} - 1)$$

---

when $w_c \neq c$ $$\frac{\partial \text{ crossentropy}}{\partial u_{w_c}} = \underbrace{\frac{v_{w_t} * \exp(u_{w_c}^T \cdot v_{w_t})}{\sum_{w' \in W} \exp(u_{w_c}^T \cdot v_{w_t})}}_{\hat{y}_{w_c}}$$

$$= v_{w_t} \hat{y}_{w_c}$$

NLP
HW3

2b $\text{Cross entropy}(p,q) = -\log_2(P(c=w_c|T=w_t))$

$$= -\log_2\left(\frac{\exp(u_{w_c}^T \cdot v_{w_t})}{\sum_{w'\in V} \exp(u_{w'}^T \cdot v_{w_t})}\right)$$

$$= -\log_2 \exp(u_{w_c}^T \cdot v_{w_t}) + \log_2 \sum_{w'\in V} \exp(u_{w'}^T \cdot v_{w_t})$$

$$= -u_{w_c}^T \cdot v_{w_t} + \log_2 \sum_{w'\in V} \exp(u^T \cdot v_{w_t})$$

$$\partial/\partial v_{w_t} = -u_{w_c}^T + \frac{\sum u^T \cdot v_{w_t} \cancel{\exp(u^T \cdot v_{w_t})}}{\sum \cancel{\exp(u^T \cdot v_{w_t})}}$$

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

$$= -u_{w_c} + U\hat{y}$$

$$= U\hat{y} - Uy$$

NLP
HW3

2a) $\text{Cross entropy}(p,q) = -\sum_{m} p_m \log(q_m)$

$\text{Cross-entropy}(y,\hat{y})$

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

$$= -\log P(C = w_c \mid T = w_t)$$

NLP
HW 3

(In the embeddings)

1d Since Word2Vec already captures relational meaning when trainned properly, It doesn't benefit as much. So in this case tf-idf benefits more from stemming. which would make the sparse model denser.

NLP
HW3

1) b $\text{Cosine}(\vec{v},\vec{w}) = \dfrac{\sum_{i}^{N} v_i w_i}{\sqrt{\sum_i^N v_i^2}\sqrt{\sum_i^N w_i^2}}$

Cosine similarity Doc1, Doc2 =
0.73101

Cosine similarity Doc2 Doc3 =
0.555254

Cosine similarity Doc1 Doc3 =

0.3913067

HW 3
NLP

1a)

$\log_{10}(\text{count}(t,d)+1)$

Tf(t,d)	Doc1	Doc2	Doc3
Car	1.462	0.778	1.414
insurance	0.698	1.301	0
auto	0	1.544	1.491
best	1.204	0	1.278

$Idf_i = \log_{10}\left(\frac{N}{df_i}\right)$

	Idf_i
Car	0.187
insurance	0.455
auto	0.259
best	0.070

$W_{t,d} = tf_{t,d} \times idf_i$

	Doc1	Doc2	Doc3
Car	0.27359	0.14558	0.26472
Insurance	0.31868	0.593181	0
auto	0	0.400897	0.387213
best	0.084988	0	0.090255

3b) Calculate how many tokens on average have a prefix or suffix in your |V|. where V is our vocabulary.

NLP
HW 3

3 a) $|V| = 100{,}000$

training Epoch = 2

hidden dimension $D = 300$

Window Size $C = 5$

$V * D = 30{,}000{,}000$ Weights

Model will update $(C+1) \times 300 =$

$(5+1) \times 300 = 1800$

$= \% \, 0.006$

and if we do $D = 200$

62, is the

$100{,}000 \times 300 + 62 \times 300$

space

4c) Before feeding word sequences into BERT, a certain percentage of the given words in each sequence are replaced with token, so the model tries to predict the value of the masked word based on neighboring non-masked words.

Autoregressive model is a feed-forward model that tries to predict future words in a given context, but the context word is constrained to two direction forward or backward.

The issue with autoregressive LM that BERT doesn't have is that it only takes into account either forward context or backward context which means it has a limited understanding of that context.

4b) since $\sin(x+K) = \sin x \cos K + \cos x \sin K$

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

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

$$= \begin{bmatrix} \sin x \cos K + \cos x \sin K \\ \cos x \sin K + \sin x \cos K \end{bmatrix}$$

$$= \underbrace{\begin{bmatrix} \cos(K), & \sin K \\ -\sin(K) & \cos K \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} \sin x \\ \cos x \end{bmatrix}}_{PE(x)}$$

word_embedding

September 28, 2020

1 NLP Homework 3 Programming Assignment

1.1 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.

1.1.1 Loading word vectors from GenSim

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

[ ]:

```
```

[2]:

```
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-news-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-

[3]:
```
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,)
```

1.1.2 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.

[4]:
```python
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', 0.6548984050750732), ('goodbye', 0.639905571937561), ('howdy',
0.6310957074165344), ('goodnight', 0.5920578241348267), ('greeting',
0.5855877995491028)]


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]

[11]:
```python
from numpy import dot
from numpy.linalg import norm

def cosine_similarity(vector1, vector2):
```

```
    cos_sim = dot(vector1, vector2)/(norm(vector1)*norm(vector2))
    return cos_sim
    ### YOUR CODE BELOW
    ### YOUR CODE ABOVE
```

[12]:
```
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), 'Computed 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 (Manhattan distance) and L2 similarity (Euclidean distance). [2 pts]

[13]:
```
import numpy as np
from numpy.linalg import norm
def L1_similarity(vector1, vector2):
    cos_sim = np.sum(np.abs(vector1 - vector2))
    return cos_sim




    ### YOUR CODE BELOW
    ### YOUR CODE ABOVE

def L2_similarity(vector1, vector2):


    cos_sim = np.sqrt((np.sum((vector2-vector1)**2)))
    return cos_sim



    ### YOUR CODE BELOW
    ### YOUR CODE ABOVE
```

[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␣
↪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: 40.15768814086914
L2 similarity: 2.8852379322052
```

1.1.3 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".

[15]:
```
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': 0.6028147339820862
```

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]

[16]:
```
#word_vectors.index2word
word_vectors.most_similar(positive=['good', 'sad'], negative=['bad'])
```

[16]:
```
[('wonderful', 0.6414927840232849),
 ('happy', 0.6154338121414185),
 ('great', 0.580367922782898),
 ('nice', 0.5683972835540771),
 ('saddening', 0.5588892698287964),
 ('bittersweet', 0.5544660687446594),
 ('glad', 0.551203727722168),
 ('fantastic', 0.5471093654632568),
 ('proud', 0.5305151343345642),
 ('saddened', 0.5293528437614441)]
```

[17]:
```
word_vectors.most_similar(positive=['obliterate'])
```

[17]:
```
[('destroy', 0.7293034791946411),
 ('annihilate', 0.6592815518379211),
 ('erase', 0.6021994352340698),
 ('obliterating', 0.6002026796340942),
 ('shatter', 0.5765564441680908),
 ('wipe', 0.561111569404602),
 ('decimate', 0.557799756526947),
 ('obliterated', 0.5501904487609863),
 ('efface', 0.5477695465087891),
 ('obliterates', 0.546683669090271)]
```

[ ]:
```

```

[29]:
```
print("Similarity between synonyms- 'sharp' and 'pointed': {}".
 ↪format(word_vectors.similarity("sharp", "pointed")))
print("Similarity between antonyms- 'sharp' and 'blunt': {}".format(word_vectors.
 ↪similarity("sharp", "blunt")))

### YOUR EXAMPLES BELOW
word_set_1=['destroy','obliterate','create']
word_set_2=['cut','slash','sew']
word_set_3 = ['hate', 'loathe','love']
word_set_4=['sad','unhappy','happy']


### YOUR EXAMPLES ABOVE

print("For word set 1:")
syn_score, ant_score = word_vectors.similarity(word_set_1[0], word_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], word_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], word_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], word_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_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 destroy - obliterate: 0.7293034791946411
Antonym similarity destroy - create: 0.3772240877151489
For word set 2:
Synonym similarity cut - slash: 0.7417387366294861
Antonym similarity cut - sew: 0.26631611585617065
For word set 3:
Synonym similarity hate - loathe: 0.5873430371284485
Antonym similarity hate - love: 0.600395679473877
For word set 4:
Synonym similarity sad - unhappy: 0.41572242975234985
Antonym similarity sad - happy: 0.53546142578125
```

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

Space for answer

Answer:

Because some words have multiple meanings, and in word2vec it doesn't necessarily get fully captured when your computing similarity between them.

1.1.4 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 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]

[123]:
```
### YOUR EXAMPLES BELOW
# Atlanta:Georgia ::Seattle:Washington
c1,b1,a1,d1='Washington','Atlanta','Georgia','Seattle'
# son:father::daughter:mother
c2,b2,a2,d2='mother','son','father','daughter'

### YOUR EXAMPLES ABOVE

assert(word_vectors.most_similar(positive=[c1, b1], negative=[a1],␣
↪topn=1))[0][0] == d1, "example 1 invalid"
assert(word_vectors.most_similar(positive=[c2, b2], negative=[a2],␣
↪topn=1))[0][0] == d2, "example 2 invalid"

### YOUR EXAMPLES BELOW

# bark:dog::meow:cat
c3,b3,a3,d3 = 'cat','bark','dog','meow'
# gladiator:fight :: artist:paint
c4,b4,a4,d4 = 'paint','gladiator','fight','artist'
# ### 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? [2 pts]

Space for answer

Answer

A possible explanation for this is that the corpus used for to train the model didn't have enough example to form correct analogies from.

1.1.5 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.

[124]:
```
word_vectors.most_similar(positive=["woman", "doctor"], negative=["man"],␣
↪topn=10)
```

[124]:
```
[('gynecologist', 0.7093892097473145),
 ('nurse', 0.647728681564331),
 ('doctors', 0.6471461057662964),
 ('physician', 0.64389967918396),
 ('pediatrician', 0.6249487996101379),
 ('nurse_practitioner', 0.6218312978744507),
 ('obstetrician', 0.6072014570236206),
 ('ob_gyn', 0.5986712574958801),
 ('midwife', 0.5927063226699829),
 ('dermatologist', 0.5739566683769226)]
```

[137]:
```
word_vectors.most_similar(positive=["cop", "black"], negative=["man"], topn=10)
```

[137]:
```
[('white', 0.4960738718509674),
 ('Ebere_Collins', 0.44753748178482056),
 ('cops', 0.4281965494155884),
 ('colorism', 0.41902923583984375),
 ('grays_browns', 0.39897066354751587),
 ('nonblack', 0.3931658864021301),
 ('TVES_logo', 0.39209261536598206),
 ('narc', 0.3894709646701813),
 ('Reginald_Denny', 0.3882511258125305),
 ('crooked_cops', 0.38701021671295166)]
```

Provide two more examples that reveal some bias in the word embeddings. Look at the top-5 matches and justify your examples. [4 pts]

[139]:
```
### YOUR EXAMPLES BELOW
a1,b1,c1='women','muslim','man'
a2,b2,c2='man','black','cop'
### 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"
```

```
women:muslim::man:?
[('OSAMA_Bin_Laden', 0.4745621681213379), ('christian', 0.4616185426712036),
('idolater', 0.4609474539756775), ('feller', 0.45455074310302734), ('moslem',
```

```
0.4540647566318512)]

man:black::cop:?
[('white', 0.4960738718509674), ('Ebere_Collins', 0.44753748178482056), ('cops',
0.4281965494155884), ('colorism', 0.41902923583984375), ('grays_browns',
0.39897066354751587)]
```

Why do you think such bias exists? [1 pt]
Space for answer
Answer:
A possible explanation for these biases is due to outdated corpus. Ideas that are considered forbidden today was once not taboo when the corpus was created.

1.1.6 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.

[207]:
```python
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/
 ↪master/src/sotu/Obama_2009.txt",
                "https://raw.githubusercontent.com/nlp-compromise/nlp-corpus/
 ↪master/src/sotu/Obama_2010.txt",
                "https://raw.githubusercontent.com/nlp-compromise/nlp-corpus/
 ↪master/src/sotu/Obama_2011.txt",
                "https://raw.githubusercontent.com/nlp-compromise/nlp-corpus/
 ↪master/src/sotu/Obama_2012.txt",
                "https://raw.githubusercontent.com/nlp-compromise/nlp-corpus/
 ↪master/src/sotu/Obama_2013.txt",
```

```
                "https://raw.githubusercontent.com/nlp-compromise/nlp-corpus/
↪master/src/sotu/Obama_2014.txt",
                "https://raw.githubusercontent.com/nlp-compromise/nlp-corpus/
↪master/src/sotu/Obama_2015.txt"]

        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]

[ ]:

```
```

[208]:

```
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
    s= TSNE()
    tsne_transformed_vectors = s.fit_transform(X=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
```

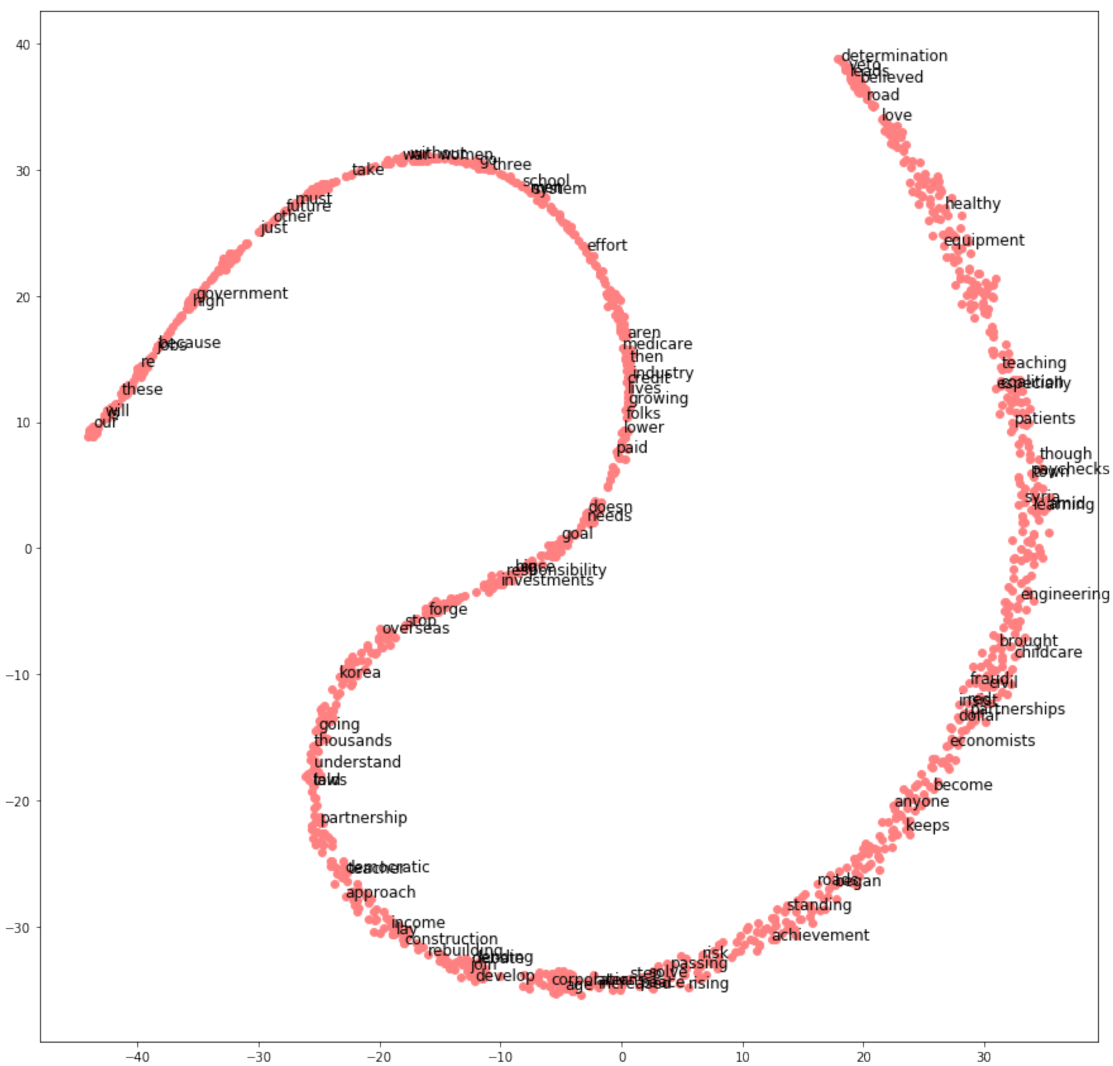
Let us plot the result and add labels for a few words on the plot. You can edit the `must_include` list to mandatorily include a few words you want to base your inferences on.

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]

[209]:
```python
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])
    for i in selected_indices:
        plt.annotate(words[i], (x_vals[i], y_vals[i]), fontsize=12)

must_include =␣
 ↪['war','peace','love','hate','teacher','women','men','music','xbox','school','dog','cats']
plot_with_matplotlib(tsne[:, 0], tsne[:, 1], list(model.wv.vocab.keys()),␣
 ↪must_include, random_include=100)
```



Space for answer

ANSWER:

'School' and 'System'(close together). They appear close to each other because 'school system' or education system are commonly used terms.

'Construction' and 'develop'(close together) are synonym .

'Construction' and 'rebuilding'(close together), A a viable contexual usage could be 'rebuilding a construction'. Builders/construction workers rebuilding a construction.

'dollar' and 'econmoist' (close together) . It's feasible to get see these words together directly or indirectly in a context , since the relationship here is that 'economist' care about money/'dollar'.

'teaching' and 'patients'(close together) . The contexual usage is 'teaching' takes 'patients', or teachers need to be patients with there students.

[ ]: