ECS 7001 - NN & NLP

Lab 3: Skip-gram Model for Word2Vec

February 4th

There are two Word2Vec architectures for creating word embeddings: the Continuous Bag of Words (CBOW) architecture and the Skip Gram architecture . In this lab, we will obtain our own word embeddings by training a skip-gram neural network model. Some of the code for this will be supplied here but in some sections, you will be required to implement the code yourself. Hints and tips will be provided.

The skip gram model is essentially a feedforward neural network with one hidden layer, trained to predict the context word given a target word. There are two ways to train this model: using hierarchical softmax function and/or by negative sampling. In this lab, we will be training using negative sampling. To train with negative sampling, the model is cast as a binary classification problem. The dataset would consist of positive and negative examples of the form:

Inputs	labels
(target_word, word_in_its_context)	1
(target_word, word_not_in_its_context)	0

created from the sentences in a corpus. As an example, consider the sentence: "***The quick brown fox jumped over the lazy dog***". For the target word '**fox**' and a window size of 2, training examples drawn from this sentence would be:

Inputs	Labels
(fox, quick)	**1**
(fox, brown)	**1**
(fox, the)	**0**
(fox, jumped)	**1**
(fox, lazy)	**0**
(fox, dog)	**0**
(fox, over)	**1**

Consequently, the model is trained to predict 1 when a word is in the context of the target word (i.e. in the window of the target word) and 0 otherwise. The model learns the statistics of the given corpus: the frequency with two words appear together would determine how similar they are (similarity is usually measured using cosine distance). After training, the word embeddings are gotten from the hidden layer weights.

0. Prepare the environment

Open Google Colab or activate the virtual environment you've created

1. Downloading the Corpus

Our training data will be comprised of 3 documents from the Gutenberg corpus. These documents can be loaded using nltk using the following instructions:

```
>>>  import nltk
>>> nltk.download('punkt')
>>> nltk.download('gutenberg')
>>> from nltk.corpus import gutenberg
>>> austen = gutenberg.sents('austen-sense.txt') + gutenberg.sents('austen-emma.txt') +
gutenberg.sents('austen-persuasion.txt')
```

Sanity check:
This training corpus contains 16498 sentences. Use the line that follows to ensure that your code has the same number of lines.

```
>> print(len(austen))
16498
```

2. Preprocessing the Training Corpus

In this section, you will write code to remove special characters, empty strings, digits and stopwords from the sentences and put all the words into lower cases.

Hints:

- The corpus is a list of lists. Each inner list contains all the words in the sentence at that position. Eg:
  `>> austen[0] = ['[', 'Sense', 'and', 'Sensibility', 'by', 'Jane', 'Austen', '1811', ']']`
- the python <string> library contains a variable “punctuation” that is a string containing all the special characters.
- You might need to write a function that takes the corpus as an argument and returns the preprocessed corpus of the same data type.

Tip: You can also return a list of strings (each sentence a string) and make use of keras text preprocessing library to create the vocabulary and training data in the next section.

You can also consider using keras text processing library to preprocess the text
https://keras.io/preprocessing/text/

Sanity check:
You can test the function with the following lines to see if printed outputs are similar to the ones below:
>>> print('Length of processed corpus:', len(normalized_corpus))
```
Length of processed corpus: 13927
```

The exact figure depends on how you preprocessed your corpus.

>>> print('Processed line:', normalized_corpus[10])
```
Processed line: therefore succession norland estate really important
sisters fortune independent might arise father inheriting property could
small
```
The above command returns a sentence, either as a string (as above) or as a list of strings ['therefore', 'succession', 'norland', 'estate', 'really', 'important', 'sisters', 'fortune', 'independent', 'might' ….] depending on how you preprocessed your corpus

3. Creating the Corpus Vocabulary and Preparing the Dataset.

Firstly, you will write code to create 3 variables:

a. <word2idx>: a lookup table (dictionary) of all the unique words and indices assigned to them (count from 1. It is good practice in Deep Learning and NLP to save the 0 index for **padding** as you will see in the later labs).
b. <idx2word>: a lookup table of words indexed by their unique indices.
c. <sents_as_ids>: The input to the model cannot be text, rather, each sentence needs now be a list of indices. As such, <word_ids> is a list of lists, each inner list a list of indices of the words in that sentence in order.

Hint:
Keras has a text processing library that can be used to create the <word2idx> variable.
>> from keras.preprocessing import text

Use help(text) to find out more about this model or go to: https://keras.io/preprocessing/text/ . It can also be used to preprocess the text.

After you have created these variables, set the <vocab_size> and <embed_size> variables with the following commands:
>>> vocab_size = len(word_ids) + 1 # 1 was added for zero padding
>>> embed_size = 100   # We are creating 100D embeddings.

Sanity Check:
Run the following lines of code:
>>> print('Number of unique words:', len(word_ids))
```
Number of unique words: 10098
```
This number might be different depending on how you how you # preprocessed your corpus

>>> print('\nSample word2idx: ', list(word2idx.items())[:10])
```
Sample word2idx:  [('talent', 2431), ('appealed', 4247), ('resist', 1602),
('gravity', 2828), ('correspond', 4457), ('indoors', 7669), ('kissed',
3430), ('going', 113), ('illegitimacy', 5880), ('sickness', 2947)]
```
The items are randomly ordered but the command should give you (word, index pairs)

>>> print('\nSample idx2word:', list(idx2word.items())[:10])
```
Sample idx2word: [(1, 'could'), (2, 'would'), (3, 'mr'), (4, 'mrs'), (5,
'must'), (6, 'said'), (7, 'one'), (8, 'much'), (9, 'miss'), (10, 'every')]
```

>>> print('\nSample normalized corpus:', normalized_corpus[:3])
```
Sample normalized corpus: ['sense sensibility jane austen', 'family
dashwood long settled sussex', 'estate large residence norland park centre
property many generations lived respectable manner engage general good
opinion surrounding acquaintance']
```
This would return a list of 3 sentences. depending on how you preprocessed your output, each sentence would either be a string (as above) or a tokenized sentence' like so:
```
[['sense', 'sensibility', 'jane', 'austen'], ['family', 'dashwood',
'long', settled', 'sussex'], ['estate', 'large', 'residence', 'norland',
'park', 'centre', 'property', 'many', 'generations', 'lived',
'respectable', 'manner', 'engage', 'general', 'good', 'opinion',
'surrounding', acquaintance']]
```

>>> print('\nAbove sentence as a list of ids:' , sents_as_ids[:3])
```
Above sentence as a list of ids: [[305, 1379, 75, 4299], [108, 101, 57,
333, 2588], [1022, 405, 1627, 597, 554, 2784, 1023, 66, 4300, 512, 768,
160, 1164, 199, 15, 190, 3044, 147]]
```
This output might differ depending on how you preprocessed your corpus but should return 3 lists, each one a list of the indices corresponding to the words in each sentence in <normalized_corpus[:3]>

4. Generating training instances

In this section we would generate the training examples of the format shown in the opening statement using the keras skip-gram generator https://keras.io/preprocessing/sequence/

>>> from keras.preprocessing.sequence import skipgrams
>>> skip_grams = [skipgrams(sent, vocabulary_size=vocab_size, window_size=5) for sent in sents_as_ids]

Sanity Check:

To view the skip_grams for the first sentence in the training data, run the lines of code below.

```
# view sample skip-grams
>>> pairs, labels = skip_grams[0][0], skip_grams[0][1]
>>> for i in range(len(pairs)):
        print('({:s} ({:d}), {:s} ({:d})) -> {:d}'.format(
        # the first word and its index
        idx2word[pairs[i][0]], pairs[i][0],
        # the second word and its index
        idx2word[pairs[i][1]], pairs[i][1],
        # the label
        labels[i]))
```

```
(sense (305), par, (8748)) -> 0
(sensibility (1379), name, (229)) -> 0
(austen (4299), perpetuated, (9117)) -> 0
(jane (75), classing, (8582)) -> 0
(sense (305), baldwin, (4283)) -> 0
(sensibility (1379), sense, (305)) -> 1
(austen (4299), jane, (75)) -> 1
(austen (4299), sense, (305)) -> 1
(sense (305), austen, (4299)) -> 1
(sensibility (1379), uniting, (3978)) -> 0
(austen (4299), work, (679)) -> 0
(austen (4299), porker, (8131)) -> 0
(jane (75), sense, (305)) -> 1
(sense (305), call, (353)) -> 0
(sensibility (1379), austen, (4299)) -> 1
(sensibility (1379), jane, (75)) -> 1
(austen (4299), sensibility, (1379)) -> 1
(sense (305), sensibility, (1379)) -> 1
(jane (75), sanguinely, (8560)) -> 0
(sensibility (1379), skilful, (9156)) -> 0
(sense (305), jane, (75)) -> 1
(jane (75), sensibility, (1379)) -> 1
```

```
(jane (75), austen, (4299)) -> 1
(jane (75), infants, (7169)) -> 0
```

5. Building the Skip-gram Neural Network Architecture

In this section we would be building the skip-gram neural network architecture using the Keras Functional API and the Sequential model introduced in the previous lab. https://keras.io/getting-started/functional-api-guide/

```
>>> from keras.layers import Dot, Input
>>> from keras.layers.core import Dense, Reshape
>>> from keras.layers.embeddings import Embedding
>>> from keras.models import Model
>>> from keras.utils import plot_model
```

The skip-gram model is two input one output feedforward neural network with one hidden layer and this will be built over a series of steps.

A. The first step is to initialize and transform the first input using the following lines of code:

```
# The input is an array of target indices e.g. [2, 45, 7, 23,...9]
>>> target_word = Input((1,), dtype='int32')

 # feed the words into the model using the Keras <Embedding> layer. This is the hidden layer from whose weights we will get the word embeddings.
>>> target_embedding = Embedding(vocab_size, embed_size, name='target_embed_layer',
                                 embeddings_initializer='glorot_uniform',
                                 input_length=1)(target_word)

# at this point, the input would of the shape (num_inputs x 1 x embed_size) and has to be flattened or reshaped into a (num_inputs x embed_size) tensor.
>>> target_input = Reshape((embed_size, ))(target_embedding)
```

B. Write similar code for the 'context_word' input.

C. Merge the inputs.

Recall, each training instance is a (target_word, context_word) combination. Since we are trying to learn the degree of closeness between the two words, the model will compute the cosine distance between the two inputs using the <Dot> layer. https://keras.io/layers/merge/, hence fusing the two inputs into one.

```
>>> merged_inputs = Dot(axes=-1, normalize=False)([target_input, context_input])
```

D. Pass the merged inputs into sigmoid activated layer

Pass the merged inputs (now a vector with a single number the cosine distance between the two input vectors for each word) into a sigmoid activated neuron. The output of this layer is the output of the model.

Hint: Use the <Dense> layer (https://keras.io/layers/core/), with the 'glorot_uniform' kernel initialization and a 'sigmoid' activation function.

E. Initialize the model:

```
>>> model = Model(inputs=[target_word, context_word], outputs=[label])  # label is the output of step D.
```

F. Compile the model using the <model.compile> command. Use Loss = 'mean_squared_error', optimizer = 'rmsprop'.

Sanity check:

You can visualize the model and the model summary by running the following lines of code.

view the model summary

```
>>> model.summary()
```

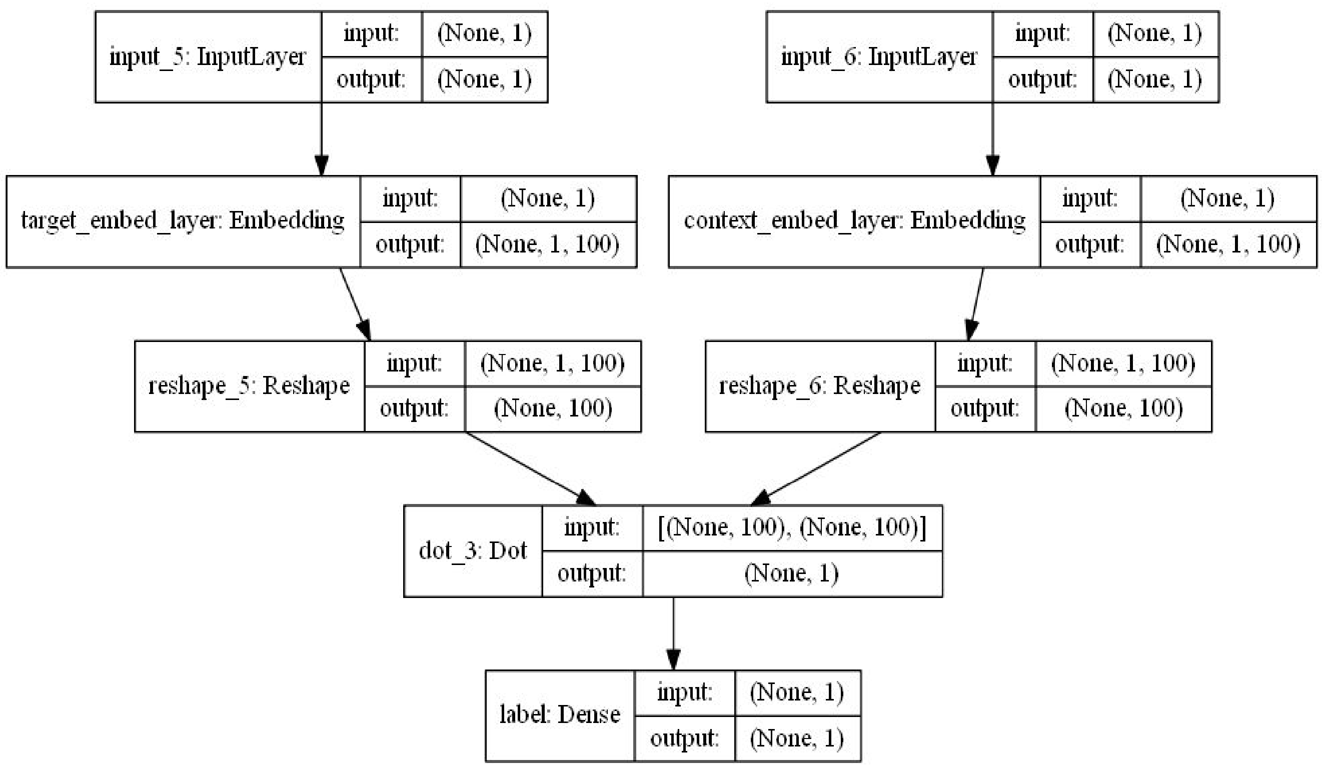
```
____________________________________________________________________________
_________________________
Layer (type)                     Output Shape         Param #     Connected
to
============================================================================
=========================
input_5 (InputLayer)             (None, 1)            0
____________________________________________________________________________
_________________________
input_6 (InputLayer)             (None, 1)            0
____________________________________________________________________________
_________________________
target_embed_layer (Embedding)  (None, 1, 100)       1009900
input_5[0][0]
____________________________________________________________________________
_________________________
context_embed_layer (Embedding) (None, 1, 100)       1009900
input_6[0][0]
____________________________________________________________________________
_________________________
reshape_5 (Reshape)             (None, 100)          0
target_embed_layer[0][0]
____________________________________________________________________________
_________________________
reshape_6 (Reshape)             (None, 100)          0
context_embed_layer[0][0]
____________________________________________________________________________
_________________________
dot_3 (Dot)                     (None, 1)            0
reshape_5[0][0]

reshape_6[0][0]
____________________________________________________________________________
_________________________
label (Dense)                   (None, 1)            2
dot_3[0][0]
============================================================================
=========================
Total params: 2,019,802
```

```
Trainable params: 2,019,802
Non-trainable params: 0
```

You can also visualize the model architecture.
You can use <plot_model>. First, you have to install graphviz https://graphviz.readthedocs.io/en/stable/manual.html and pydot https://pypi.org/project/pydot/ . Then plot it to file using the following lines of code:
>>> from keras.utils import plot_model
>>> plot_model(model, to_file='skipgram_keras', show_shapes=True, show_layer_names=True, rankdir='TB')

Alternatively, you can visualize it using vis_utils using the following lines of code:
>>> from IPython.display import SVG
>>> from keras.utils import vis_utils
>>> SVG(vis_utils.model_to_dot(model, show_shapes=True,
show_layer_names=True).create(prog='dot', format='svg'))



6. Training the Model

Run the following lines of code to train the model for 5 epochs:

```
>>> for epoch in range(1, 6):
        epoch_loss = 0
        # in each epoch, train all the sentences, one per iteration=> batch_size = num_sents
        for i, sent_examples in enumerate(skip_grams):
            target_wds = np.array([pair[0] for pair in sent_examples[0]], dtype='int32')
            context_wds = np.array([pair[1] for pair in sent_examples[0]], dtype='int32')
            labels = np.array(sent_examples[1], dtype='int32')
            X = [target_wds, context_wds]
            Y = labels
            if i % 5000 == 0: # after 5000 sentences i.e. 5000 iterations
                print('Processed %d sentences' %i)
            epoch_loss += model.train_on_batch(X, Y)
        print('Processed all %d sentences' %i)
        print('Epoch:', epoch, 'Loss:', epoch_loss, '\n')
```

The training takes about 10 minutes to run and the print statements should appear in the following format:

The actual value of the loss doesn't have to be exactly as above.

While waiting for the training to complete, you can read this article on the softmax skip-gram implementation. http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

- What would the inputs and outputs to the model be?
- How would you use the Keras framework to create this architecture?
- Can you think of reasons why this model is considered to be inefficient?

7. Getting the Word Embeddings

The word embeddings are the weights of the target word embedding layer.

```
>>> word_embeddings = model.get_layer('target_embed_layer').get_weights()[0][1:] # Recall that 0 was left for padding
```

Sanity Check:

```
>>> print(word_embeddings.shape)
(10098, 100)
```

```
>>> from pandas import DataFrame
```

```
>>> print(DataFrame(weights, index=idx2word.values()).head(10))
```

```
              0         1         2         3         4         5         6  \
could -0.053845  0.078008  0.192627 -0.109420  0.081059 -0.032775 -0.093648
would  0.067021 -0.293036 -0.043924 -0.022885  0.160918 -0.058537  0.179228
mr    -0.029986 -0.027346  0.112609 -0.037861 -0.199847 -0.015855  0.082381
mrs    0.001823  0.160674  0.165437  0.182159  0.055967 -0.085066  0.053510
must   0.137356 -0.102320  0.085263 -0.066205 -0.047705 -0.040095  0.226901
said  -0.027512 -0.069858  0.127058  0.082656  0.053633 -0.121620 -0.164046
one    0.033047 -0.113147  0.105907 -0.068891  0.131159 -0.084675 -0.150654
much  -0.054012  0.106995 -0.044088  0.029724  0.068506 -0.089523  0.048099
miss  -0.066710 -0.227043  0.246139  0.050581  0.077042 -0.063427 -0.027480
every -0.038398  0.096737 -0.056688 -0.027508 -0.119196  0.002885  0.217931

              7         8         9    ...           90        91        92  \
could  0.120536 -0.075619  0.033622    ...     0.020059 -0.115686  0.156078
would  0.043619 -0.137241  0.062000    ...    -0.053763 -0.250681  0.294423
mr    -0.068823 -0.159722  0.144301    ...     0.230267 -0.113654  0.344071
mrs   -0.069978 -0.204509  0.150778    ...     0.264131  0.063224  0.440879
must  -0.119852  0.107529 -0.111287    ...    -0.011106  0.031511  0.305808
said  -0.101632  0.113226 -0.035657    ...    -0.056466 -0.042309  0.243528
one    0.023622  0.075587  0.123809    ...     0.001648  0.062925  0.120780
much  -0.012249  0.114400  0.091373    ...     0.012350  0.068399  0.094535
miss  -0.100172  0.113781  0.072466    ...    -0.023654 -0.164943  0.155953
every  0.160736 -0.158576 -0.194456    ...    -0.096196  0.101962 -0.042990

             93        94        95        96        97        98        99
could  0.041663 -0.177343 -0.125847 -0.307792 -0.006311  0.283965  0.055550
would -0.008801  0.099036  0.005128 -0.409054 -0.042482 -0.049890 -0.054074
mr    -0.152716  0.079066  0.059824 -0.437984 -0.079991  0.086770  0.100775
mrs   -0.097926 -0.099051  0.072806 -0.345828 -0.108707 -0.024807  0.282743
must   0.087862  0.014571  0.027280 -0.225805  0.065963 -0.065508 -0.105692
said   0.012433  0.065090  0.099017 -0.307569 -0.098328  0.066315 -0.070196
one   -0.114458 -0.045783 -0.023736 -0.123718 -0.022160 -0.057780  0.256288
much  -0.133036  0.003154 -0.225230 -0.310352 -0.083591 -0.008327 -0.055348
miss  -0.032347 -0.191296  0.119380 -0.296855 -0.016562  0.155227 -0.123548
every -0.101362 -0.084066  0.012614 -0.047140 -0.001202 -0.097049  0.354726

[10 rows x 100 columns]
```

Your output may not be exactly as above but the command should print 10 words and their respective word vectors.

8. Measuring Similarity Between Word Pairs

```
>>> from sklearn.metrics.pairwise import cosine_similarity
>>> similarity_matrix = cosine_similarity(word_embeddings)
```

Check:

```
>>> print(similarity_matrix.shape)
(10098, 10098)
```

The similarity matrix gives the similarity between each pairs of words in the vocabulary. Given a pair of words eg. ('wisdom', 'folly') how can you obtain their cosine similarity?

9. Exploring and Visualizing your Word Embeddings using t-SNE

A. **Get the most similar words to the search items in the list below**

```
>>> search_terms = ['man', 'love', 'hatred', 'woman', 'wisdom',  'kindness', 'god', 'man', 'folly', 'fool']

# Get the 5 words most similar to the search terms
 >>>similar_words = {term: [idx2word[idx]
                        for idx in (-1 * similarity_matrix[word2idx[term]-1]).argsort()[1:6] + 1]
                        for term in search_items}
```

Sanity check:

```
>>> print(similar_words)
{'love': ['follow', 'cod', 'possesses', 'inferiority', 'enter'],
'man': ['matters', 'parishes', 'liking', 'know', 'motionless'],
'god': ['row', 'studiously', 'housemaid', 'preceded', 'lowered'],
'kindness': ['urgency', 'law', 'moves', 'gloom', 'report'],
'wisdom': ['misconduct', 'wisely', 'provide', 'eaten', 'attribute'],
'hatred': ['ecstatic', 'uppermost', 'posts', 'stealth', 'afflicting'],
'fool': ['interposed', 'aghast', 'crisis', 'owning', 'coast'],
'woman': ['children', 'terrors', 'deserving', 'besides', 'imaginist'],
'folly': ['strangers', 'elizabeths', 'mute', 'nervously', 'risen']}
```

The similar words obtained would depend on your training but the above command should print a dictionary. Each key is a search term and each value is a list of the 5 words the model predicts to be most similar to the key word.

B. **Plot the words in the dictionary above using t-SNE** https://lvdmaaten.github.io/tsne/

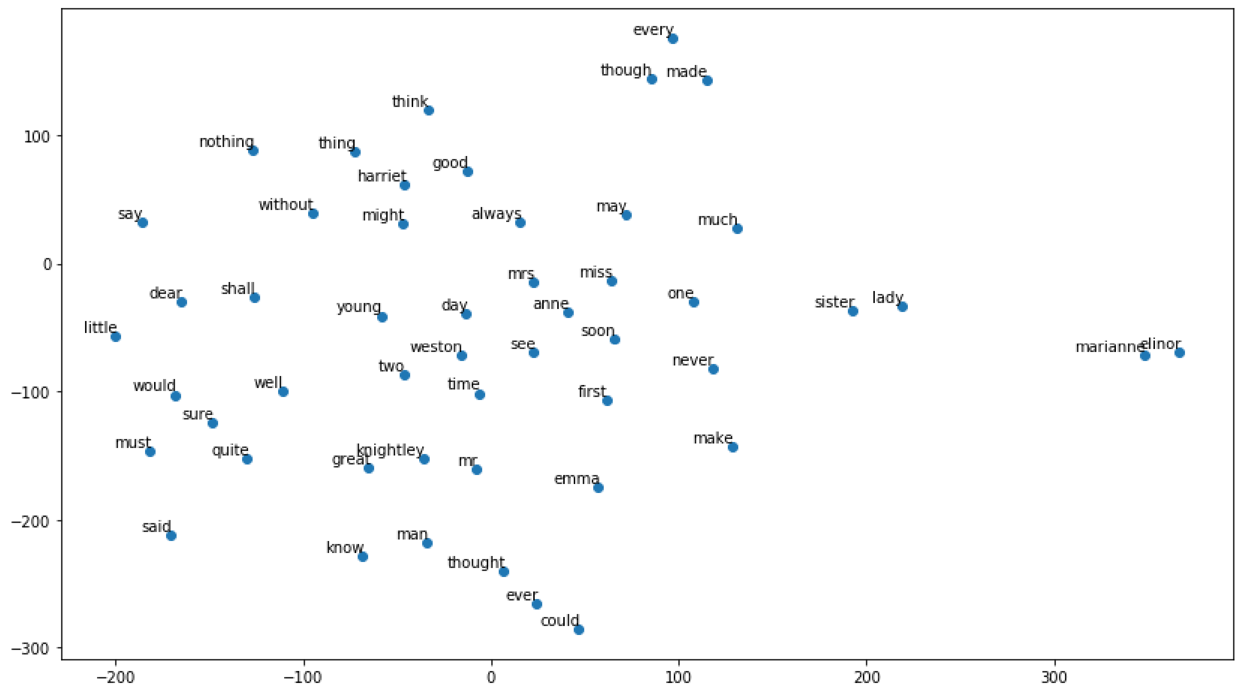
Plot 500 of the word embeddings using the code snippets below:

```
>>> from sklearn.manifold import TSNE
>>> import matplotlib.pyplot as plt

>>> tsne = TSNE(perplexity=3, n_components=2, init='pca', n_iter=5000, method='exact')
>>> np.set_printoptions(suppress=True)
>>> plot_only = 50
>>> T = tsne.fit_transform(word_embeddings[:plot_only, :])
>>> labels = [idx2word[i+1] for i in range(plot_only)]
>>> plt.figure(figsize=(14, 8))
>>> plt.scatter(T[:, 0], T[:, 1])
>>> for label, x, y in zip(labels, T[:, 0], T[:, 1]):
        plt.annotate(label, xy=(x+1, y+1), xytext=(0, 0), textcoords='offset points', ha='right',
                     va='bottom')
```

Sanity Check:

If you've implemented it well, it should look somewhat like this:



10. Resources used

https://adventuresinmachinelearning.com/word2vec-tutorial-tensorflow/
https://towardsdatascience.com/understanding-feature-engineering-part-4-deep-learning-methods-for-text-data-96c44370bbfa
https://adventuresinmachinelearning.com/word2vec-keras-tutorial/
https://www.tensorflow.org/tutorials/representation/word2vec#the_skip-gram_model
https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/tutorials/word2vec/word2vec_basic.py