ECS 7001 - NN & NLP

Assignment 1: Word Representation and Text Classification with Neural Networks

Spyridon Roumpis

181004877

Part A: Word Embeddings with Word2Vec

1. Preprocessing the training corpus:

To preprocess the Training Corpus a function was created (def preprocessing) to remove punctuation(special characters, empty strings, digits), stopwords and put all the words into lower cases. Also because of the window_size=5 we excluded any sentence's length which was less than 3.

Running the Sanity Check, the length of the normalized_corpus is 13927 and the normalized_corpus[10] = ['therefore', 'succession', 'norland', 'estate', 'really', 'important', 'sisters', 'fortune', 'independent', 'might', 'arise', 'father', 'inheriting', 'property', 'could', 'small'].

The below screenshot also shows these results.

```
print('Length of processed corpus:', len(normalized_corpus))
print('Processed line:', normalized_corpus[10])
```

```
Length of processed corpus: 13927
Processed line: ['therefore', 'succession', 'norland', 'estate', 'really', 'important', 'sisters', 'fortune', 'independent', 'might', 'arise', 'father', 'inheriting', 'property', 'could', 'small']
```

2. Creating the corpus vocabulary and preparing the dataset:

For the purpose of this task we had to create three variables, a lookup table(dictionary) of all the unique words and indices assigned to them (count from 1), a lookup table of words indexed by their unique indices and a list of indices.

```
[ ] print('Number of unique words:', len(idx2word))
```

```
Number of unique words: 10098
```

```
[ ] print('Vocabulary Sample:', list(idx2word.items())[:10])
```

```
Vocabulary Sample: [(1, 'could'), (2, 'would'), (3, 'mr'), (4, 'mrs'), (5, 'must'), (6, 'said'), (7, 'one'), (8, 'much'), (9, 'miss'), (10, 'every')]
```

```
[ ] print('\nSample word2idx: ', list(word2idx.items())[:10])
```

```
Sample word2idx:  [('could', 1), ('would', 2), ('mr', 3), ('mrs', 4), ('must', 5), ('said', 6), ('one', 7), ('much', 8), ('miss', 9), ('every', 10)]
```

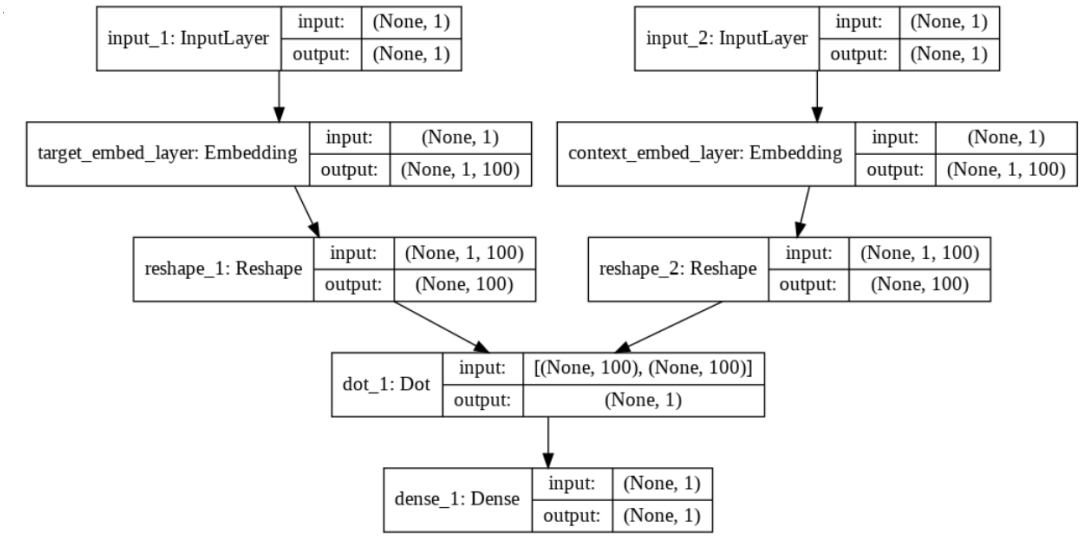
```
[ ] print('\nAbove sentence as a list of ids:' , sents_as_ids[:3])
```

```
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,
```

3. Building the skip-gram neural network architecture

```
[ ] model.summary()
```

```
Model: "model_2"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_3 (InputLayer)            (None, 1)            0
__________________________________________________________________________________________________
input_4 (InputLayer)            (None, 1)            0
__________________________________________________________________________________________________
target_embed_layer (Embedding)  (None, 1, 100)       1009900     input_3[0][0]
__________________________________________________________________________________________________
context_embed_layer (Embedding) (None, 1, 100)       1009900     input_4[0][0]
__________________________________________________________________________________________________
reshape_3 (Reshape)             (None, 100)          0           target_embed_layer[0][0]
__________________________________________________________________________________________________
reshape_4 (Reshape)             (None, 100)          0           context_embed_layer[0][0]
__________________________________________________________________________________________________
dot_2 (Dot)                     (None, 1)            0           reshape_3[0][0]
                                                                 reshape_4[0][0]
__________________________________________________________________________________________________
dense_2 (Dense)                 (None, 1)            2           dot_2[0][0]
==================================================================================================
Total params: 2,019,802
Trainable params: 2,019,802
Non-trainable params: 0
__________________________________________________________________________________________________
```



4. Training the models (and reading McCormick's tutorial)

1. What would the inputs and outputs to the model be?

A one-hot vector would be used to represent the input of the model and the dimension of it would be the vocabulary length. A '1' would be placed in the word selected to be analysed while '0' to the rest of the words. The network will give as an output the probability of every word in the vocabulary being the 'nearby word' to the one we selected.
The network would be trained on word pairs and is going to learn the statistics from the number of times each pairing shows up.

2. How would you use the Keras framework to create this architecture?

The skip gram model is essentially a feedforward neural network with one hidden layer, trained to predict the context word given a target word. For the skip-gram neural network architecture a sequential model would be introduced in association with the keras framework. Firstly, an embedding layer would be used to feed the words into the model, this would be the hidden layer from whose weights we will get the word embeddings. Then to merge the inputs the Dot layer will be added. Finally, a sigmoid activated layer will be added which would have as an output the output of the model.

3. Can you think of reasons why this model is considered to be inefficient?

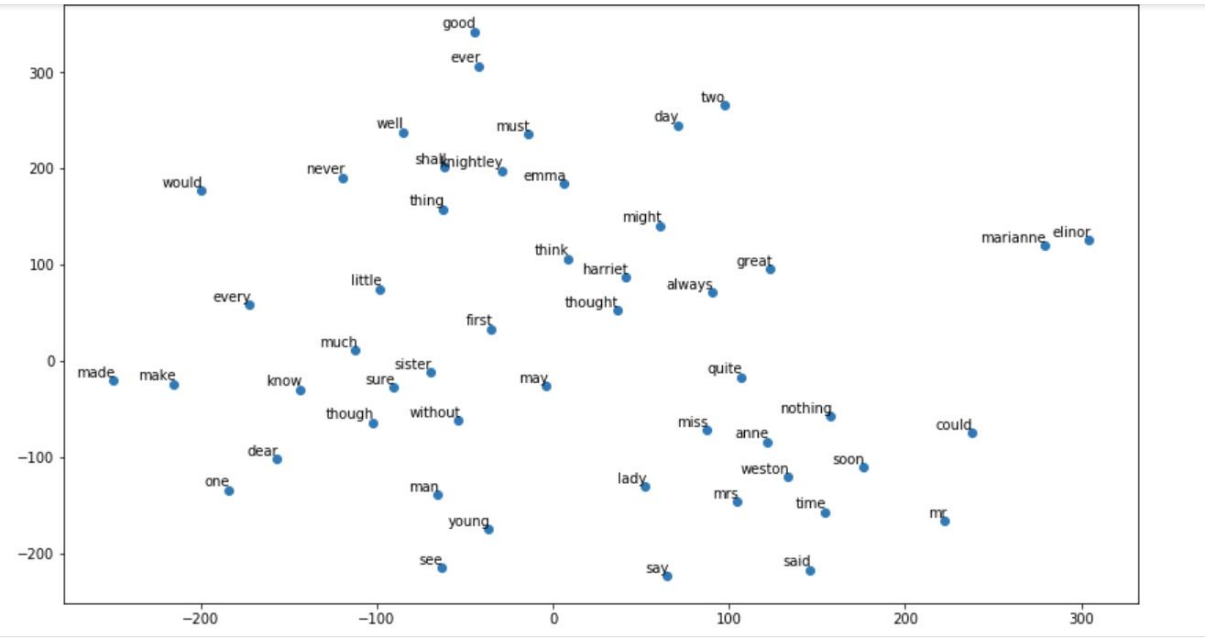
The main reason to be considered is that the skip-gram model for Word2Vec is a huge neural network and running gradient descent on a neural network that large is going to be slow. Also, a huge amount of training data would be needed in order to tune that many weights and avoid over-fitting. Although, subsampling frequent words and applying Negative Sampling would not only reduce the compute burden of the training process, but also would improve the quality of the resulting word vectors as well, that's why we use negative sampling in our example.

5. Getting the word embeddings

```
[88] word_embeddings = model.get_layer('target_embed_layer').get_weights()[0][1:] # Recall that 0 was left for padding
     print(word_embeddings.shape)
     from pandas import DataFrame
     print(DataFrame(word_embeddings, index=idx2word.values()).head(10))
```

```
(10098, 100)
             0         1         2   ...        97        98        99
could  0.044822  0.120959 -0.068937  ... -0.045648 -0.123783 -0.262600
would  0.224772 -0.070448 -0.051844  ... -0.060307 -0.091860 -0.046358
mr    -0.031517  0.142785  0.027659  ... -0.048642 -0.166579 -0.104604
mrs    0.096082  0.030190  0.214388  ... -0.121067 -0.244435 -0.098161
must   0.174753 -0.012755  0.077266  ...  0.072584 -0.094698  0.060863
said  -0.013192  0.088080  0.214581  ... -0.064781 -0.177161 -0.008881
one    0.049435 -0.139781  0.027098  ... -0.109997 -0.190295 -0.154851
much  -0.223187 -0.097670 -0.156537  ... -0.063591 -0.064143 -0.015831
miss   0.071210  0.039390 -0.013914  ...  0.000021 -0.113068 -0.196881
every -0.091163  0.064578  0.035996  ...  0.060528  0.173599 -0.039000

[10 rows x 100 columns]
```

6. Exploring and visualizing your word embeddings using t-SNE



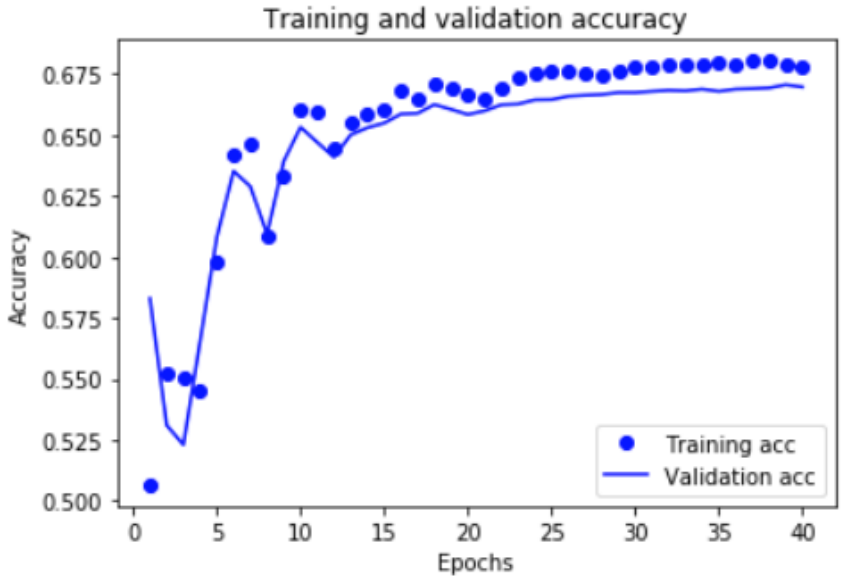
Part B: Basic Text Classification

1.Build a neural network classifier using one-hot word vectors, and train and evaluate it:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lambda_1 (Lambda)            (None, 256, 10000)        0
_________________________________________________________________
global_average_pooling1d_mas (None, 10000)             0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 10001
=================================================================
Total params: 10,001
Trainable params: 10,001
Non-trainable params: 0
_________________________________________________________________
```

```
[20] print(results)
     # loss, accuracay
```

```
[0.6785142780303955, 0.67368]
```



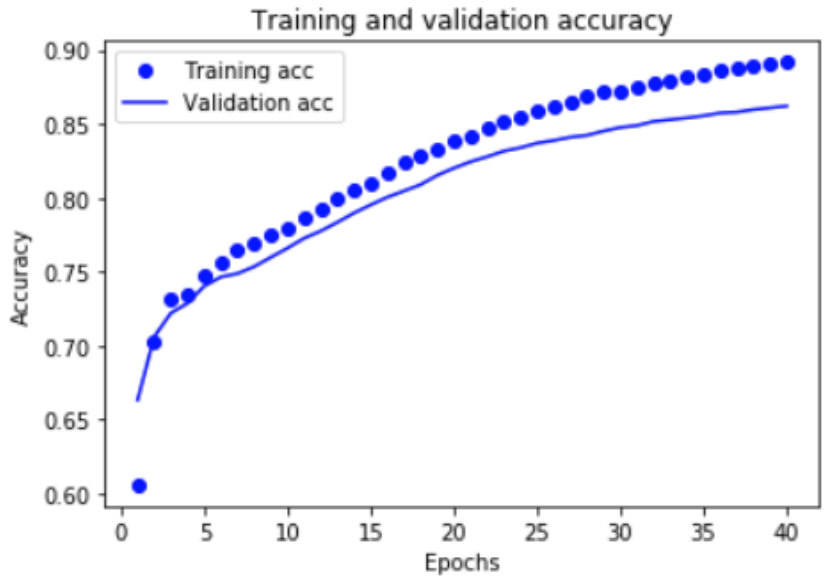
2. Adapt your model to use the pre-trained word embeddings you built in part A, and train and evaluate it:

```
model2.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 256, 16)           160000
_________________________________________________________________
global_average_pooling1d_mas (None, 16)                0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 17
=================================================================
Total params: 160,017
Trainable params: 160,017
Non-trainable params: 0
_________________________________________________________________
```

```
[24] print (results)
```

```
[0.38632840531349183, 0.85508]
```



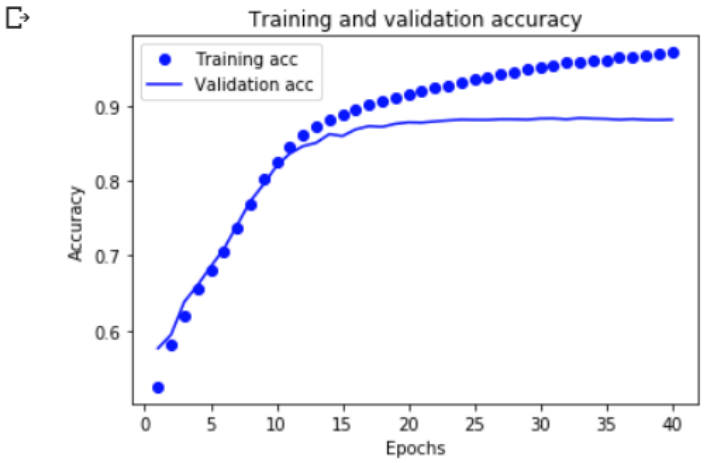
3. Now modify your training procedure so that the embeddings are learned along with the model itself

```
[35] model3.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, None, 50)          20000050
_________________________________________________________________
global_average_pooling1d_mas (None, 50)                0
_________________________________________________________________
dense_3 (Dense)              (None, 16)                816
_________________________________________________________________
dense_4 (Dense)              (None, 1)                 17
=================================================================
Total params: 20,000,883
Trainable params: 20,000,883
Non-trainable params: 0
_________________________________________________________________
```

```
[36] print(results)
```

```
[0.3307963452911377, 0.87044]
```



4. One way to improve the performance is to add another fully-connected layer to your network. Try this, and explain why the performance does not improve. What can you do to improve the situation?

In general, Choosing the right number of layers can only be achievable with practice.

Adding more layers can help you to extract more features. But we can do that up to a certain extent. After some point, instead of extracting features, we tend to overfit the data.

One possible reason that the performance did not improve is that by adding more layers, more trainable parameters would be added to the model, which means more training.

Another reason can be the sigmoid activation function, using relu instead might improve the performance.

Part C: Using LSTMs for Text Classification

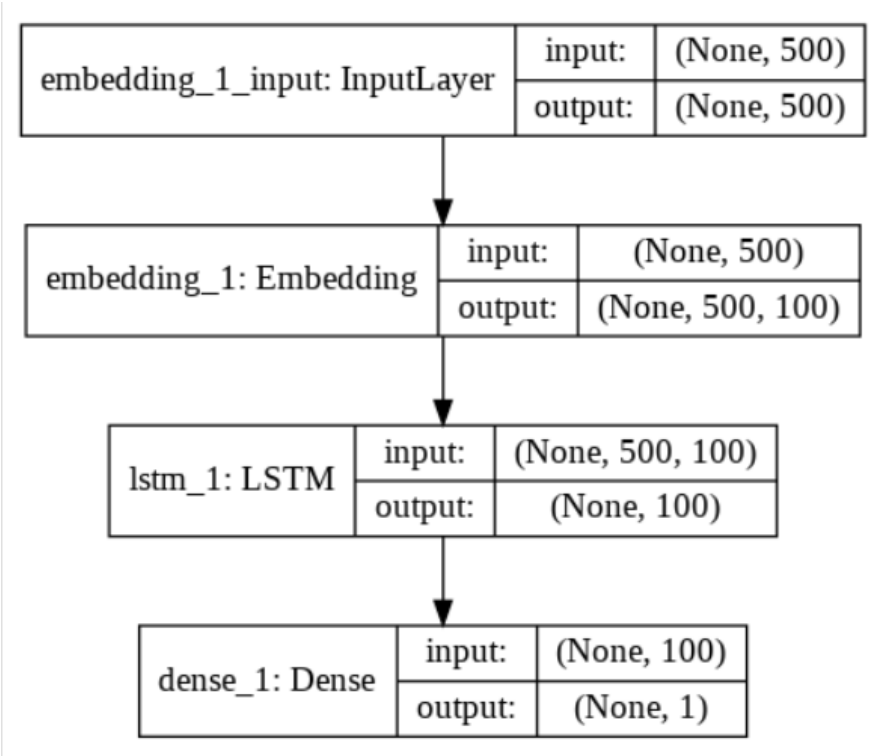
1. Section 2, Readying the inputs for the LSTM:

```
print('Length of sample train_data before preprocessing:', len(train_data[0]))
print('Length of sample train_data after preprocessing:', len(preprocessed_train_data[0]))
```

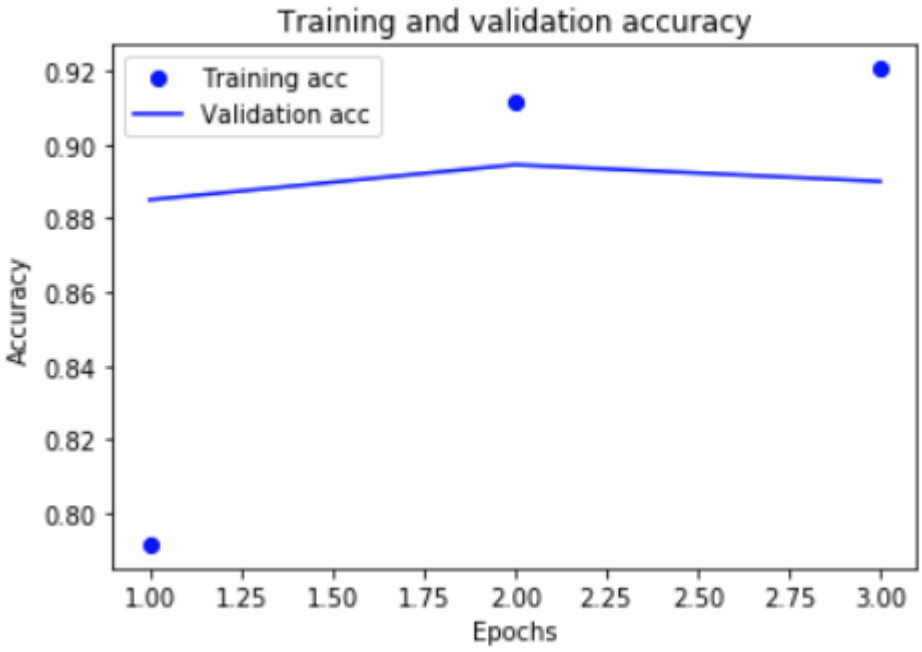
```
Length of sample train_data before preprocessing: 218
Length of sample train_data after preprocessing: 500
```

2. Building the model:

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 500, 100)          1000000
_________________________________________________________________
lstm_1 (LSTM)                (None, 100)               80400
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 101
=================================================================
Total params: 1,080,501
Trainable params: 1,080,501
Non-trainable params: 0
_________________________________________________________________
None
```



3. Section 4, training the model



It can be seen from the above image which shows the plot of training and validation accuracy through the epochs, that the optimal stopping point for the model is for 2 epochs with a 0.87 accuracy. So that means that our model could just have 2 epochs instead of 3.

4. Evaluating the model on the test data (section 5)

```
print('test_loss:', results[0], 'test_accuracy:', results[1])
```

```
test_loss: 0.3374149591064453 test_accuracy: 0.8724
```

5. Section 6, extracting the word

```
[ ]  print(model2.summary())
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 500, 100)          1000000
_________________________________________________________________
dropout_1 (Dropout)          (None, 500, 100)          0
_________________________________________________________________
lstm_2 (LSTM)                (None, 100)               80400
_________________________________________________________________
dropout_2 (Dropout)          (None, 100)               0
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 101
=================================================================
Total params: 1,080,501
Trainable params: 1,080,501
Non-trainable params: 0
_________________________________________________________________
None
```

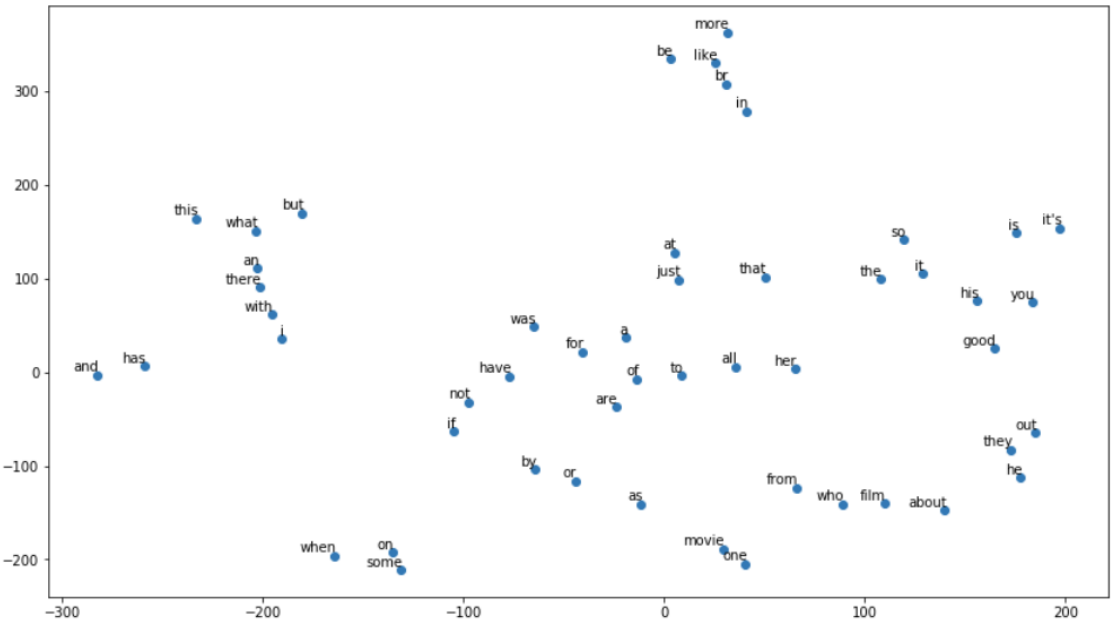
6. Visualizing the reviews

```
[ ] idx2word = {v:k for k, v in word2idx.items()}

[ ] print(' '.join(idx2word[idx] for idx in train_data[0]))

<START> this film was just brilliant casting location scenery story direction everyone's really suited the part they played and you could just imagine being there robert <UNK> is an amazing ac

[ ] print(train_data[0])

[1, 13, 21, 15, 42, 529, 972, 1621, 1384, 64, 457, 4467, 65, 3940, 3, 172, 35, 255, 4, 24, 99, 42, 837, 111, 49, 669, 2, 8, 34, 479, 283, 4, 149, 3, 171, 111, 166, 2, 335, 384, 38, 3, 171, 453
```

7. Visualizing the word embeddings

```
from pandas import DataFrame
print(DataFrame(word_embeddings, index=idx2word.values()).head(10))
```

```
                0         1         2   ...        97        98        99
woods     -0.056084 -0.009316  0.036493  ...  0.038313  0.027441  0.031448
hanging   -0.008935  0.034421  0.064842  ... -0.020322  0.057453  0.020192
woody     -0.051170 -0.043947 -0.029328  ...  0.034730  0.012947  0.035052
arranged  -0.047272  0.018267 -0.008508  ... -0.014974 -0.031867  0.044953
bringing  -0.002595  0.044875  0.007269  ...  0.026099 -0.045427 -0.030224
wooden     0.017759  0.014488 -0.002568  ...  0.048126 -0.011540 -0.053035
errors     0.038332 -0.030846 -0.018364  ... -0.041110  0.025619  0.013343
dialogs    0.007093 -0.044092 -0.023308  ...  0.017899  0.006517  0.003781
kids      -0.031411  0.007128  0.045554  ... -0.026650 -0.017859 -0.021363
uplifting -0.006416  0.032678 -0.015333  ...  0.015880 -0.022679  0.020787

[10 rows x 100 columns]
```



8. **Section 9 [4 marks]. For this paper, you have to write down your answers to the questions. 2 points each for questions 1 and 2.**

4. **Create a new model that is a copy of the model step 3. To this new model, add two dropout layers, one between the embedding layer and the LSTM layer and another between the LSTM layer and the output layer. Repeat steps 4 and 5 for this model. What do you observe? How about if you train this new model for 6 epochs instead?**

Adding the new layers did not improve the performance, instead it dropped by a small amount. Also training the model for 6 epochs does not have any possitive impact, and that is something that also can be seen from the graph in the previous section.

5. **Experiment with compiling the model with batch sizes of 1, 32, len(training_data). What do you observe?**

Experimenting with the batch size, does not seem to have any influence on the accuracy of the model. The only thing that was noticed is for the 1,32 values the training is very slower compare to the original, while for len(training data) it takes seconds.

Part D: A Real Text Classification Task

1. **Build and evaluate a basic classifier for Subtask A**

For this task the LSTM model was used for binary Text Classification. First the tweets were pre-processed and then got ready to be the inputs for the model. The first "layer" in the architecture is a randomly initialized embedding layer, the second keras layer is an LSTM layer and the final keras layer is an output layer (using the sigmoid activation function).

```
[38]  print(model.summary())
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_3 (Embedding)      (None, 256, 100)          1897500
_________________________________________________________________
lstm_3 (LSTM)                (None, 100)               80400
_________________________________________________________________
dense_3 (Dense)              (None, 1)                 101
=================================================================
Total params: 1,978,001
Trainable params: 1,978,001
Non-trainable params: 0
_________________________________________________________________
None
```

```
print(results)
```

```
860/860 [==============================] - 2s 2ms/step
[0.5985215977180836, 0.7209302331126013]
```

2. Build and evaluate a basic classifier for Subtasks A-C combined, treating this as one single multi-class problem.

The same architecture and model as before are used also for the multi-classifier.

```
[40]  print(model2.summary())
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_4 (Embedding)      (None, 256, 100)          1897500
_________________________________________________________________
lstm_4 (LSTM)                (None, 100)               80400
_________________________________________________________________
dense_4 (Dense)              (None, 3)                 303
=================================================================
Total params: 1,978,203
Trainable params: 1,978,203
Non-trainable params: 0
_________________________________________________________________
None
```

```
[44] results2 = model2.evaluate(preprocessed_test_data, testmultilabels)

     print(results2)
```

```
860/860 [==============================] - 2s 2ms/step
[7.025197723299958, 0.7209302325581395]
```

3. Try to improve your overall accuracy on Subtasks A-C.

Firstly, a dropout layer was added to see if there is any improvement in the performance, different activation functions(relu,softmax) and extra hidden layer also used but neither of them improved the accuracy although the loss was decreased.