CS6910 : Deep Learning (for Computer Vision)

Irfaan Arif
ME18B048

Assignment 3

1 Part-A :

Train and Test your own Word2Vec

For this part we have to train our own Word2Vec model for the 'text8' dataset. The following models as described in class were implemented

(a) Bag-of-words
(b) Skip-gram
(c) LSTM-based models

The python script (nlppreprocessing.py) given in Moodle was used to preprocess the dataset.

All experiments and results below are on the outputs of the proprocessing script on the 'text8' dataset. The models were trained on a AWS instance (g4dn.xlarge) with varying training parameters. The model with the best validation accuracy among all the epochs was taken as the result for each experiment.

For each of these, in the word_to_idx generated from the text8 vocabulary, the index '0' is reserved for words which we might come across later but arent there in the vocabulary, 'OOV' words. Our train vocabulary size made from the 'text8' dataset is 253702.

Bag of words

For this we are using the continuous bag of words approach. The CBOW model architecture tries to predict the current target word (the center word) based on the source context words (the surrounding words with a window size). The model used is attached as Figure 1. We are using nn.NLLLoss on logsoftmaxed outputs with SGD as the optimizer, after training the loss plateaued to around **2.80**

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
         Embedding-1           [-1, 1, 8, 300]      76,110,900
            Linear-2              [-1, 8, 128]          38,528
              ReLU-3              [-1, 8, 128]               0
            Linear-4           [-1, 8, 253703]      32,727,687
        LogSoftmax-5           [-1, 8, 253703]               0
================================================================
Total params: 108,877,115
Trainable params: 108,877,115
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.00
Forward/backward pass size (MB): 31.00
Params size (MB): 415.33
Estimated Total Size (MB): 446.34
----------------------------------------------------------------
```

Figure 1: The CBOW model with window size 4

Skip-gram

Skip-gram on the other hand is used to predict the context word for a given target word. It's the reverse of CBOW algorithm. Here, target word is input and the context words are output. The model used is attached as Figure 2. For this skipgram implementation we are using 2 different embedding layers for the context and target, and both are trainable. For skipgram we take negative of the logsigmoid of the embedding product as the loss and SGD as the optimizer again, after training the total loss came to about **3.15**. We will be experimenting using both in Part B.

```python
class skipgram(nn.Module):
    def __init__(self, embedding_size, vocab_size):
        super(skipgram, self).__init__()

        self.embeddings_input = nn.Embedding(vocab_size, embedding_size)
        self.embeddings_context = nn.Embedding(vocab_size, embedding_size)
        self.vocab_size = vocab_size

        # Initialize both embedding tables with uniform distribution
        self.embeddings_input.weight.data.uniform_(-1,1)
        self.embeddings_context.weight.data.uniform_(-1,1)

    def forward(self, input_word, context_word):

        # computing out loss
        emb_input = self.embeddings_input(input_word)     # bs, emb_dim
        emb_context = self.embeddings_context(context_word)  # bs, emb_dim
        emb_product = torch.mul(emb_input, emb_context)     # bs, emb_dim
        emb_product = torch.sum(emb_product, dim=1)          # bs
        out_loss = F.logsigmoid(emb_product)

        return -(out_loss).mean()
```

Figure 2: The Skipgram model

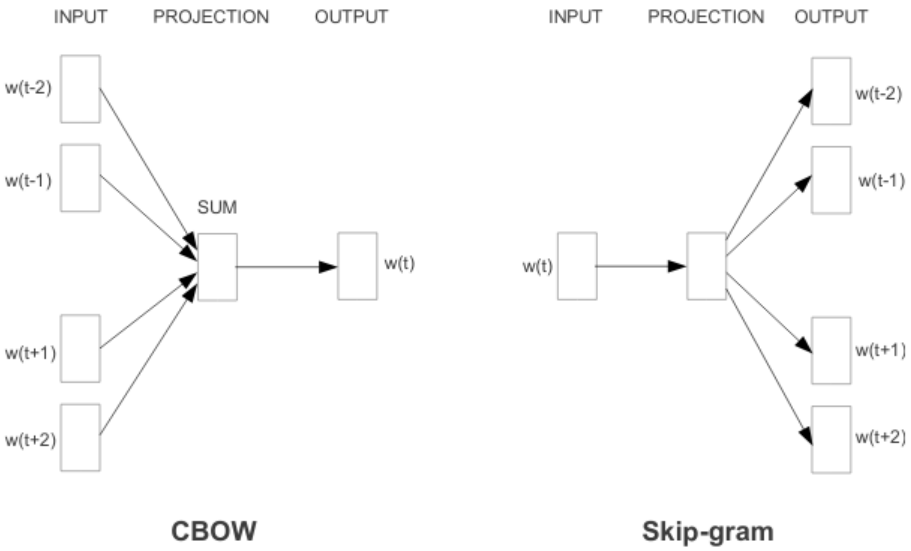


Figure 3: CBOW vs skipgram (source: arXiv:1301.3781)

LSTM-based models

For LSTM-based model, we can define the training task to predict a target word given context words before it, similar to what is done in the CBOW model. Here we take a target word as the output and try to predict it given window_size words before it as input. The model used is attached as Figure 4. We are again using nn.NLLLoss on the logsoftmaxed outputs with Adam as the optimizer, after training the loss plateaued to around **1.40**

```
EmbeddingLSTM(
  (embeddings): Embedding(253703, 300)
  (lstm): LSTM(300, 128, batch_first=True)
  (linear1): Linear(in_features=128, out_features=253703, bias=True)
  (act1): LogSoftmax(dim=-1)
)
```

Figure 4: LSTM-based model

2 Part-B :

Sentiment Analysis in movie reviews

The task given was to classify movie reviews in one of the 5 classes based on the sentiment of the review. We are to make a feature representation of the review using pre-trained word embedding & our own embedding which were trained in Part-A and use an LSTM based model to classify to one of the 5 classes mentioned below.

The sentiment labels are:

- 0 - negative
- 1 - somewhat negative
- 2 - neutral
- 3 - somewhat positive
- 4 - positive

Using pre-trained word embeddings

The following 4 pre-trained word embeddings were used from the torchtext library

- FastText - 'simple' - 300 dim
- GloVe - '6B' - 100 dim
- GloVe - '840B' - 300 dim
- FastText - 'en' - 300 dim

For all of these pre-trained word embeddings, experiments were run with different models such as by varying the number of LSTM layers, hidden dim, etc. The best results have only been included in this report. Also in the Embeddings layer for all the models, index '0' was reserved for words which might not not be in our vocabulary 'OOV' words. This is because some words in val.csv and test.csv might be missing from our vocabulary made from train.csv

And there were words in our vocabulary for which pre-trained word embeddings were not available (ref Table 1). For these few words, a random vector was initialized from the normal distribution (scale of 0.6). The same is also done for index '0' of our embedding weight matrix. For every other word for which an embedding was available, it was copied to the embedding weight matrix. Our vocabulary made from the reviews in train.csv had 15136 words.

Pre-trained word embedding	Word embeddings found	Words embeddings missing	%found
FastText - 'simple'	12072	3064	79.83
GloVe - '6B'	14742	394	97.39
GloVe - '840B'	14742	394	97.39
FastText - 'en'	**14845**	291	**98.08**

Table 1: Pre-trained word Embeddings availability

Experiments were run in which the embeddings weights were frozen and also in which they were also trainable. The results are tabulated below in Table 2 & 3. The model used for the experiments is attached as Figure 5

Pre-trained word embedding	%Test Accuracy
FastText - 'simple'	59.93
FastText - 'simple' - frozen	60.34
GloVe - '6B'	61.30
GloVe - '6B' - frozen	62.00
GloVe - '840B'	63.01
GloVe - '840B' - frozen	**63.77**
FastText - 'en'	62.53
FastText - 'en' - frozen	62.81

Table 2: Test accuracy for each model

Pre-trained word embedding	0	1	2	3	4
FastText - 'simple'	**20.17**	42.61	77.1	52.5	19.99
FastText - 'simple' - frozen	0.0	54.58	74.42	60.94	0.0
GloVe - '6B'	0.0	58.01	74.89	**61.49**	0.0
GloVe - '6B' - frozen	16.97	52.68	77.19	59.97	0.14
GloVe - '840B'	14.7	54.49	78.11	61.33	0.94
GloVe - '840B' - frozen	7.73	57.91	**78.85**	51.02	**39.54**
FastText - 'en'	1.32	**60.84**	76.71	60.23	0.14
FastText - 'en' - frozen	0.0	55.78	78.23	52.91	34.03

Table 3: Class wise accuracy for each

```
SentimentLSTM(
  (embedding): Embedding(15124, 300)
  (lstm): LSTM(300, 128, num_layers=2, batch_first=True)
  (fc1): Linear(in_features=128, out_features=5, bias=True)
  (act2): Sigmoid()
)
```

Figure 5: LSTM model used

Figures 6 to 21 have the model accuracy's over training and confusion matrix's for each of the pre-trained word embeddings with both the embedding layers frozen and trainable.

- FastText 'simple'

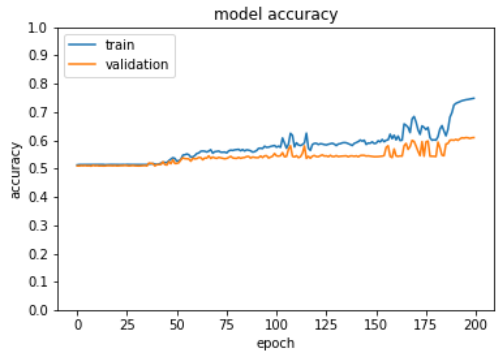


Figure 6: Embedding layer frozen

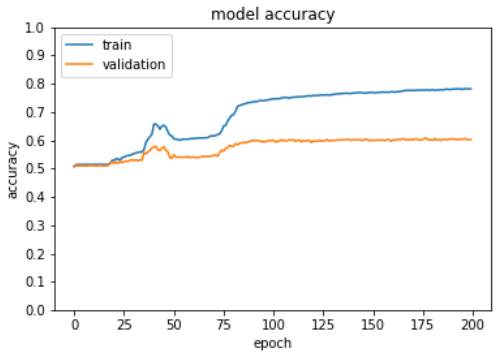


Figure 7: Embedding layer trainable

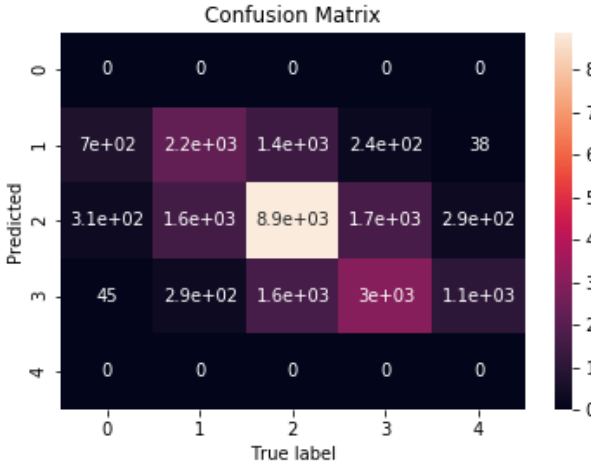


Figure 8: Embedding layer frozen

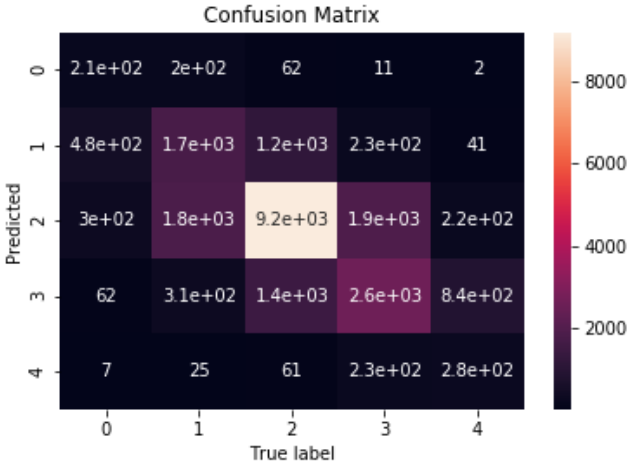


Figure 9: Embedding layer trainable

- GloVe - '6B'

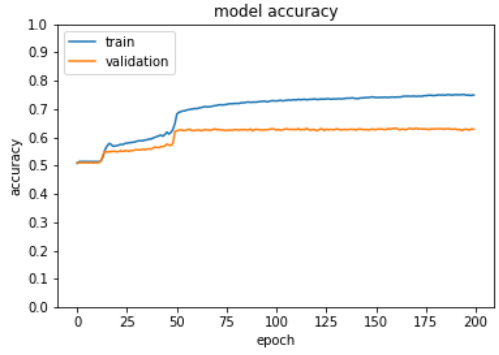


Figure 10: Embedding layer frozen

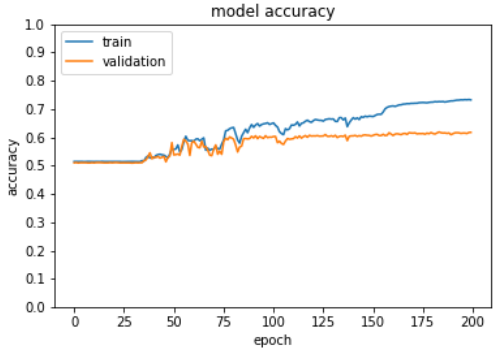


Figure 11: Embedding layer trainable

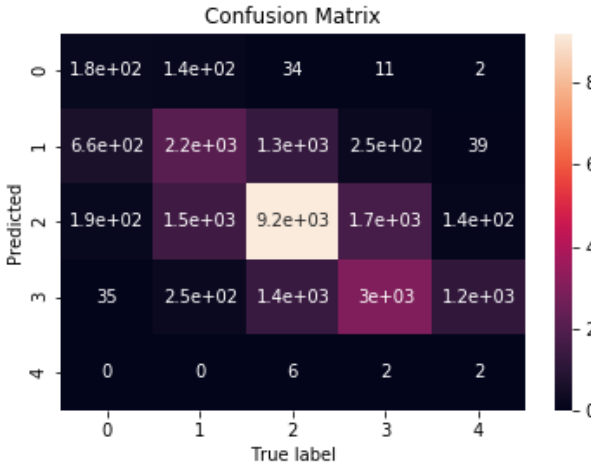


Figure 12: Embedding layer frozen

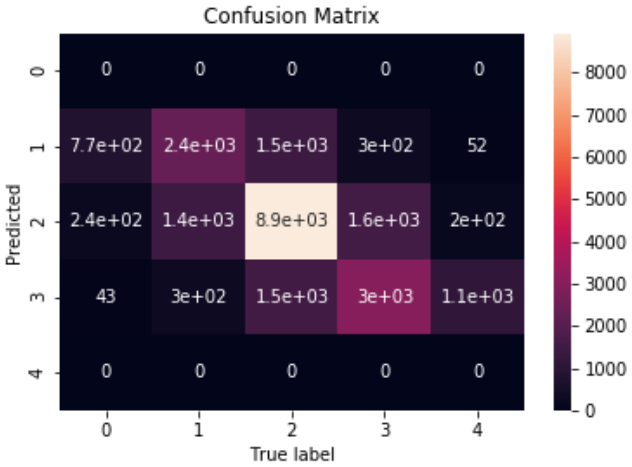


Figure 13: Embedding layer trainable

- GloVe - '840B'

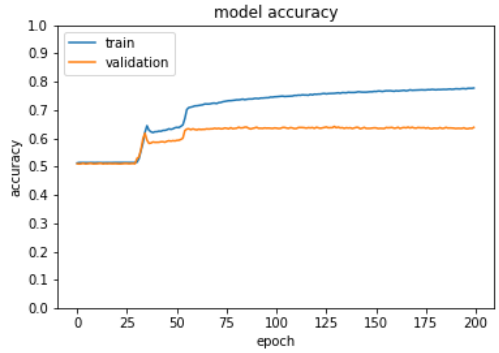


Figure 14: Embedding layer frozen

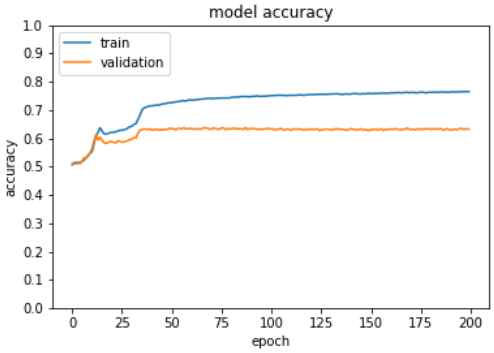


Figure 15: Embedding layer trainable

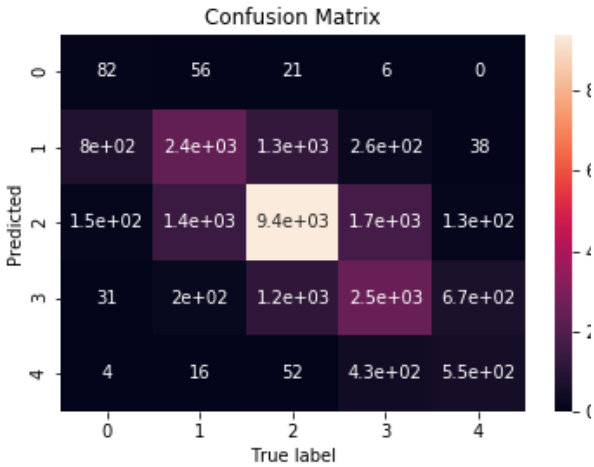


Figure 16: Embedding layer frozen

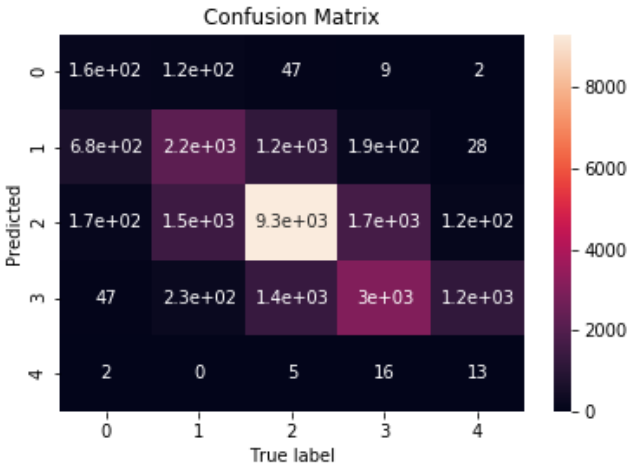


Figure 17: Embedding layer trainable

- FastText - 'en'

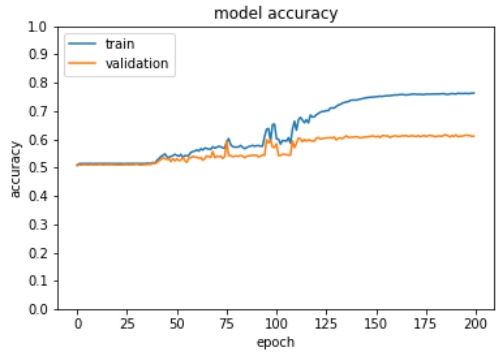


Figure 18: Embedding layer frozen

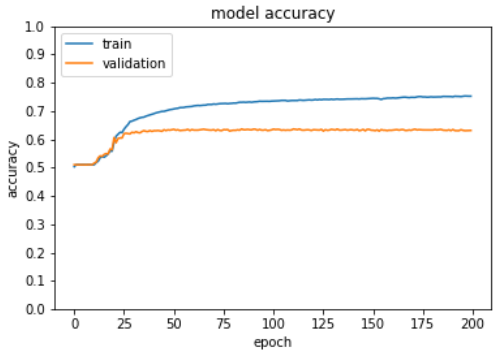


Figure 19: Embedding layer trainable

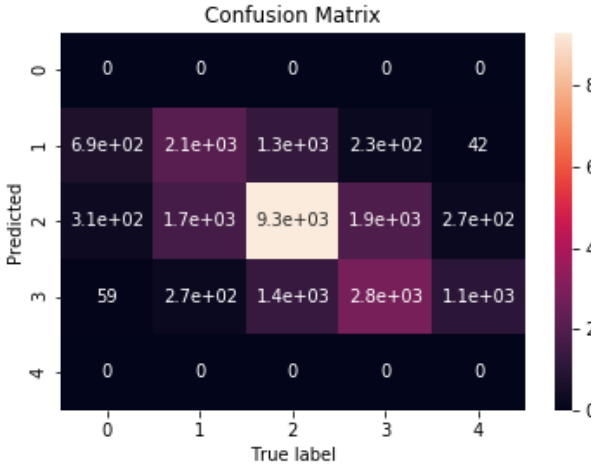


Figure 20: Embedding layer frozen

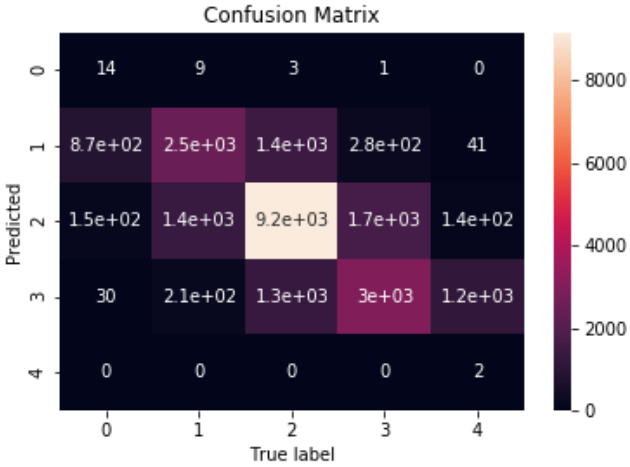


Figure 21: Embedding layer trainable

As can be seen from the accuracy tables and the confusion matrices, the model have very good performance for neutral category and relatively good performance for the somewhat negative and somewhat positive categories. Most of the models tend to find it hard to categorize the completely positive and negative reviews. Also since not many embedings were found in the FastText - 'simple' pre trained embedding, it as expected had the worst performance among these pre-trained word embeddings. Between the GloVe pre-trained word embeddings, since '6B' only had 100 dim embedding space compared to '840B' having 300 dim, GloVe '840' had better performance.

It is to note that the there were more number of reviews for the neutral category and the slighly positive and negative classes in the train.csv file as seen in table 4, this also leads to our models to lean towards predicting these majority classes as the training accuracy would be higher, as this is a training class-imbalance problem, techniques like under-sampling, minority oversampling could be used to take care of this.

Train data classes	0	1	2	3	4
Number of train samples	4650	18443	54488	22327	6086
Percentage of dataset	0.044	**0.174**	**0.514**	**0.211**	0.057

Table 4: Train dataset class imbalance

As a result the size of the embedding dimension and the number of word embeddings available for our vocabulary are the major factors to consider when selecting a pre-trained word embedding to use.

Using embeddings from the models in Part A

For each of the models trained in Part A, the word_to_idx dictionary was pickled and stored, also the weights of the embeddings layer in the models were also extracted and stored in a .npz compressed format. Now for every word in the vocabulary from train.csv, if the word in present in the loaded word_to_idx dictionary, then the trained embedding was used and copied to the weight matrix. The words for which embeddings were not available, a random vector was initialized from the normal distribution (scale of 0.6).The same is also done for index '0' of our embedding weight matrix for out of vocabulary 'OOV' words that might appear during validation and testing.

Word embedding from	Word embeddings found	Words embeddings missing	%found
Vocab from 'text8'	13832	1291	**91.46**

Table 5: Word Embeddings availability

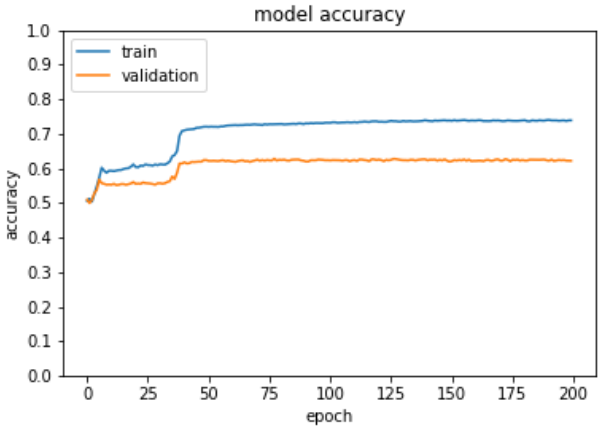
The model is used for the experiments below is basically the same as the one used when we were using pre-trained embeddings. It is attached as Figure 22.

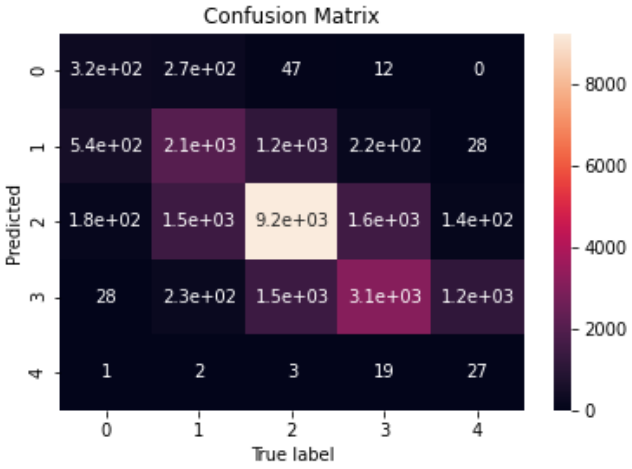
```
class SentimentLSTM(nn.Module):

    def __init__(self,corpus_size,output_size,embedd_dim,hidden_dim,n_layers):
        super().__init__()
        self.output_size = output_size
        self.n_layers = n_layers
        self.hidden_dim = hidden_dim

        self.embedding = nn.Embedding.from_pretrained(weights_matrix)
        self.lstm = nn.LSTM(embedd_dim, hidden_dim,n_layers,batch_first=True)
        self.dropout = nn.Dropout(0.3)
        self.fc = nn.Linear(hidden_dim,output_size)
        self.act = nn.Sigmoid()

    def forward(self,x,hidden):
        batch_size = x.size(0)
        embeds = self.embedding(x)
        lstm_out, hidden = self.lstm(embeds,hidden)
        lstm_out = lstm_out.contiguous().view(-1,self.hidden_dim)
        out = self.dropout(lstm_out)
        out = self.fc(out)
        out = self.act(out)
        out = out.view(batch_size,-1)
        out = out[:,-5:]
        return out, hidden

    def init_hidden(self,batch_size):
        weight = next(self.parameters()).data
        hidden = (weight.new(self.n_layers, batch_size, self.hidden_dim).zero_(),
                  weight.new(self.n_layers, batch_size, self.hidden_dim).zero_())
        return hidden
```

Figure 22: LSTM model used

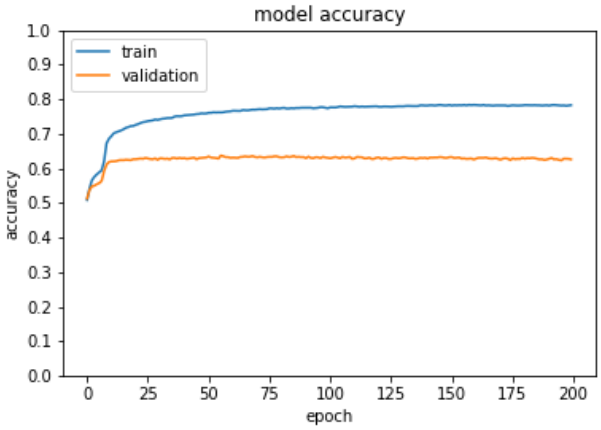
Figures attached below have the model accuracy's over training and confusion matrix's for each of the models by which word embeddings have been trained in Part A. For these models we have kept the weights frozen from what we got from Part A.

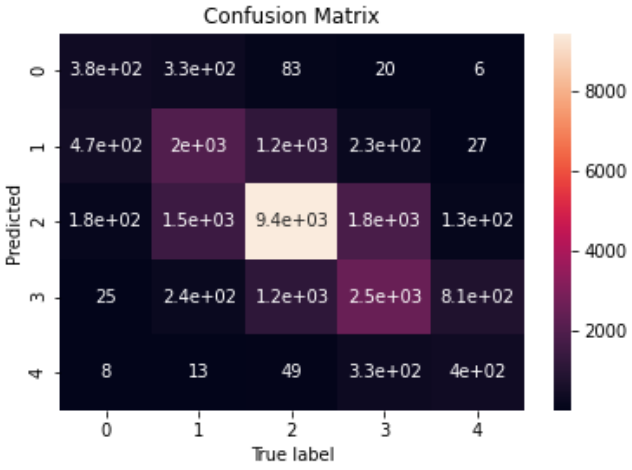
- CBOW model



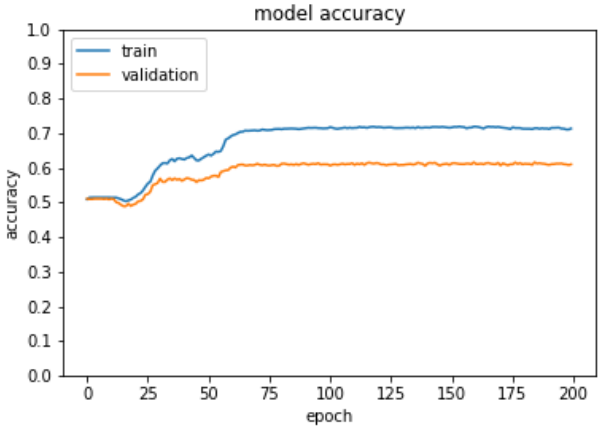


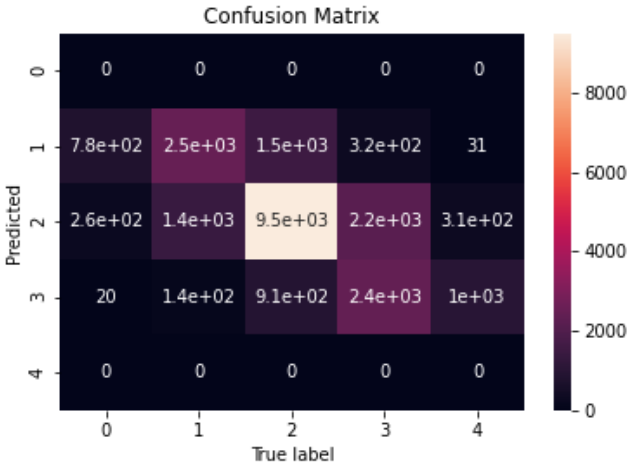
- Skipgram





- LSTM based model





The relevant metrics for these experiments for each of the models have also been attached in Table 6 and Table 7

Model based on	%Test Accuracy
CBoW	**63.78**
Skipgram	63.17
LSTM - based	61.60

Table 6: Test accuracy for each model

Model based on	0	1	2	3	4
CBoW	29.78	50.57	77.44	**63.43**	1.96
Skipgram	**36.0**	48.86	79.01	51.27	**29.18**
LSTM - based	0.0	**61.38**	**79.53**	48.94	0.0

Table 7: Class wise accuracy for each

Conclusions

After all the experiments the main parameters that seems to affect the accuracy's of our models are:

1. **Embedding dimension** of the embedding layer

   For pre-trained embedding the embedding dimension available seems to be directly correlated to the amount of data used to create it, as a result higher the embedding dimension, better the accuracy as discussed before in its section.

   For the model in Part A, it seems that bigger the embedding dimension the better the results obtained from using it in Part B for sentiment classification. The results have been tabulated in Table 8. As the embedding dimension increases the training time drastically increases especially for a large dataset like 'text8' with very minimal improvements in accuracy, hence only 3 values were tested with.

Embedding dimension	% Accuracy
100	61.20
300	63.02
500	63.78

Table 8: Accuracy for varying embedding dimension (CBoW model was used)

2. **Sequence length** of input to the LSTM

   As the input to the LSTM must be fixed, we either truncate long review vector or pad short review vectors with 0 to achieve the sequence length we specify. The results have been tabulated in Table 9. Here also as the sequence length increases the accuracy does increase however after a while you reach the point of very little gain from further increases. Note that all very short and long reviews were also completely excluded from the training process since it would just be bad training data.

Seq length	% Accuracy
4	54.67
8	63.78
12	63.83

Table 9: Accuracy for varying sequence lengths (CBoW model was used)

3. **Window sizes** for each model in Part-A

- **CBoW model**

  For the cbow model, here window size refers to the number of context words around the middle target word used to predict it. The results of varying the window size are attached in Table 10

Window size	% Accuracy
2	60.67
4	63.78
6	**63.83**

Table 10: Accuracy for varying window size in CBoW

- **Skipgram model**

  In the skipram implementation, window size refers to how many context words around the target word are taken to use as training data. The table below, 11 shows the results of varying this window size.

Window size	% Accuracy
4	**63.17**
8	63.06
10	62.91

Table 11: Accuracy for varying window size in Skipgram

- **LSTM based model**

  In the case of LSTM based models, as we have treated it like a word prediction problem, here window size refers to how many words before the target we take to predict the last word. Table 12 show the results obtained by varying this window size.

Window size	% Accuracy
2	59.33
4	**61.60**
6	61.56

Table 12: Accuracy for varying window size in LSTM based model