Informatics Institute of Technology
In Collaboration With

Robert Gorden University

MSc in Big Data Analytics

CMM706 – Text Analytics

Coursework
Comparative Study Report

R.M.Maneesha Indrachapa

IIT ID - 20200399

RGU ID - 2016894

1.0 Introduction

Natural language processing (NLP) is one of the rapidly changing and innovative fields which is a subcategory in artificial intelligence which helps computers to understand, interpret and manipulate human language. Natural Language Processing draws many disciplines which include computer science and computational linguistics. Computers are always good at structured data such as databases and financial records, but they can't understand natural language. So, to get computers closer to the human level understanding of language need Natural Language Processing.
Traditionally, sequential algorithms such as SVM and Logistic regression were widely used due to the sequential nature of human language. Human languages depend on the order of works as well as the context in which they are spoken. In traditional methods, linguistic information was represented with sparse representations (high-dimensional features), but according to the recent research results, low-dimensional, distributed representations, neural-based models outperform the accuracy of traditional models.

2.0 Deep Learning for NLP

Deep learning is an area where machine learning use the power of neural networks to create a high accuracy, prediction models, where it normally needs a high volume of data to improve the accuracy of the model if the low volume of data is used for building the neural network it is tend to overfit for the particular data set and we cannot expect it to give a good results for the particular predictions.

2.1 Convolutional Neural Networks (CNN)

A CNN or Convolutional Neural Network is a technique that represents a feature function that can be applied to n-grams and words to extract features. These determined features can be used for machine translation, sentiment analysis, search algorithms and question-answer knowledge systems.Collobert and Weston were among the first researchers to apply CNN-based frameworks to NLP tasks. In the CNN model, sentences are tokenized and converted to a matrix of D dimension. convolutional filters are applied on top of it and a feature map is created. Finally, a max-pooling operation is performed to reduce the dimensions of the output. Due to this approach, CNN shows state of the art performance when it comes to implementing aspect detection and NER. This is achieved by using a window-based approach where a fixed-length window is used to detect the neighboring words. However, the drawback of this approach is, it is not efficient in identifying long-distance dependencies. Techniques such as merging time-delayed neural networks (TDNN) with CNN can be used to overcome this problem.

2.2 Recurrent Neural Network (RNN)

RNNs or Recurrent Neural Networks are the neural-based approaches that are specialized and effective at processing sequential information. An RNN recursively applies a computation to every instance of an input sequence conditioned on the previous computed results. These sequences are typically represented by a fixed-size vector of tokens which are fed one by one sequentially to a recurrent unit. The main advantage of an RNN is, it's ability to memorize the prevailing computation results and use that information in the current computation. This makes RNN models suitable to model context dependencies in inputs of arbitrary length so as to create a proper composition of the input. RNNs have been used to study various NLP tasks such as machines.

2.3 Sequence-to-sequence models

Normally, a grouping-to-succession model comprises two intermittent neural systems: an encoder that forms the information and a decoder that delivers the yield. Encoder and decoder can utilize the equivalent or various arrangements of boundaries. Arrangement-to-Sequence models are fundamentally utilized being referred to as noting frameworks, chatbots, and machine interpretation. Such multi-layer cells have been effectively utilized in arrangement-to-succession models for interpretation in Sequence to Sequence Learning with Neural Networks study. In Paraphrase Detection Using Recursive Autoencoder, a novel recursive autoencoder engineering is introduced. The portrayals are vectors in an n-dimensional semantic space where phrases with comparable implications are near one another.

2.4 Word Embeddings

Word embeddings mainly convert text words into vector format with suitable dimensions which computer algorithms can perform. In the present time researchers came up with different word embedding approaches for different language-related processing (e.g.: Fasttext, Word2Vec, BERT, Glove, Flair embeddings etc.). When building word embeddings on the collection of text we can use different embedding styles with selected different vector dimensions for our specific implementation of computer algorithms. Below models are already trained using Deep Learning techniques.

2.4.1. GloVe – Global Vectors

GloVe is an unsupervised learning algorithm for obtaining vector representations for words. Training is performed on aggregated global word-word co-occurrence statistics from a corpus, and the resulting representations showcase interesting linear substructures of the word vector space. GloVe trains word embeddings by performing a weighted factorization of the log of the word co-occurrence matrix. The model scales to very large corpora and performs well on word analogy tasks. Count models, like a GloVe, Learn the vectors by essentially doing some sort of dimensionality reduction on the co-occurrence counts matrix. They start by constructing a matrix with counts of word co-occurrence information, each row tells how often a word occurs with every other word in some defined context-size in a large corpus. This matrix is then factorized, resulting in a lower dimension matrix, where each row is some vector representation for each word.

The dimensionality reduction is typically done by minimizing some kind of 'reconstruction loss' that finds lower-dimension representations of the original matrix and which can explain most of the variance in the original high-dimensional matrix.

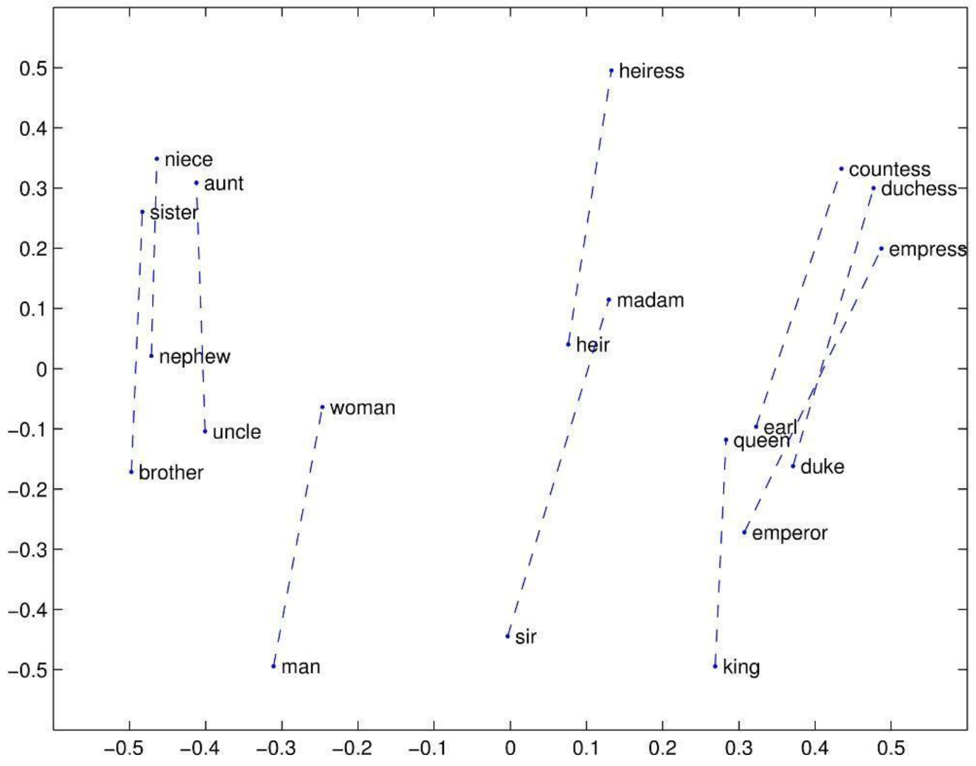


Figure 2.4.1: Similarity Visualization

3. Conclusion

Compared with a CNN model, an RNN model can be similarly effective or even better at specific natural language tasks but not necessarily superior. This is because RNN models very different aspects of the data, which only makes them effective depending on the semantics required by the task at hand. The input expected by a RNN are typically one-hot encodings or word embeddings, but in some cases they are coupled with the abstract representations constructed by, say, a CNN model. Even though Sequence-to-sequence models have proven to be strong in conversational processing, it is not best when working with sequential and complex sentences. According to many studies, RNN can outperform CNN when processing sequential data such as natural language sentences. Text content such as tweets can have multiple sentences and the prevailing ones referring to the information from the previous ones. Therefore it is evident that we need an algorithm which not only excels in text classification, but also it should have the ability to retain the earlier information. Since LSTM has the ability to preserve long-term memory, it can be considered as the best fit for tasks such as tweet classification.

4.0 References

[1] B. Hettige, A. S. Karunananda, and G. Rzevski, "Existing Systems and Approaches for Machine Translation: A Review," p. 9, 2011.

[2] K. Erk and S. Padó, "A structured vector space model for word meaning in context," in Proceedings of the Conference on Empirical Methods in Natural Language Processing - EMNLP '08, Honolulu, Hawaii, 2008, p. 897.

[3] 1.12. Multiclass and multilabel algorithms — scikit-learn 0.23.1 documentation. 2020. 1.12. Multiclass and multilabel algorithms — scikit-learn 0.23.1 documentation. [ONLINE] Available at:https://scikit-learn.org/stable/modules/multiclass.html. [Accessed 20 July 2021].

[4] Elvis. 2020. Deep Learning for NLP: An Overview of Recent Trends | by Elvis | dair.ai | Medium. [ONLINE] Available at:https://medium.com/dair-ai/deep-learning-for-nlp-an-overview-of-recent-trends-d0d8f40a776d.[Accessed 20 July 2021].

[5] FloydHub Blog. 2020. Ten trends in Deep learning NLP. [ONLINE] Available at:https://blog.floydhub.com/ten-trends-in-deep-learning-nlp/. [Accessed 20 July 2021].

[6] Mathéo Daly. 2020. Natural Language Processing Classification Using Deep Learning And Word2Vec | by Mathéo Daly | Towards Data Science. [ONLINE] Available at:https://towardsdatascience.com/natural-language-processing-classification-using-deep-learning-and-word2vec-50cbadd3bd6a. [Accessed 20 July 2021].

[7] Young, T., Hazarika, D., Poria, S. and Cambria, E. (2018). Recent Trends in Deep Learning Based Natural Language Processing [Review Article]. IEEE Computational Intelligence Magazine , 13(3), pp.55-75

[8]"Deep learning for search: Using word2vec - JAXenter." [Online]. Available: https://jaxenter.com/deep-learning-search-word2vec-147782.html. [Accessed: 29-Nov-2019].

[9]J. Pennington, R. Socher, and C. Manning, "Glove: Global Vectors for Word Representation," in Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP), Doha, Qatar, 2014, pp. 1532–1543.

[10]K. Nakayama, T. Hara, and S. Nishio, "Wikipedia Mining for an Association Web Thesaurus Construction," in Web Information Systems Engineering – WISE 2007, vol. 4831, B. Benatallah, F. Casati, D. Georgakopoulos, C. Bartolini, W. Sadiq, and C. Godart, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 322–334.

[11]W. J. H. H. L. Somers, An introduction to machine translation, vol. 362. London: Academic Press.

Appendix

A.1.0 Create LDA Model and Label the Train Data with Cluster Names

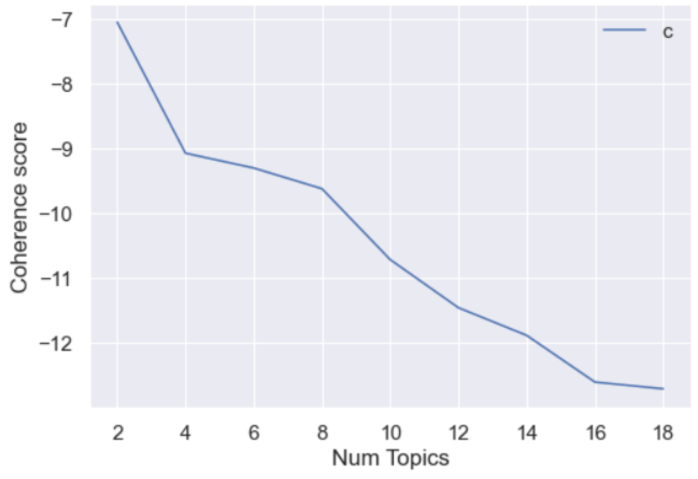
```
#create a corpus
corpus_gensim =suggestion_df.iloc[:,6]
id2word = corpora.Dictionary(corpus_gensim)

corpus = [id2word.doc2bow(text) for text in corpus_gensim]
```

```
lda_model = gensim.models.ldamodel.LdaModel(corpus=corpus,
                                            id2word=id2word,
                                            num_topics=20,
                                            random_state=100,
                                            update_every=1,
                                            chunksize=100,
                                            passes=10,
                                            alpha='auto',
                                            per_word_topics=True)
```

```
def compute_coherence_values(dictionary, corpus, texts, limit, start=2, step=3):
    coherence_values = []
    model_list = []
    for num_topics in range(start, limit, step):
        model = gensim.models.ldamodel.LdaModel(corpus=corpus, num_topics=num_topics, id2word=id2word)
        model_list.append(model)
        coherencemodel = CoherenceModel(model=model, texts=texts, dictionary=dictionary, coherence='u_mass')
        coherence_values.append(coherencemodel.get_coherence())
    return model_list, coherence_values
```

```
model_list, coherence_values = compute_coherence_values(dictionary=id2word, corpus=corpus, texts=corpus_gensim, start=2, limit=20, step=2)
```

```
limit=20; start=2; step=2;
x = range(start, limit, step)
plt.plot(x, coherence_values)
plt.xlabel("Num Topics")
plt.ylabel("Coherence score")
plt.legend(("coherence_values"), loc='best')
plt.show()
```



```python
for m, cv in zip(x, coherence_values):
    print("Num Topics =", m, " has Coherence Value of", round(cv, 4))
```

```
Num Topics = 2  has Coherence Value of -7.0421
Num Topics = 4  has Coherence Value of -9.0644
Num Topics = 6  has Coherence Value of -9.2931
Num Topics = 8  has Coherence Value of -9.6125
Num Topics = 10  has Coherence Value of -10.7075
Num Topics = 12  has Coherence Value of -11.4505
Num Topics = 14  has Coherence Value of -11.8772
Num Topics = 16  has Coherence Value of -12.5982
Num Topics = 18  has Coherence Value of -12.7023
```

If the coherence score seems to keep increasing, it may make better sense to pick the model that gave the highest CV before flattening out. This is exactly the case here.Here We can use 12 Topics

```python
# Select the model and print the topics
optimal_model = model_list[5]
model_topics = optimal_model.show_topics(formatted=False)
print(optimal_model.print_topics(num_words=10))
```

```
[(0, '0.016*"would" + 0.012*"breakfast" + 0.010*"money" + 0.010*"book" + 0.010*"stay" + 0.010*"pool" + 0.010*"good" + 0.010*"recommend" + 0.008*"make" + 0.00
'0.018*"get" + 0.014*"room" + 0.013*"stay" + 0.012*"place" + 0.012*"recommend" + 0.011*"would" + 0.011*"go" + 0.010*"berlin" + 0.010*"want" + 0.009*"free"'),
el" + 0.022*"room" + 0.014*"recommend" + 0.011*"walk" + 0.011*"go" + 0.011*"need" + 0.011*"avoid" + 0.010*"staff" + 0.008*"bring" + 0.008*"tour"'), (3, '0.02
*"hotel" + 0.019*"go" + 0.012*"need" + 0.011*"would" + 0.011*"small" + 0.009*"less" + 0.009*"walk" + 0.008*"want" + 0.008*"eat"'), (4, '0.037*"would" + 0.026
*"recommend" + 0.012*"room" + 0.010*"stay" + 0.009*"take" + 0.008*"star" + 0.008*"even" + 0.008*"tip" + 0.008*"need"'), (5, '0.039*"hotel" + 0.021*"recommend
+ 0.014*"would" + 0.012*"highly" + 0.009*"go" + 0.009*"avoid" + 0.008*"sure" + 0.008*"get" + 0.007*"seattle"'), (6, '0.022*"want" + 0.017*"hotel" + 0.013*"ge
```

```python
# Compute Perplexity
print('\nPerplexity: ', optimal_model.log_perplexity(corpus))  # a measure of how good the model is. lower the better.

# Compute Coherence Score
coherence_model_lda = CoherenceModel(model=optimal_model, texts=corpus_gensim, dictionary=id2word, coherence='u_mass')
coherence_lda = coherence_model_lda.get_coherence()
print('\nCoherence Score: ', coherence_lda)
```

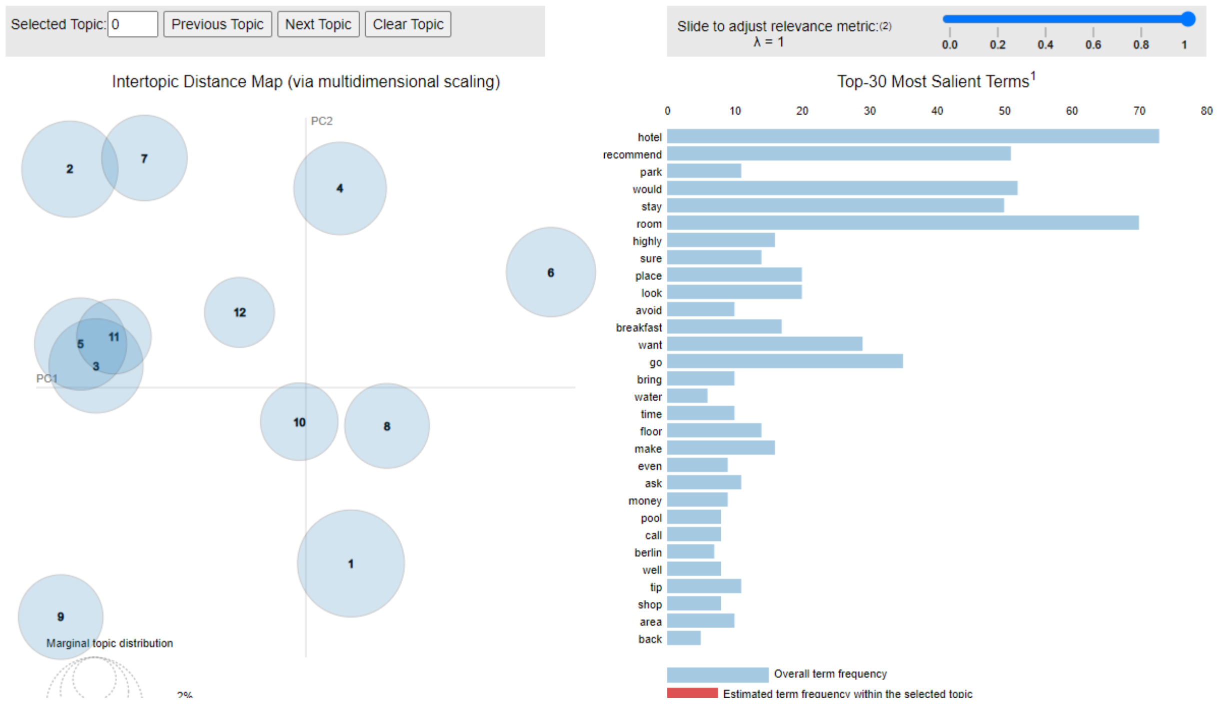
```
Perplexity:  -7.662278704023772

Coherence Score:  -11.450456622572995
```

```python
suggestion_original_reviews=suggestion_df['reviews'].tolist()
suggestion_original_cleaned=[]
suggestion_df['stop_words_removed_lemmatized'].apply(lambda x:array_to_str(x,suggestion_original_cleaned))

def format_topics_sentences(ldamodel=optimal_model, corpus=corpus):
    # Init output
    sent_topics_df = pd.DataFrame()
    # Get main topic in each document
    for i, row in enumerate(ldamodel[corpus]):
        row = sorted(row, key=lambda x: (x[1]), reverse=True)
        aspects =[]
        # Get the Dominant topic, Perc Contribution and Keywords for each document
        for j, (topic_num, prop_topic) in enumerate(row):
            wp = ldamodel.show_topic(topic_num)
            topic_keywords = ", ".join([word for word, prop in wp])
            val = False
            if j==0: #j<3 for prominent topics 3
#                 if(len(aspects)<3):
#                     if(topic_num not in aspects):
#                         aspects.append(topic_num)
#                     if(j==2):
                aspects.append(topic_num)
                sent_topics_df = sent_topics_df.append(pd.Series([aspects, round(prop_topic,4), topic_keywords]), ignore_index=True)
            else:
                break
    sent_topics_df.columns = ['Dominant_Topic', 'Perc_Contribution', 'Topic_Keywords']

    # Add original text to the end of the output
    contents = pd.Series(suggestion_original_reviews)
    cleaned = pd.Series(suggestion_original_cleaned)
    sent_topics_df = pd.concat([sent_topics_df, contents,cleaned], axis=1)
    return(sent_topics_df)


df_topic_sents_keywords = format_topics_sentences(ldamodel=optimal_model, corpus=corpus)

# Format
df_dominant_topic_12 = df_topic_sents_keywords.reset_index()
df_dominant_topic_12.columns = ['Document_No', 'Dominant_Topic', 'Topic_Perc_Contrib', 'Keywords', 'Text','Cleaned Data']

# Show
print(df_dominant_topic_12.shape)
df_dominant_topic_12.to_csv('./clustered_data.csv', encoding='utf-8')
df_dominant_topic_12.head()
```

```
(448, 6)
```

Save the labeled training data in clustered_data.csv for further use, and use pyLDAvis to see the cluster's distribution.

```
pyLDAvis.enable_notebook()
vis = pyLDAvis.gensim_models.prepare(optimal_model, corpus, id2word)
vis
```



A.1.1 Test and Train Data for NN

```
import pandas as pd
import numpy as np
import nltk

# This data is the processed data written after doing the first part of the course work
textDf = pd.read_csv('./clustered_data.csv')
textDf.isnull().values.any()

#split data to X and Y(labels)
textDf_dataset = textDf['Cleaned Data']
textDf_labels = textDf['Dominant_Topic']

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# split in to a test set
label_encoder = LabelEncoder()
text_X_train, text_X_test, text_Y_train, text_Y_test \
= train_test_split(textDf_dataset, label_encoder.fit_transform(textDf_labels), test_size=0.2, random_state=1)
```

```python
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# This will be the maximum number of words from our resulting tokenized data vocabulary which are to be used,
#truncated after the 1000 most common words in our case
num_words = 1000
# our maximum sentence length will be determined by searching our sentences for the one of maximum length,
# and padding characters will be '0'
pad_type = 'post'

# Tokenize our training data
tokenizer = Tokenizer(num_words=num_words)
tokenizer.fit_on_texts(text_X_train)

# Get our training data word index
word_index = tokenizer.word_index

# Encode training data sentences into sequences
train_sequences = tokenizer.texts_to_sequences(text_X_train)
test_sequences = tokenizer.texts_to_sequences(text_X_test)
# val_sequences = tokenizer.texts_to_sequences(text_X_val)

# Get max training sequence length
maxlen = max([len(x) for x in train_sequences])

# Pad the training sequences (pad_sequece makes the sentences in input dataset uniform in length)
X_train = pad_sequences(train_sequences, padding=pad_type,maxlen=maxlen)
X_test = pad_sequences(test_sequences, padding=pad_type,maxlen=maxlen)

Y_train = to_categorical(text_Y_train)
Y_test = to_categorical(text_Y_test)

vocabSize = len(tokenizer.word_index) + 1  # Adding 1 because of reserved 0 index

# Output the results of our work
print("\nVocab size:\n", vocabSize)
print("\nPadded training sequences:\n", X_train[0, :10])
print("\nPadded training shape:", X_train.shape)
print("Training sequences data type:", type(train_sequences))
print("Padded Training sequences data type:", type(X_train))
```

A.1.2 Create Embedded dictionary using GloVe

```python
from numpy import asarray
from numpy import zeros

glove_file = open("./glove/glove.6B.100d.txt", encoding="utf8")
embeddings_dictionary = dict()

for line in glove_file:
    records = line.split()
    word = records[0]
    vector_dimensions = asarray(records[1:], dtype='float32')
    embeddings_dictionary [word] = vector_dimensions

glove_file.close()
```

```python
# It's used check whether the passed word is available in the dictionary
embedding_matrix = zeros((vocabSize, 100))
for word, index in tokenizer.word_index.items():
    embedding_vector = embeddings_dictionary.get(word)
    if embedding_vector is not None:
        embedding_matrix[index] = embedding_vector
```

A.1.3 Create LSTM Model using RNN

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers
from tensorflow.keras.layers import SpatialDropout1D, LSTM, Dense, InputLayer

embeddingDim = 100

model = Sequential()
# The first layer is the embedded layer that uses max length vectors to represent each word.
model.add(layers.Embedding(input_dim=vocabSize,
                           output_dim=embeddingDim,
                           weights=[embedding_matrix],
                           input_length=maxlen))
model.add(SpatialDropout1D(0.2))
# The next layer is the LSTM layer with 256 memory units.
model.add(layers.LSTM(512, dropout=0.2, recurrent_dropout=0.2))
# The output layer must create 13 output values, one for each class.
model.add(Dense(12, activation='softmax'))
# Activation function is softmax for multi-class classification.
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model.summary())
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
embedding (Embedding)       (None, 31, 100)           123200

spatial_dropout1d (SpatialDr (None, 31, 100)          0

lstm (LSTM)                 (None, 512)               1255424

dense (Dense)               (None, 12)                6156
=================================================================
Total params: 1,384,780
Trainable params: 1,384,780
Non-trainable params: 0
_________________________________________________________________
None
```

A.1.4 Create CNN Model

```python
# Setup a Convulation Neural Network (CNN)
model_cnn = Sequential()
model_cnn.add(layers.Embedding(vocabSize, 100, input_length=maxlen))

model_cnn.add(layers.Conv1D(filters=512, kernel_size=3, padding='same', activation='relu'))
model_cnn.add(layers.MaxPooling1D(pool_size=2))
model_cnn.add(layers.Flatten())
model_cnn.add(layers.Dense(250, activation='relu'))

model_cnn.add(layers.Dense(12, activation='softmax'))
model_cnn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model_cnn.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
embedding_1 (Embedding)     (None, 31, 100)           123200

conv1d (Conv1D)             (None, 31, 512)           154112

max_pooling1d (MaxPooling1D) (None, 15, 512)          0

flatten (Flatten)           (None, 7680)              0

dense_1 (Dense)             (None, 250)               1920250

dense_2 (Dense)             (None, 12)                3012
=================================================================
Total params: 2,200,574
Trainable params: 2,200,574
Non-trainable params: 0
_________________________________________________________________
```

A.1.5 Plot Training and validation history for LSTM

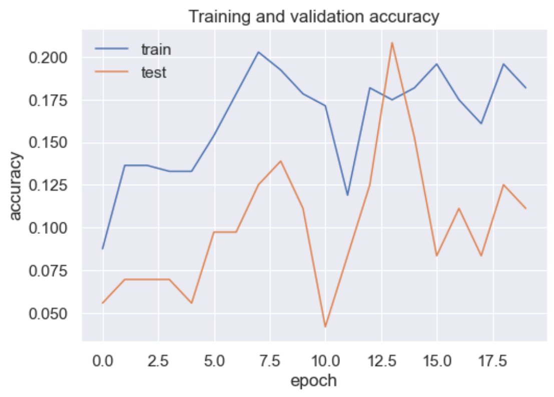
```python
history = model.fit(X_train, Y_train, batch_size=512, epochs=20, verbose=1, validation_split=0.2)
```

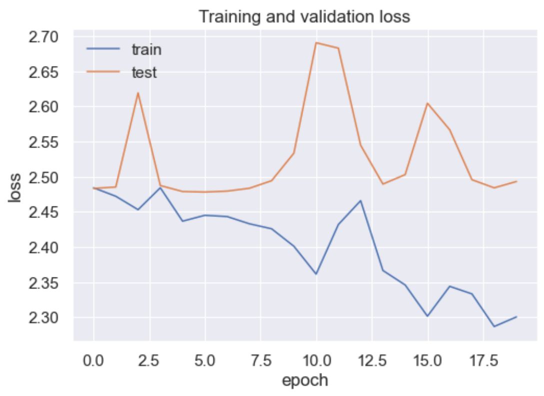
```
Epoch 1/20
1/1 [==============================] - 1s 578ms/step - loss: 2.4842 - accuracy: 0.0874 - val_loss: 2.4833 - val_accuracy: 0.0556
Epoch 2/20
1/1 [==============================] - 0s 165ms/step - loss: 2.4721 - accuracy: 0.1364 - val_loss: 2.4853 - val_accuracy: 0.0694
Epoch 3/20
1/1 [==============================] - 0s 163ms/step - loss: 2.4531 - accuracy: 0.1364 - val_loss: 2.6191 - val_accuracy: 0.0694
Epoch 4/20
1/1 [==============================] - 0s 215ms/step - loss: 2.4842 - accuracy: 0.1329 - val_loss: 2.4874 - val_accuracy: 0.0694
Epoch 5/20
1/1 [==============================] - 0s 187ms/step - loss: 2.4367 - accuracy: 0.1329 - val_loss: 2.4789 - val_accuracy: 0.0556
Epoch 6/20
1/1 [==============================] - 0s 159ms/step - loss: 2.4451 - accuracy: 0.1538 - val_loss: 2.4781 - val_accuracy: 0.0972
Epoch 7/20
1/1 [==============================] - 0s 178ms/step - loss: 2.4433 - accuracy: 0.1783 - val_loss: 2.4795 - val_accuracy: 0.0972
Epoch 8/20
1/1 [==============================] - 0s 180ms/step - loss: 2.4330 - accuracy: 0.2028 - val_loss: 2.4835 - val_accuracy: 0.1250
Epoch 9/20
1/1 [==============================] - 0s 173ms/step - loss: 2.4259 - accuracy: 0.1923 - val_loss: 2.4942 - val_accuracy: 0.1389
Epoch 10/20
1/1 [==============================] - 0s 216ms/step - loss: 2.4011 - accuracy: 0.1783 - val_loss: 2.5334 - val_accuracy: 0.1111
Epoch 11/20
1/1 [==============================] - 0s 168ms/step - loss: 2.3615 - accuracy: 0.1713 - val_loss: 2.6904 - val_accuracy: 0.0417
Epoch 12/20
1/1 [==============================] - 0s 170ms/step - loss: 2.4320 - accuracy: 0.1189 - val_loss: 2.6826 - val_accuracy: 0.0833
Epoch 13/20
1/1 [==============================] - 0s 171ms/step - loss: 2.4659 - accuracy: 0.1818 - val_loss: 2.5447 - val_accuracy: 0.1250
Epoch 14/20
1/1 [==============================] - 0s 164ms/step - loss: 2.3666 - accuracy: 0.1748 - val_loss: 2.4895 - val_accuracy: 0.2083
Epoch 15/20
1/1 [==============================] - 0s 169ms/step - loss: 2.3461 - accuracy: 0.1818 - val_loss: 2.5029 - val_accuracy: 0.1528
Epoch 16/20
1/1 [==============================] - 0s 183ms/step - loss: 2.3016 - accuracy: 0.1958 - val_loss: 2.6044 - val_accuracy: 0.0833
Epoch 17/20
1/1 [==============================] - 0s 170ms/step - loss: 2.3441 - accuracy: 0.1748 - val_loss: 2.5667 - val_accuracy: 0.1111
Epoch 18/20
1/1 [==============================] - 0s 166ms/step - loss: 2.3333 - accuracy: 0.1608 - val_loss: 2.4956 - val_accuracy: 0.0833
Epoch 19/20
1/1 [==============================] - 0s 166ms/step - loss: 2.2870 - accuracy: 0.1958 - val_loss: 2.4841 - val_accuracy: 0.1250
Epoch 20/20
1/1 [==============================] - 0s 174ms/step - loss: 2.3007 - accuracy: 0.1818 - val_loss: 2.4931 - val_accuracy: 0.1111
```

```python
# Below method will be printing the test score and accurecy of the system
score = model.evaluate(X_test, Y_test, verbose=1)
print("Test Score:", score[0])
print("Test Accuracy:", score[1])
```

```
3/3 [==============================] - 0s 57ms/step - loss: 2.5267 - accuracy: 0.1222
Test Score: 2.5266802310943604
Test Accuracy: 0.12222222238779068
```

```python
import matplotlib.pyplot as plt

# Below method used to used to print the
def print_plot_history(history):
    plt.plot(history.history['accuracy'])
    plt.plot(history.history['val_accuracy'])

    plt.title('Training and validation accuracy')
    plt.ylabel('accuracy')
    plt.xlabel('epoch')
    plt.legend(['train','test'], loc='upper left')
    plt.show()

    plt.plot(history.history['loss'])
    plt.plot(history.history['val_loss'])

    plt.title('Training and validation loss')
    plt.ylabel('loss')
    plt.xlabel('epoch')
    plt.legend(['train','test'], loc='upper left')
    plt.show()

# Display LSTM both accuracy and loss
print_plot_history(history)
```





A.1.6 Plot Training and validation history for CNN

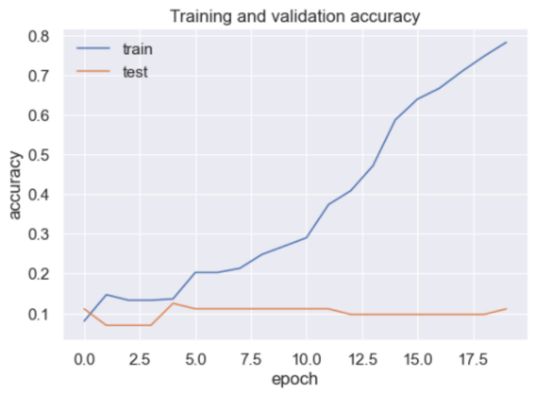
```
history_cnn = model_cnn.fit(X_train, Y_train, batch_size=512, epochs=20, verbose=1, validation_split=0.2)
```

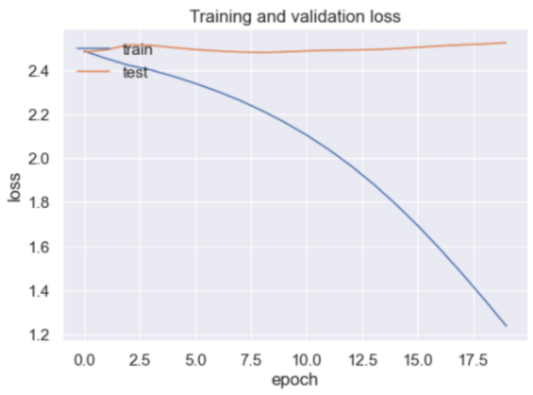
```
Epoch 1/20
1/1 [==============================] - 0s 178ms/step - loss: 2.4856 - accuracy: 0.0804 - val_loss: 2.4815 - val_accuracy: 0.1111
Epoch 2/20
1/1 [==============================] - 0s 27ms/step - loss: 2.4518 - accuracy: 0.1469 - val_loss: 2.4911 - val_accuracy: 0.0694
Epoch 3/20
1/1 [==============================] - 0s 27ms/step - loss: 2.4211 - accuracy: 0.1329 - val_loss: 2.5137 - val_accuracy: 0.0694
Epoch 4/20
1/1 [==============================] - 0s 28ms/step - loss: 2.3999 - accuracy: 0.1329 - val_loss: 2.5117 - val_accuracy: 0.0694
Epoch 5/20
1/1 [==============================] - 0s 35ms/step - loss: 2.3714 - accuracy: 0.1364 - val_loss: 2.5013 - val_accuracy: 0.1250
Epoch 6/20
1/1 [==============================] - 0s 33ms/step - loss: 2.3387 - accuracy: 0.2028 - val_loss: 2.4923 - val_accuracy: 0.1111
Epoch 7/20
1/1 [==============================] - 0s 27ms/step - loss: 2.3029 - accuracy: 0.2028 - val_loss: 2.4858 - val_accuracy: 0.1111
Epoch 8/20
1/1 [==============================] - 0s 33ms/step - loss: 2.2622 - accuracy: 0.2133 - val_loss: 2.4811 - val_accuracy: 0.1111
Epoch 9/20
1/1 [==============================] - 0s 27ms/step - loss: 2.2156 - accuracy: 0.2483 - val_loss: 2.4792 - val_accuracy: 0.1111
Epoch 10/20
1/1 [==============================] - 0s 29ms/step - loss: 2.1636 - accuracy: 0.2692 - val_loss: 2.4817 - val_accuracy: 0.1111
Epoch 11/20
1/1 [==============================] - 0s 31ms/step - loss: 2.1052 - accuracy: 0.2902 - val_loss: 2.4860 - val_accuracy: 0.1111
Epoch 12/20
1/1 [==============================] - 0s 29ms/step - loss: 2.0396 - accuracy: 0.3741 - val_loss: 2.4889 - val_accuracy: 0.1111
Epoch 13/20
1/1 [==============================] - 0s 29ms/step - loss: 1.9659 - accuracy: 0.4091 - val_loss: 2.4899 - val_accuracy: 0.0972
Epoch 14/20
1/1 [==============================] - 0s 26ms/step - loss: 1.8840 - accuracy: 0.4720 - val_loss: 2.4919 - val_accuracy: 0.0972
Epoch 15/20
1/1 [==============================] - 0s 30ms/step - loss: 1.7936 - accuracy: 0.5874 - val_loss: 2.4962 - val_accuracy: 0.0972
Epoch 16/20
1/1 [==============================] - 0s 28ms/step - loss: 1.6952 - accuracy: 0.6399 - val_loss: 2.5023 - val_accuracy: 0.0972
Epoch 17/20
1/1 [==============================] - 0s 29ms/step - loss: 1.5894 - accuracy: 0.6678 - val_loss: 2.5087 - val_accuracy: 0.0972
Epoch 18/20
1/1 [==============================] - 0s 29ms/step - loss: 1.4771 - accuracy: 0.7098 - val_loss: 2.5139 - val_accuracy: 0.0972
Epoch 19/20
1/1 [==============================] - 0s 30ms/step - loss: 1.3599 - accuracy: 0.7483 - val_loss: 2.5176 - val_accuracy: 0.0972
Epoch 20/20
1/1 [==============================] - 0s 26ms/step - loss: 1.2382 - accuracy: 0.7832 - val_loss: 2.5240 - val_accuracy: 0.1111
```

```
score = model_cnn.evaluate(X_test, Y_test, verbose=1)

print("Test Score:", score[0])
print("Test Accuracy:", score[1])
```

```
3/3 [==============================] - 0s 4ms/step - loss: 2.6610 - accuracy: 0.0333
Test Score: 2.660961627960205
Test Accuracy: 0.03333333507180214
```

```
# Display CNN both accuracy and loss
print_plot_history(history_cnn)
```

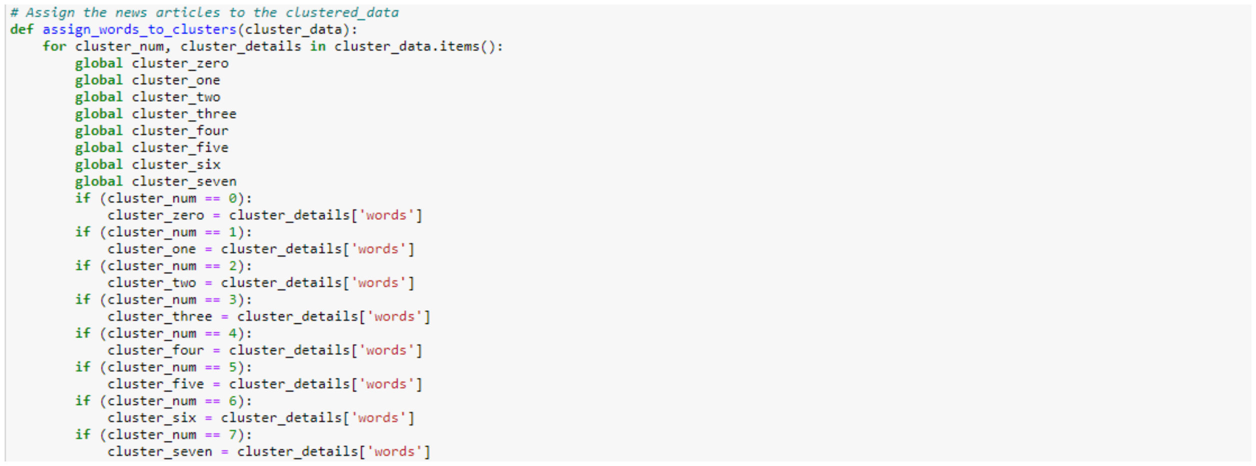




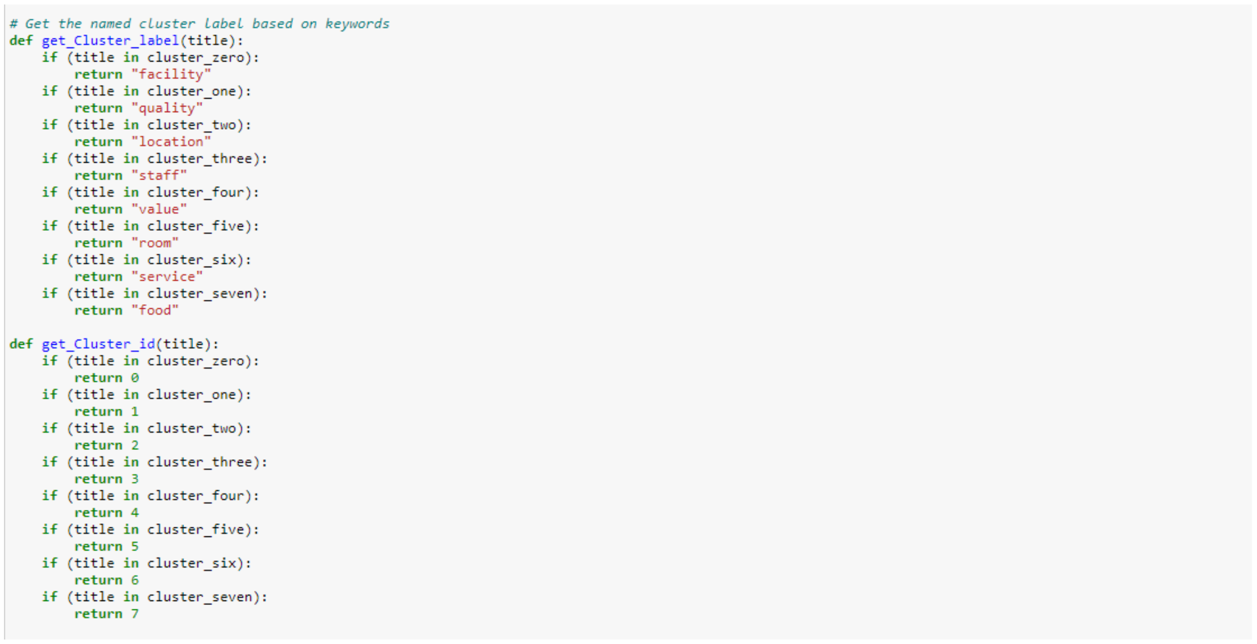
A.1.6. Conclusion

According to the above 2 diagrams we can see in RNN the training and testing accuracies are closer than CNN's. Hence we can identify RNN as the better model.

A.2.1 BoW with CountVectorizer Labelling the Training Model



```python
# Assign the news articles to the clustered_data
def assign_words_to_clusters(cluster_data):
    for cluster_num, cluster_details in cluster_data.items():
        global cluster_zero
        global cluster_one
        global cluster_two
        global cluster_three
        global cluster_four
        global cluster_five
        global cluster_six
        global cluster_seven
        if (cluster_num == 0):
            cluster_zero = cluster_details['words']
        if (cluster_num == 1):
            cluster_one = cluster_details['words']
        if (cluster_num == 2):
            cluster_two = cluster_details['words']
        if (cluster_num == 3):
            cluster_three = cluster_details['words']
        if (cluster_num == 4):
            cluster_four = cluster_details['words']
        if (cluster_num == 5):
            cluster_five = cluster_details['words']
        if (cluster_num == 6):
            cluster_six = cluster_details['words']
        if (cluster_num == 7):
            cluster_seven = cluster_details['words']
```



```python
# Get the named cluster label based on keywords
def get_Cluster_label(title):
    if (title in cluster_zero):
        return "facility"
    if (title in cluster_one):
        return "quality"
    if (title in cluster_two):
        return "location"
    if (title in cluster_three):
        return "staff"
    if (title in cluster_four):
        return "value"
    if (title in cluster_five):
        return "room"
    if (title in cluster_six):
        return "service"
    if (title in cluster_seven):
        return "food"

def get_Cluster_id(title):
    if (title in cluster_zero):
        return 0
    if (title in cluster_one):
        return 1
    if (title in cluster_two):
        return 2
    if (title in cluster_three):
        return 3
    if (title in cluster_four):
        return 4
    if (title in cluster_five):
        return 5
    if (title in cluster_six):
        return 6
    if (title in cluster_seven):
        return 7
```

```
# Write to csv in-order to complete compative study report
def write_to_csv(dataset, path_name):
    csv_record = []
    for i in range(len(dataset)):
        if isinstance(dataset.iloc[i,:9].count_vect_clusters, int) is False and (dataset.iloc[i,:9].count_vect_clusters) > 0:
            text = ""
            for j in range(len(dataset.iloc[i,:9].stop_words_removed_lemmatized)):
                text+=dataset.iloc[i,:9].stop_words_removed_lemmatized[j]+" "
            suggestions_row_data = []
            title = dataset.iloc[i,:9].hotel_name
            body = dataset.iloc[i,:9].reviews
            cleaned_data = text.strip()
            cluster_label = get_Cluster_label(dataset.iloc[i,:9].stop_words_removed_lemmatized)
            cluster_id =get_Cluster_id(dataset.iloc[i,:9].stop_words_removed_lemmatized)
            suggestions_row_data.append(title)
            suggestions_row_data.append(body)
            suggestions_row_data.append(cleaned_data)
            suggestions_row_data.append(cluster_label)
            suggestions_row_data.append(cluster_id)
            csv_record.append(suggestions_row_data)
    suggestionsbowDf = pd.DataFrame(csv_record, columns = ['Title', 'Body', 'Cleaned Data', 'Cluster','cluster_id'])
    suggestionsbowDf.to_csv(path_name, sep=',', encoding='utf-8')


# Assign cleaned data set as 3rd column in the corpus
suggestion_df['count_vect_clusters'] = count_vect_clusters
# # Identify the the cluster list
assign_words_to_clusters(cluster_data)

write_to_csv(suggestion_df, './clustered_data_bow.csv')
```

A.2.2 Test and Train Data for NN

```
import pandas as pd
import numpy as np
import nltk

# This data is the processed data written after doing the first part of the course work
textDf8 = pd.read_csv('./clustered_data_bow.csv')
textDf8.isnull().values.any()

#split data to X and Y(labels)
textDf_dataset8 = textDf8['Cleaned Data']
textDf_labels8 = textDf8['Cluster']

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

# split in to a test set
label_encoder = LabelEncoder()
text8_X_train, text8_X_test, text8_Y_train, text8_Y_test \
= train_test_split(textDf_dataset8, label_encoder.fit_transform(textDf_labels8), test_size=0.2, random_state=1)
```

A.2.3 Create LSTM Model using RNN

```
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences

# This will be the maximum number of words from our resulting tokenized data vocabulary which are to be used,
#truncated after the 1000 most common words in our case
num_words = 1000
# our maximum sentence length will be determined by searching our sentences for the one of maximum length,
# and padding characters will be '0'
pad_type = 'post'

# Tokenize our training data
tokenizer = Tokenizer(num_words=num_words)
tokenizer.fit_on_texts(text8_X_train)

# Get our training data word index
word_index = tokenizer.word_index

# Encode training data sentences into sequences
train8_sequences = tokenizer.texts_to_sequences(text8_X_train)
test8_sequences = tokenizer.texts_to_sequences(text8_X_test)

# Get max training sequence length
maxlen = max([len(x) for x in train8_sequences])

# Pad the training sequences (pad_sequece makes the sentences in input dataset uniform in length)
X8_train = pad_sequences(train8_sequences, padding=pad_type,maxlen=maxlen)
X8_test = pad_sequences(test8_sequences, padding=pad_type,maxlen=maxlen)

Y8_train = to_categorical(text8_Y_train)
Y8_test = to_categorical(text8_Y_test)

vocabSize8 = len(tokenizer.word_index) + 1  # Adding 1 because of reserved 0 index

# Output the results of our work
print("\nVocab size:\n", vocabSize8)
print("\nPadded training sequences:\n", X8_train[0, :10])
print("\nPadded training shape:", X8_train.shape)
print("Training sequences data type:", type(train8_sequences))
print("Padded Training sequences data type:", type(X8_train))
```

```
Vocab size:
 1254

Padded training sequences:
 [ 78  40 436 437 438 187 439   2 440 441]

Padded training shape: (357, 31)
Training sequences data type: <class 'list'>
Padded Training sequences data type: <class 'numpy.ndarray'>
```

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers
from tensorflow.keras.layers import SpatialDropout1D, LSTM, Dense, InputLayer

embeddingDim = 100

model8 = Sequential()
# The first layer is the embedded layer that uses max length vectors to represent each word.
model8.add(layers.Embedding(input_dim=vocabSize8,
                            output_dim=embeddingDim,
                            input_length=maxlen))
model8.add(SpatialDropout1D(0.2))
# The next layer is the LSTM layer with 256 memory units.
model8.add(layers.LSTM(256, dropout=0.2, recurrent_dropout=0.2))
# The output layer must create 8 output values, one for each class.
model8.add(Dense(7, activation='softmax'))
# Activation function is softmax for multi-class classification.
model8.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
print(model8.summary())
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_2 (Embedding)      (None, 31, 100)           125400    
_________________________________________________________________
spatial_dropout1d_1 (Spatial (None, 31, 100)           0         
_________________________________________________________________
lstm_1 (LSTM)                (None, 256)               365568    
_________________________________________________________________
dense_3 (Dense)              (None, 7)                 1799      
=================================================================
Total params: 492,767
Trainable params: 492,767
Non-trainable params: 0
_________________________________________________________________
```

A.2.4 Create CNN Model

```
# Setup a Convulation Neural Network (CNN)
model8_cnn = Sequential()
model8_cnn.add(layers.Embedding(vocabSize, 100, input_length=maxlen))

model8_cnn.add(layers.Conv1D(filters=512, kernel_size=3, padding='same', activation='relu'))
model8_cnn.add(layers.MaxPooling1D(pool_size=2))
model8_cnn.add(layers.Flatten())
model8_cnn.add(layers.Dense(250, activation='relu'))

model8_cnn.add(layers.Dense(7, activation='softmax'))
model8_cnn.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model8_cnn.summary()
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_3 (Embedding)      (None, 31, 100)           123200    
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 31, 512)           154112    
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 15, 512)           0         
_________________________________________________________________
flatten_1 (Flatten)          (None, 7680)              0         
_________________________________________________________________
dense_4 (Dense)              (None, 250)               1920250   
_________________________________________________________________
dense_5 (Dense)              (None, 7)                 1757      
=================================================================
Total params: 2,199,319
Trainable params: 2,199,319
Non-trainable params: 0
_________________________________________________________________
```

A.2.5 Plot Training and validation history for LSTM RNN

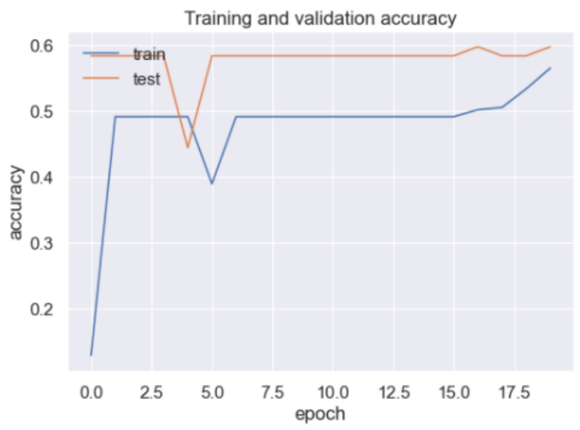
```
history8 = model8.fit(X8_train, Y8_train, batch_size=256, epochs=20, verbose=1, validation_split=0.2)
```

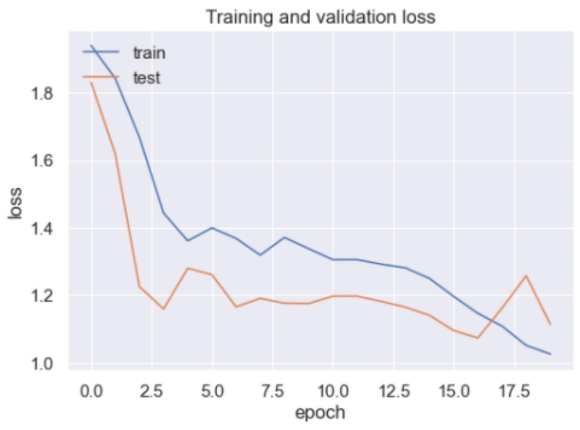
```
Epoch 1/20
2/2 [==============================] - 0s 249ms/step - loss: 1.9407 - accuracy: 0.1298 - val_loss: 1.8310 - val_accuracy: 0.5833
Epoch 2/20
2/2 [==============================] - 0s 98ms/step - loss: 1.8427 - accuracy: 0.4912 - val_loss: 1.6189 - val_accuracy: 0.5833
Epoch 3/20
2/2 [==============================] - 0s 135ms/step - loss: 1.6683 - accuracy: 0.4912 - val_loss: 1.2249 - val_accuracy: 0.5833
Epoch 4/20
2/2 [==============================] - 0s 138ms/step - loss: 1.4434 - accuracy: 0.4912 - val_loss: 1.1587 - val_accuracy: 0.5833
Epoch 5/20
2/2 [==============================] - 0s 122ms/step - loss: 1.3609 - accuracy: 0.4912 - val_loss: 1.2794 - val_accuracy: 0.4444
Epoch 6/20
2/2 [==============================] - 0s 95ms/step - loss: 1.3994 - accuracy: 0.3895 - val_loss: 1.2605 - val_accuracy: 0.5833
Epoch 7/20
2/2 [==============================] - 0s 93ms/step - loss: 1.3679 - accuracy: 0.4912 - val_loss: 1.1650 - val_accuracy: 0.5833
Epoch 8/20
2/2 [==============================] - 0s 132ms/step - loss: 1.3187 - accuracy: 0.4912 - val_loss: 1.1902 - val_accuracy: 0.5833
Epoch 9/20
2/2 [==============================] - 0s 127ms/step - loss: 1.3709 - accuracy: 0.4912 - val_loss: 1.1757 - val_accuracy: 0.5833
Epoch 10/20
2/2 [==============================] - 0s 95ms/step - loss: 1.3376 - accuracy: 0.4912 - val_loss: 1.1750 - val_accuracy: 0.5833
Epoch 11/20
2/2 [==============================] - 0s 99ms/step - loss: 1.3052 - accuracy: 0.4912 - val_loss: 1.1966 - val_accuracy: 0.5833
Epoch 12/20
2/2 [==============================] - 0s 119ms/step - loss: 1.3051 - accuracy: 0.4912 - val_loss: 1.1969 - val_accuracy: 0.5833
Epoch 13/20
2/2 [==============================] - 0s 96ms/step - loss: 1.2917 - accuracy: 0.4912 - val_loss: 1.1812 - val_accuracy: 0.5833
Epoch 14/20
2/2 [==============================] - 0s 105ms/step - loss: 1.2808 - accuracy: 0.4912 - val_loss: 1.1642 - val_accuracy: 0.5833
Epoch 15/20
2/2 [==============================] - 0s 111ms/step - loss: 1.2493 - accuracy: 0.4912 - val_loss: 1.1400 - val_accuracy: 0.5833
Epoch 16/20
2/2 [==============================] - 0s 116ms/step - loss: 1.1959 - accuracy: 0.4912 - val_loss: 1.0948 - val_accuracy: 0.5833
Epoch 17/20
2/2 [==============================] - 0s 102ms/step - loss: 1.1458 - accuracy: 0.5018 - val_loss: 1.0726 - val_accuracy: 0.5972
Epoch 18/20
2/2 [==============================] - 0s 96ms/step - loss: 1.1080 - accuracy: 0.5053 - val_loss: 1.1612 - val_accuracy: 0.5833
Epoch 19/20
2/2 [==============================] - 0s 123ms/step - loss: 1.0509 - accuracy: 0.5333 - val_loss: 1.2570 - val_accuracy: 0.5833
Epoch 20/20
2/2 [==============================] - 0s 98ms/step - loss: 1.0248 - accuracy: 0.5649 - val_loss: 1.1133 - val_accuracy: 0.5972
```

```
# Below method will be printing the test score and accurecy of the system
score8 = model8.evaluate(X8_test, Y8_test, verbose=1)
print("Test Score:", score8[0])
print("Test Accuracy:", score8[1])
```

```
3/3 [==============================] - 0s 16ms/step - loss: 1.1361 - accuracy: 0.6111
Test Score: 1.1360830068588257
Test Accuracy: 0.6111111044883728
```

```
# Display LSTM both accuracy and loss
print_plot_history(history8)
```





A.2.6 Plot Training and validation history for CNN

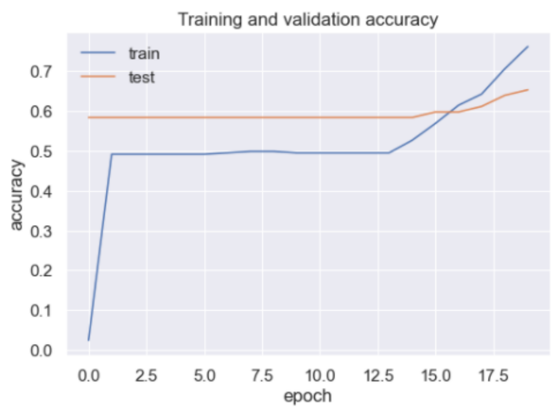
```
history8_cnn = model8_cnn.fit(X8_train, Y8_train, batch_size=512, epochs=20, verbose=1, validation_split=0.2)
```

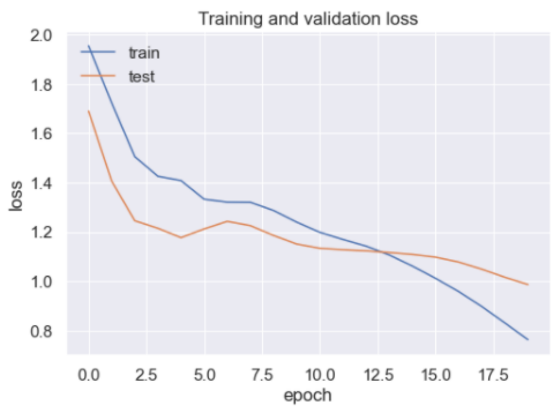
```
Epoch 1/20
1/1 [==============================] - 0s 184ms/step - loss: 1.9544 - accuracy: 0.0246 - val_loss: 1.6897 - val_accuracy: 0.5833
Epoch 2/20
1/1 [==============================] - 0s 33ms/step - loss: 1.7233 - accuracy: 0.4912 - val_loss: 1.4063 - val_accuracy: 0.5833
Epoch 3/20
1/1 [==============================] - 0s 33ms/step - loss: 1.5051 - accuracy: 0.4912 - val_loss: 1.2450 - val_accuracy: 0.5833
Epoch 4/20
1/1 [==============================] - 0s 26ms/step - loss: 1.4254 - accuracy: 0.4912 - val_loss: 1.2144 - val_accuracy: 0.5833
Epoch 5/20
1/1 [==============================] - 0s 31ms/step - loss: 1.4078 - accuracy: 0.4912 - val_loss: 1.1764 - val_accuracy: 0.5833
Epoch 6/20
1/1 [==============================] - 0s 37ms/step - loss: 1.3324 - accuracy: 0.4912 - val_loss: 1.2118 - val_accuracy: 0.5833
Epoch 7/20
1/1 [==============================] - 0s 29ms/step - loss: 1.3205 - accuracy: 0.4947 - val_loss: 1.2425 - val_accuracy: 0.5833
Epoch 8/20
1/1 [==============================] - 0s 29ms/step - loss: 1.3198 - accuracy: 0.4982 - val_loss: 1.2250 - val_accuracy: 0.5833
Epoch 9/20
1/1 [==============================] - 0s 29ms/step - loss: 1.2867 - accuracy: 0.4982 - val_loss: 1.1853 - val_accuracy: 0.5833
Epoch 10/20
1/1 [==============================] - 0s 30ms/step - loss: 1.2394 - accuracy: 0.4947 - val_loss: 1.1506 - val_accuracy: 0.5833
Epoch 11/20
1/1 [==============================] - 0s 32ms/step - loss: 1.1982 - accuracy: 0.4947 - val_loss: 1.1333 - val_accuracy: 0.5833
Epoch 12/20
1/1 [==============================] - 0s 31ms/step - loss: 1.1691 - accuracy: 0.4947 - val_loss: 1.1275 - val_accuracy: 0.5833
Epoch 13/20
1/1 [==============================] - 0s 31ms/step - loss: 1.1422 - accuracy: 0.4947 - val_loss: 1.1228 - val_accuracy: 0.5833
Epoch 14/20
1/1 [==============================] - 0s 38ms/step - loss: 1.1064 - accuracy: 0.4947 - val_loss: 1.1162 - val_accuracy: 0.5833
Epoch 15/20
1/1 [==============================] - 0s 40ms/step - loss: 1.0615 - accuracy: 0.5263 - val_loss: 1.1085 - val_accuracy: 0.5833
Epoch 16/20
1/1 [==============================] - 0s 31ms/step - loss: 1.0120 - accuracy: 0.5684 - val_loss: 1.0974 - val_accuracy: 0.5972
Epoch 17/20
1/1 [==============================] - 0s 32ms/step - loss: 0.9583 - accuracy: 0.6140 - val_loss: 1.0777 - val_accuracy: 0.5972
Epoch 18/20
1/1 [==============================] - 0s 31ms/step - loss: 0.8977 - accuracy: 0.6421 - val_loss: 1.0487 - val_accuracy: 0.6111
Epoch 19/20
1/1 [==============================] - 0s 27ms/step - loss: 0.8310 - accuracy: 0.7053 - val_loss: 1.0161 - val_accuracy: 0.6389
Epoch 20/20
1/1 [==============================] - 0s 28ms/step - loss: 0.7630 - accuracy: 0.7614 - val_loss: 0.9864 - val_accuracy: 0.6528
```

```
score8 = model8_cnn.evaluate(X8_test, Y8_test, verbose=1)

print("Test Score:", score8[0])
print("Test Accuracy:", score8[1])
```

```
3/3 [==============================] - 0s 5ms/step - loss: 1.0154 - accuracy: 0.6778
Test Score: 1.015368938446045
Test Accuracy: 0.6777777671813965
```

```
print_plot_history(history8_cnn)
```





A.2.7. Conclusion

For the 8 clusters one RNN is having accuracy around 61% while CNN has an accuracy of 67% so CNN is better with the 8 clustering