Configuration Manual

MSc Research Project
MSc in Data Analytics

Kruthika Surendrakumar
Student ID: X22241965

School of Computing
National College of Ireland

Supervisor : Abdul Shahid

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Kruthika Surendrakumar		
Student ID:	X22241965		
Programme:	MSc In Data Analytics	**Year:**	2024-2025
Module:	MSc In Data Analytics		
Lecturer:	Abdul Shahid		
Submission Due Date:	12/12/2024		
Project Title:	Abstractive Summarization Using Neural Networks with Attention Mechanisms		
Word Count:	1,691	**Page Count:**	15

Signature:	Kruthika Surendrakumar
Date:	12/12/2024

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Kruthika Surendrakumar
Student ID: X2241965

1. Introduction

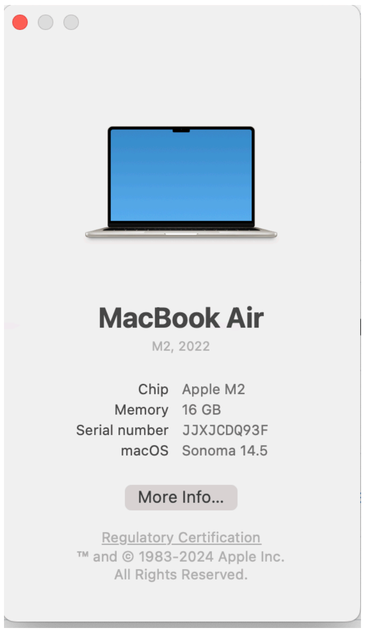
The main idea of this manual is to make the users able to replicate the code in their own environment. It explains the hardware specification, specification, some essential libraries that are mandatory to run the code, source of the dataset, data preprocessed, model building, training and evaluation. A few of the code snippets are accompanied with some images.

2. Environment

The environment should be setup in first place to run the code, this section helps to set the environment for the successful replication of the code. This research is carried out in the google colab pro environment as it provides access to computing higher resources like CPU, A100 GPU, L4 GPU, T4 GPU and TPU v2-8.

3. System Configuration

The computational task in this research was performed using Apple Macbook Air hardware. Table 1 describes the system, giving detailed specification of the hardware in the system.



MacBook Air
M2, 2022

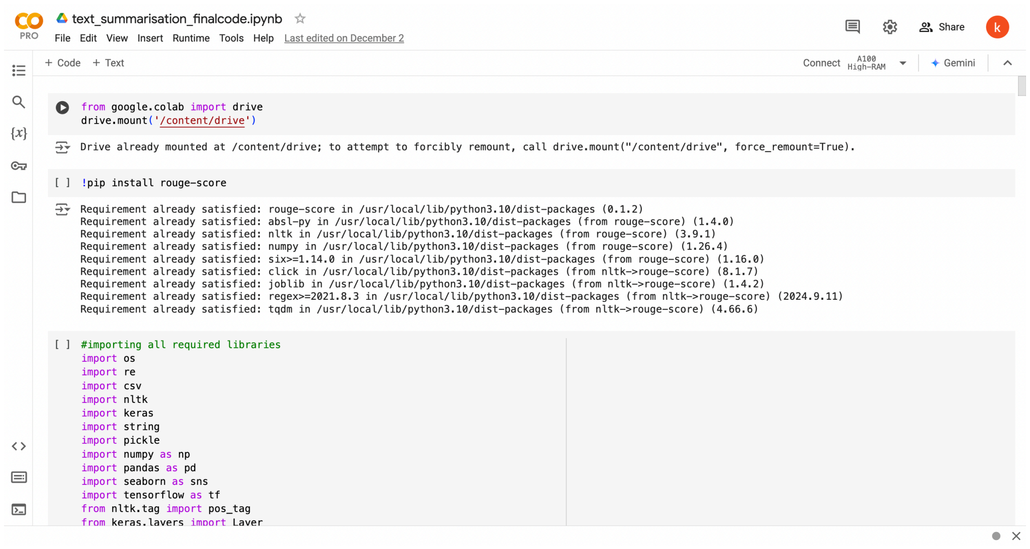
Chip	Apple M2
Memory	16 GB
Serial number	JJXJCDQ93F
macOS	Sonoma 14.5

™ and © 1983-2024 Apple Inc.
All Rights Reserved.

Table 1 : Hardware Specification

4. Setting Up The Google Colab Pro Environment

My research project uses Python as the main programming language and Google Colab Pro to run the code, as you can see in figure 1. However, if you want a cloud based Jupyter notebook environment to run machine learning and deep learning models with very minimal setup, you can use Google Colab. This platform provides high performance GPU access (e.g., to NVIDIA A100 cards) in the Pro version. And with a lot of the computations taking place much more quickly thanks to the A100 GPU, it's perfectly suited to training large models and handling complicated problems. As compared to other platforms, for example, Kaggle, Colab Pro is a premium service that carries with it better resource availability and capabilities to help with smooth execution of machine learning workflows.


text_summarisation_finalcode.ipynb
File Edit View Insert Runtime Tools Help Last edited on December 2
Share
+ Code + Text
Connect A100 High-RAM
Gemini

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

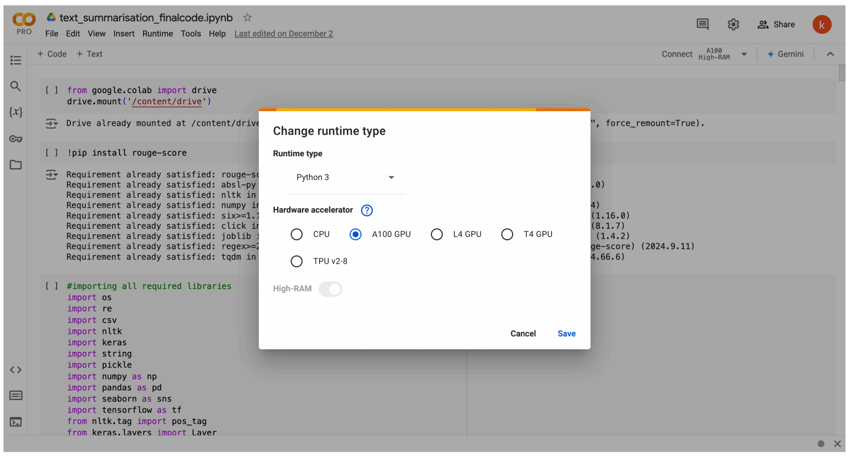
```
!pip install rouge-score
```

```
Requirement already satisfied: rouge-score in /usr/local/lib/python3.10/dist-packages (0.1.2)
Requirement already satisfied: absl-py in /usr/local/lib/python3.10/dist-packages (from rouge-score) (1.4.0)
Requirement already satisfied: nltk in /usr/local/lib/python3.10/dist-packages (from rouge-score) (3.9.1)
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-packages (from rouge-score) (1.26.4)
Requirement already satisfied: six>=1.14.0 in /usr/local/lib/python3.10/dist-packages (from rouge-score) (1.16.0)
Requirement already satisfied: click in /usr/local/lib/python3.10/dist-packages (from nltk->rouge-score) (8.1.7)
Requirement already satisfied: joblib in /usr/local/lib/python3.10/dist-packages (from nltk->rouge-score) (1.4.2)
Requirement already satisfied: regex>=2021.8.3 in /usr/local/lib/python3.10/dist-packages (from nltk->rouge-score) (2024.9.11)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from nltk->rouge-score) (4.66.6)
```

```
#importing all required libraries
import os
import re
import csv
import nltk
import keras
import string
import pickle
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
from nltk.tag import pos_tag
from keras.layers import Layer
```

Figure 1 : Google Colab Pro set up

To use the google colab version , you can simply go to the top right corner and accept the necessary terms and conditions and buy the pro version. This is how your pro environment looks like in figure 2.


text_summarisation_finalcode.ipynb
File Edit View Insert Runtime Tools Help Last edited on December 2
Share
+ Code + Text
Connect A100 High-RAM
Gemini
Cancel
Save

Change runtime type

Runtime type: Python 3

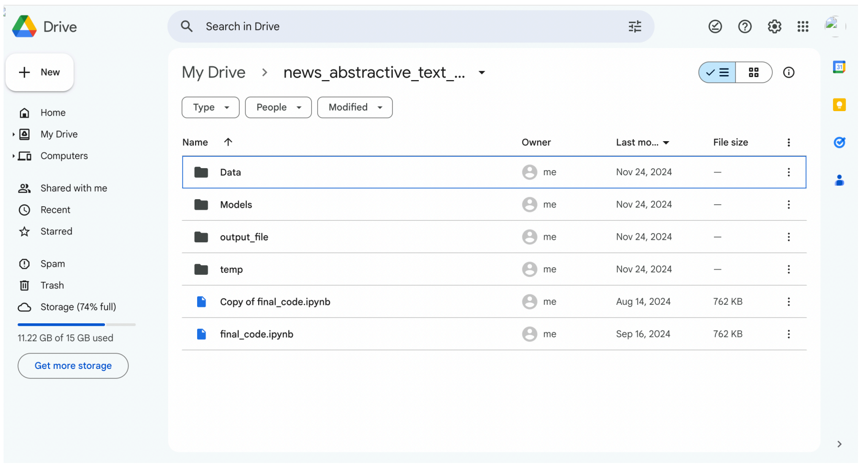
Hardware accelerator: CPU [✓ A100 GPU] L4 GPU T4 GPU TPU v2-8

High-RAM

Figure 2 : Google colab Pro

5. Connecting to the Google Drive

This research has been entirely connected to the google drive as shown in the figure 3 .



Name	Owner	Last mo...	File size
Data	me	Nov 24, 2024	—
Models	me	Nov 24, 2024	—
output_file	me	Nov 24, 2024	—
temp	me	Nov 24, 2024	—
Copy of final_code.ipynb	me	Aug 14, 2024	762 KB
final_code.ipynb	me	Sep 16, 2024	762 KB

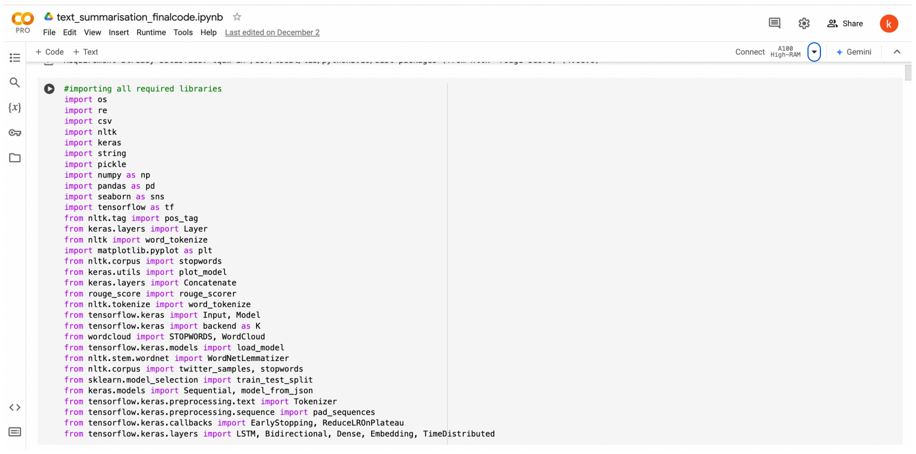
Figure 3 : Google Drive environment

6. Implementation

Once we finish environmental setup and successfully connect with google drive, we can move onto the implementation part. The part under Implementation guides from importing the dataset to required libraries and frameworks and then jump onto model building, training and evaluation.

6.1 Importing Libraries

Significant number of libraries are used for preprocessing, training the model, evaluation and visualisation of the results of the research. In first place, the libraries need to be imported. Therefore import the libraries as named in the beginning part of the code as shown in the Figure 4.

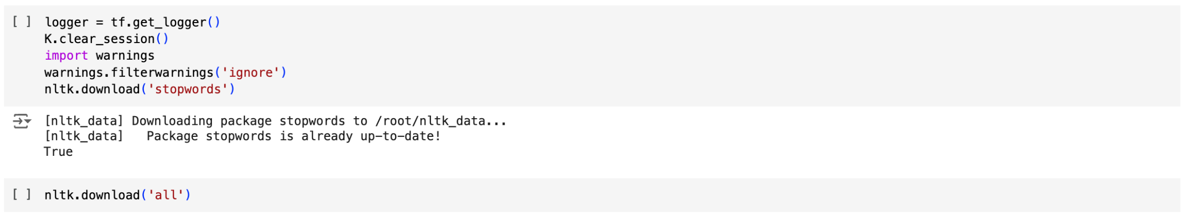

text_summarisation_finalcode.ipynb
File Edit View Insert Runtime Tools Help Last edited on December 2
Share
+ Code + Text
Connect A100 High-RAM
Gemini

```
#importing all required libraries
import os
import re
import csv
import nltk
import keras
import string
import pickle
import numpy as np
import pandas as pd
import seaborn as sns
import tensorflow as tf
from nltk.tag import pos_tag
from keras.layers import Layer
from nltk import word_tokenize
import matplotlib.pyplot as plt
from nltk.corpus import stopwords
from keras.utils import plot_model
from keras.layers import Concatenate
from rouge_score import rouge_scorer
from nltk.tokenize import word_tokenize
from tensorflow.keras import Input, Model
from tensorflow.keras import backend as K
from wordcloud import STOPWORDS, WordCloud
from tensorflow.keras.models import load_model
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.corpus import twitter_samples, stopwords
from sklearn.model_selection import train_test_split
from keras.models import Sequential, model_from_json
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
from tensorflow.keras.layers import LSTM, Bidirectional, Dense, Embedding, TimeDistributed
```

Figure 4 : Necessary Libraries

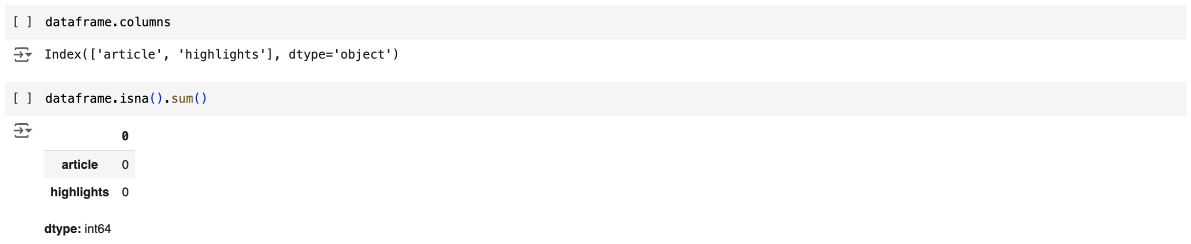
6.2 Data Preparartion

Articles and their corresponding highlights dataset is loaded and the unnecessary columns are dropped. Lowercasing, expanding contractions, remove special characters, URLs and optionally stopword removal is text preprocessing. Two new sentence types have been introduced which we use to highlight <sostok> and <eostok> tokens for the start and end of summaries respectively. We clean the dataset by removing null or empty summaries. Histograms of summary and text lengths are used to analyze the distribution of the length. All of these steps has been shown in Figure 5,6,7,8.



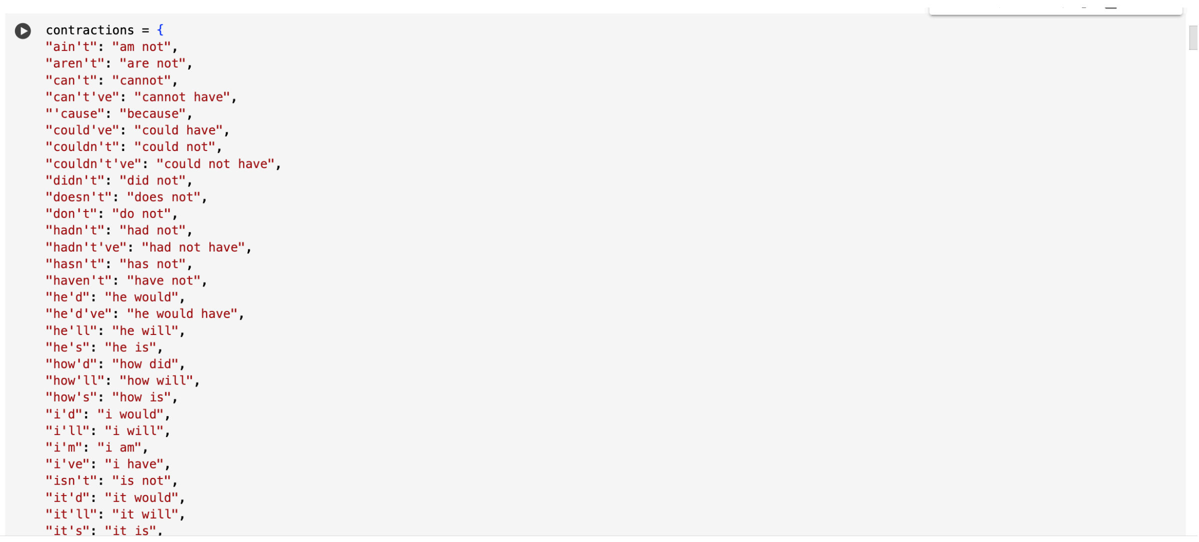
```
[ ] logger = tf.get_logger()
    K.clear_session()
    import warnings
    warnings.filterwarnings('ignore')
    nltk.download('stopwords')

    [nltk_data] Downloading package stopwords to /root/nltk_data...
    [nltk_data]   Package stopwords is already up-to-date!
    True

[ ] nltk.download('all')
```

Figure 5 : Downloaded necessary vocabularies.

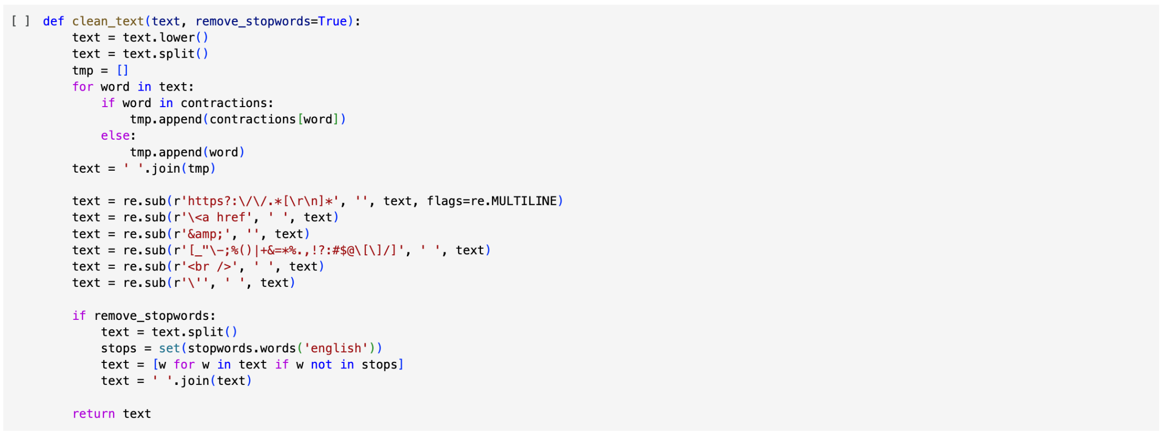


```
[ ] dataframe.columns

    Index(['article', 'highlights'], dtype='object')

[ ] dataframe.isna().sum()

                0
    article     0
    highlights  0

    dtype: int64
```

Figure 6 :Checked Null values

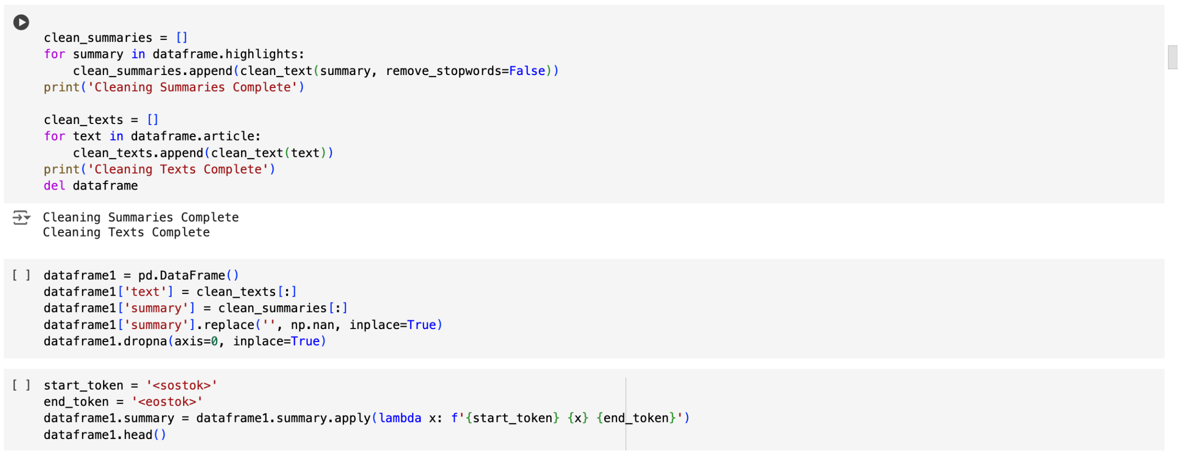


```
contractions = {
"ain't": "am not",
"aren't": "are not",
"can't": "cannot",
"can't've": "cannot have",
"'cause": "because",
"could've": "could have",
"couldn't": "could not",
"couldn't've": "could not have",
"didn't": "did not",
"doesn't": "does not",
"don't": "do not",
"hadn't": "had not",
"hadn't've": "had not have",
"hasn't": "has not",
"haven't": "have not",
"he'd": "he would",
"he'd've": "he would have",
"he'll": "he will",
"he's": "he is",
"how'd": "how did",
"how'll": "how will",
"how's": "how is",
"i'd": "i would",
"i'll": "i will",
"i'm": "i am",
"i've": "i have",
"isn't": "is not",
"it'd": "it would",
"it'll": "it will",
"it's": "it is",
```



```
[ ] def clean_text(text, remove_stopwords=True):
        text = text.lower()
        text = text.split()
        tmp = []
        for word in text:
            if word in contractions:
                tmp.append(contractions[word])
            else:
                tmp.append(word)
        text = ' '.join(tmp)

        text = re.sub(r'https?:\/\/.*[\r\n]*', '', text, flags=re.MULTILINE)
        text = re.sub(r'\<a href', ' ', text)
        text = re.sub(r'&amp;', '', text)
        text = re.sub(r'[_"\-;%()|+&=*%.,!?:#$@\[\]/]', ' ', text)
        text = re.sub(r'<br />', ' ', text)
        text = re.sub(r'\'', ' ', text)

        if remove_stopwords:
            text = text.split()
            stops = set(stopwords.words('english'))
            text = [w for w in text if w not in stops]
            text = ' '.join(text)

        return text
```

Figure 7 : Contractions Removal



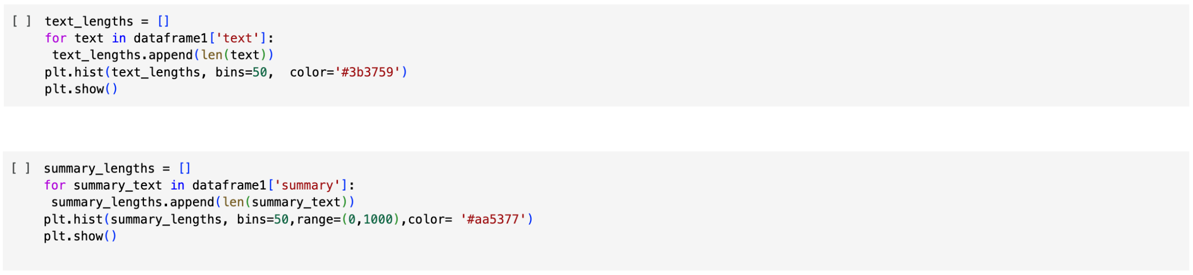
```
clean_summaries = []
for summary in dataframe.highlights:
    clean_summaries.append(clean_text(summary, remove_stopwords=False))
print('Cleaning Summaries Complete')

clean_texts = []
for text in dataframe.article:
    clean_texts.append(clean_text(text))
print('Cleaning Texts Complete')
del dataframe
```

```
Cleaning Summaries Complete
Cleaning Texts Complete
```

```
[ ] dataframe1 = pd.DataFrame()
    dataframe1['text'] = clean_texts[:]
    dataframe1['summary'] = clean_summaries[:]
    dataframe1['summary'].replace('', np.nan, inplace=True)
    dataframe1.dropna(axis=0, inplace=True)
```

```
[ ] start_token = '<sostok>'
    end_token = '<eostok>'
    dataframe1.summary = dataframe1.summary.apply(lambda x: f'{start_token} {x} {end_token}')
    dataframe1.head()
```

Figure 8 : Cleaning the text and start and end tokens denoted



```
[ ] text_lengths = []
    for text in dataframe1['text']:
     text_lengths.append(len(text))
    plt.hist(text_lengths, bins=50,  color='#3b3759')
    plt.show()
```

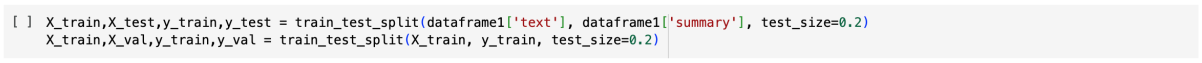
```
[ ] summary_lengths = []
    for summary_text in dataframe1['summary']:
     summary_lengths.append(len(summary_text))
    plt.hist(summary_lengths, bins=50,range=(0,1000),color= '#aa5377')
    plt.show()
```

Figure 9 : Text visualization Code

After the successful preprocessing of the data, the model training starts further.

6.3 Data Splitting

The data has been splitted into training, validation and testing data as shown in figure 10.

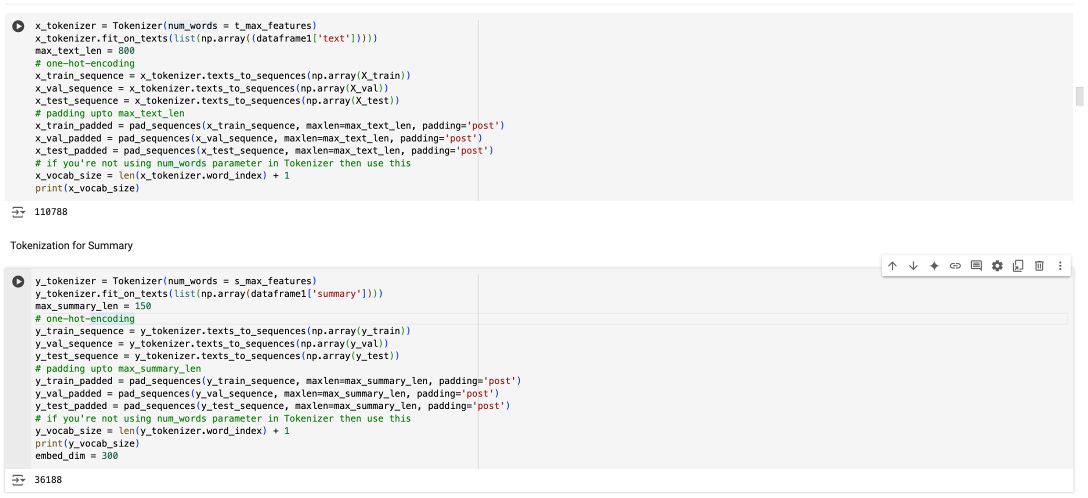


```
[ ] X_train,X_test,y_train,y_test = train_test_split(dataframe1['text'], dataframe1['summary'], test_size=0.2)
    X_train,X_val,y_train,y_val = train_test_split(X_train, y_train, test_size=0.2)
```

Figure 10 : Data Splitting

6.4 Tokenization

In this code, text and summary data is preprocessed for a text summarization model by tokenizing words into numerical indices and padding sequences each to have uniform lengths. The text and the summary are applied for separate tokenization and their vocabulary sizes are truncated with t_max_features and s_max_features respectively. The maximum length for text sequences is padded up to 800 and for summary sequences up to 150. The vocabulary sizes are calculated and the processed data can be used for embedding and training a model with embedding dimension of 300. This has been shown in figure 11.



```
x_tokenizer = Tokenizer(num_words = t_max_features)
x_tokenizer.fit_on_texts(list(np.array((dataframe1['text']))))
max_text_len = 800
# one-hot-encoding
x_train_sequence = x_tokenizer.texts_to_sequences(np.array(X_train))
x_val_sequence = x_tokenizer.texts_to_sequences(np.array(X_val))
x_test_sequence = x_tokenizer.texts_to_sequences(np.array(X_test))
# padding upto max_text_len
x_train_padded = pad_sequences(x_train_sequence, maxlen=max_text_len, padding='post')
x_val_padded = pad_sequences(x_val_sequence, maxlen=max_text_len, padding='post')
x_test_padded = pad_sequences(x_test_sequence, maxlen=max_text_len, padding='post')
# if you're not using num_words parameter in Tokenizer then use this
x_vocab_size = len(x_tokenizer.word_index) + 1
print(x_vocab_size)
```

```
110788
```

Tokenization for Summary

```
y_tokenizer = Tokenizer(num_words = s_max_features)
y_tokenizer.fit_on_texts(list(np.array(dataframe1['summary'])))
max_summary_len = 150
# one-hot-encoding
y_train_sequence = y_tokenizer.texts_to_sequences(np.array(y_train))
y_val_sequence = y_tokenizer.texts_to_sequences(np.array(y_val))
y_test_sequence = y_tokenizer.texts_to_sequences(np.array(y_test))
# padding upto max_summary_len
y_train_padded = pad_sequences(y_train_sequence, maxlen=max_summary_len, padding='post')
y_val_padded = pad_sequences(y_val_sequence, maxlen=max_summary_len, padding='post')
y_test_padded = pad_sequences(y_test_sequence, maxlen=max_summary_len, padding='post')
# if you're not using num_words parameter in Tokenizer then use this
y_vocab_size = len(y_tokenizer.word_index) + 1
print(y_vocab_size)
embed_dim = 300
```

```
36188
```

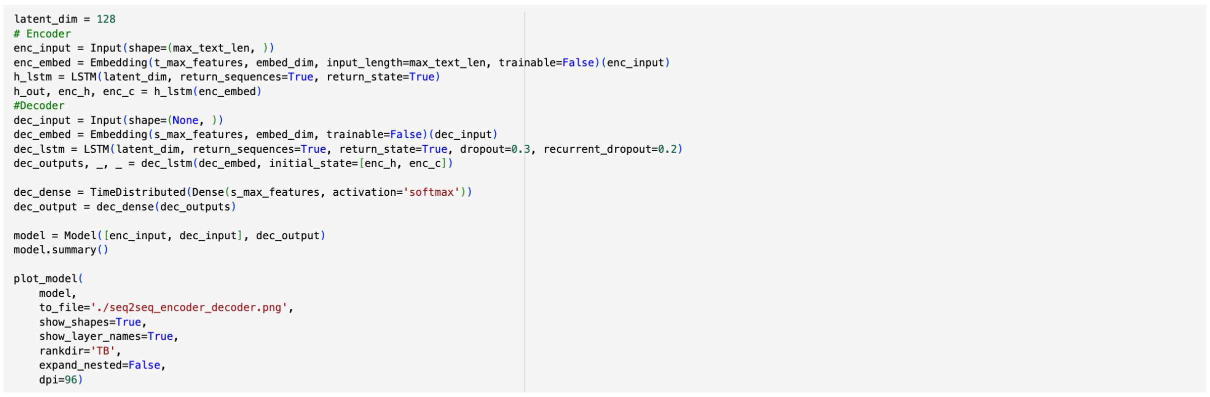
Figure 11 : Tokenization for the text

7. Model Implementation

In this session, we will be discussing about 4 models for the abstractive text summarizatiopn project.

7.1 Study 1 : LSTM without attention layer

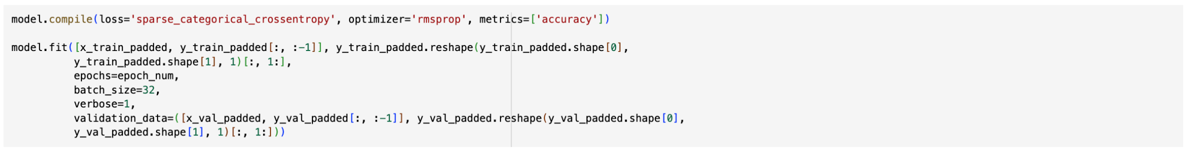
The model consists of an encoder LSTM which operates on embedded text sequences, a decoder LSTM, which produces summaries conditioned on encoder states, and a softmaxbased dense layer for predicting next words show in figure 12.



```
latent_dim = 128
# Encoder
enc_input = Input(shape=(max_text_len, ))
enc_embed = Embedding(t_max_features, embed_dim, input_length=max_text_len, trainable=False)(enc_input)
h_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
h_out, enc_h, enc_c = h_lstm(enc_embed)
#Decoder
dec_input = Input(shape=(None, ))
dec_embed = Embedding(s_max_features, embed_dim, trainable=False)(dec_input)
dec_lstm = LSTM(latent_dim, return_sequences=True, return_state=True, dropout=0.3, recurrent_dropout=0.2)
dec_outputs, _, _ = dec_lstm(dec_embed, initial_state=[enc_h, enc_c])

dec_dense = TimeDistributed(Dense(s_max_features, activation='softmax'))
dec_output = dec_dense(dec_outputs)

model = Model([enc_input, dec_input], dec_output)
model.summary()

plot_model(
    model,
    to_file='./seq2seq_encoder_decoder.png',
    show_shapes=True,
    show_layer_names=True,
    rankdir='TB',
    expand_nested=False,
    dpi=96)
```

Figure 12 : Model Architecture

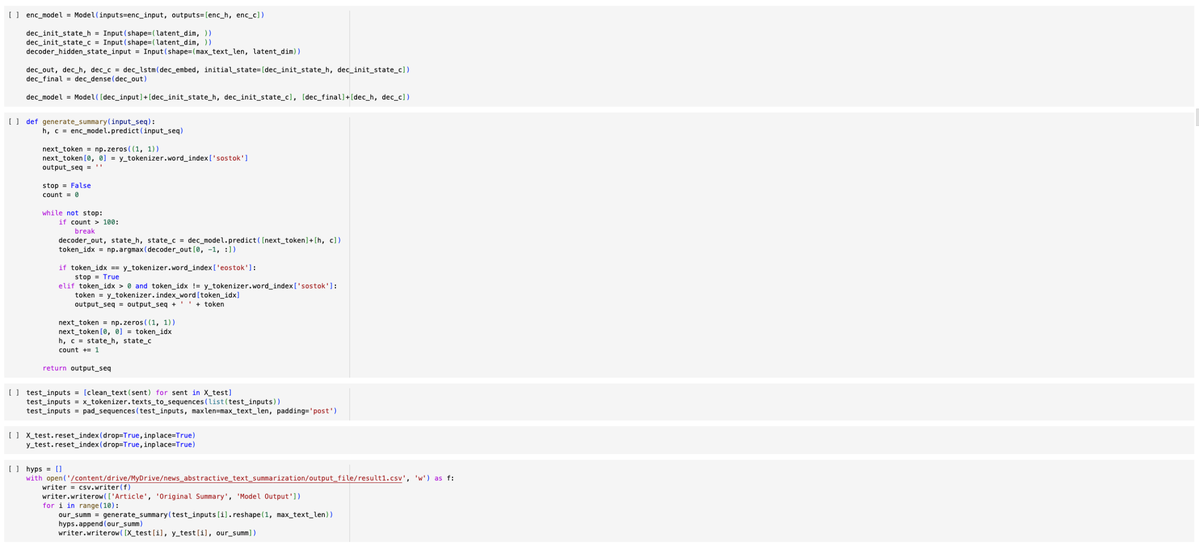
Our model is compiled as using sparse categorical crossentropy loss and is trained on padded text and summary sequences, with a validation data, batch size 32 and specified epochs is shown in figure 13.



```
model.compile(loss='sparse_categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])

model.fit([x_train_padded, y_train_padded[:, :-1]], y_train_padded.reshape(y_train_padded.shape[0],
          y_train_padded.shape[1], 1)[:, 1:],
          epochs=epoch_num,
          batch_size=32,
          verbose=1,
          validation_data=([x_val_padded, y_val_padded[:, :-1]], y_val_padded.reshape(y_val_padded.shape[0],
          y_val_padded.shape[1], 1)[:, 1:]))
```

Figure 13: Training Configuration

A prediction way is created where we separate encoder and decoder models and encoder gives hidden states and decoder creates summaries step by step using predicted tokens is shown in figure 14.



```
[ ] enc_model = Model(inputs=enc_input, outputs=[enc_h, enc_c])

    dec_init_state_h = Input(shape=(latent_dim, ))
    dec_init_state_c = Input(shape=(latent_dim, ))
    decoder_hidden_state_input = Input(shape=(max_text_len, latent_dim))

    dec_out, dec_h, dec_c = dec_lstm(dec_embed, initial_state=[dec_init_state_h, dec_init_state_c])
    dec_final = dec_dense(dec_out)

    dec_model = Model([dec_input]+[dec_init_state_h, dec_init_state_c], [dec_final]+[dec_h, dec_c])
```

```
[ ] def generate_summary(input_seq):
        h, c = enc_model.predict(input_seq)

        next_token = np.zeros((1, 1))
        next_token[0, 0] = y_tokenizer.word_index['sostok']
        output_seq = ''

        stop = False
        count = 0

        while not stop:
            if count > 100:
                break
            decoder_out, state_h, state_c = dec_model.predict([next_token]+[h, c])
            token_idx = np.argmax(decoder_out[0, -1, :])

            if token_idx == y_tokenizer.word_index['eostok']:
                stop = True
            elif token_idx > 0 and token_idx != y_tokenizer.word_index['sostok']:
                token = y_tokenizer.index_word[token_idx]
                output_seq = output_seq + ' ' + token

            next_token = np.zeros((1, 1))
            next_token[0, 0] = token_idx
            h, c = state_h, state_c
            count += 1

        return output_seq
```

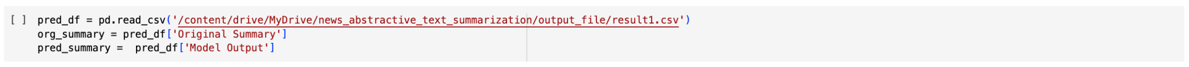
```
[ ] test_inputs = [clean_text(sent) for sent in X_test]
    test_inputs = x_tokenizer.texts_to_sequences(list(test_inputs))
    test_inputs = pad_sequences(test_inputs, maxlen=max_text_len, padding='post')
```

```
[ ] X_test.reset_index(drop=True,inplace=True)
    y_test.reset_index(drop=True,inplace=True)
```

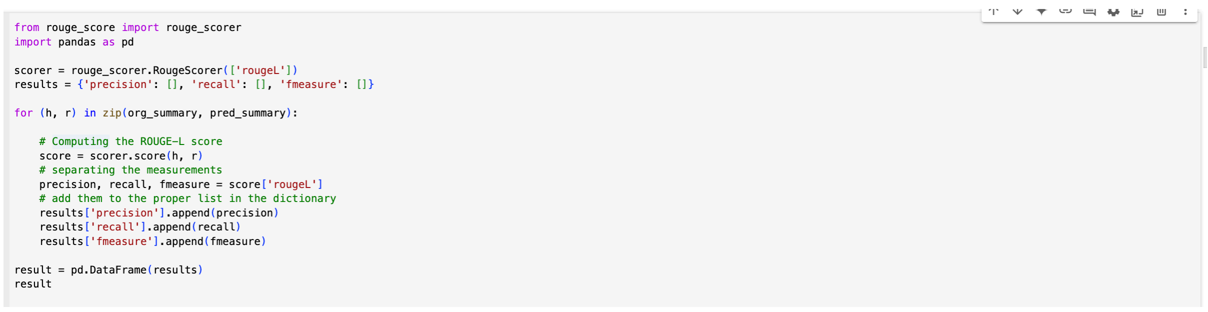
```
[ ] hyps = []
    with open('/content/drive/MyDrive/news_abstractive_text_summarization/output_file/result1.csv', 'w') as f:
        writer = csv.writer(f)
        writer.writerow(['Article', 'Original Summary', 'Model Output'])
        for i in range(10):
            our_summ = generate_summary(test_inputs[i].reshape(1, max_text_len))
            hyps.append(our_summ)
            writer.writerow([X_test[i], y_test[i], our_summ])
```

Figure 14 : Inference Setup

Results for precision, recall, and F-measure are recorded in a DataFrame comparing generated summaries against reference summaries using ROUGE-L metrics in figure 15.



```
[ ] pred_df = pd.read_csv('/content/drive/MyDrive/news_abstractive_text_summarization/output_file/result1.csv')
    org_summary = pred_df['Original Summary']
    pred_summary =  pred_df['Model Output']
```

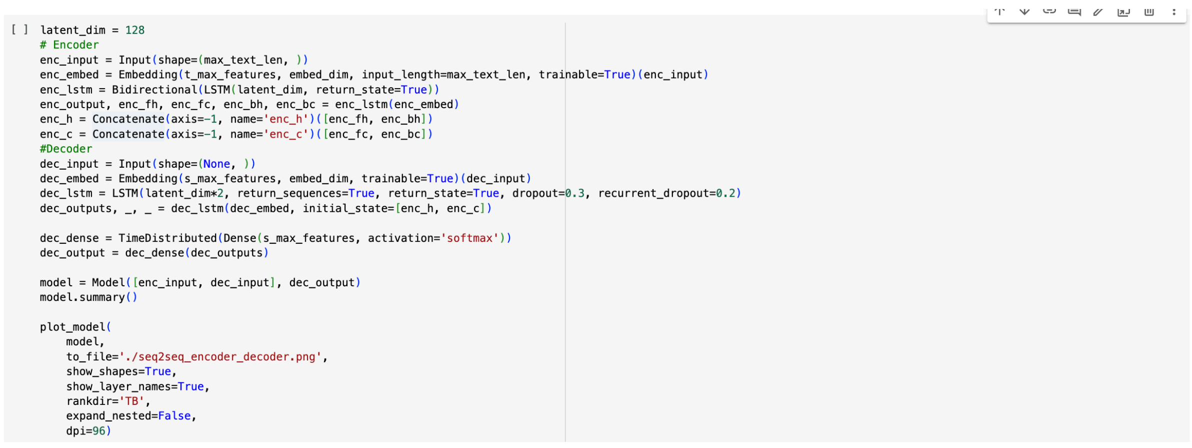


```
from rouge_score import rouge_scorer
import pandas as pd

scorer = rouge_scorer.RougeScorer(['rougeL'])
results = {'precision': [], 'recall': [], 'fmeasure': []}

for (h, r) in zip(org_summary, pred_summary):

    # Computing the ROUGE-L score
    score = scorer.score(h, r)
    # separating the measurements
    precision, recall, fmeasure = score['rougeL']
    # add them to the proper list in the dictionary
    results['precision'].append(precision)
    results['recall'].append(recall)
    results['fmeasure'].append(fmeasure)

result = pd.DataFrame(results)
result
```

Figure 15 : Evaluation Metrics

7.2 Study 2 : BiLSTM without attention layer

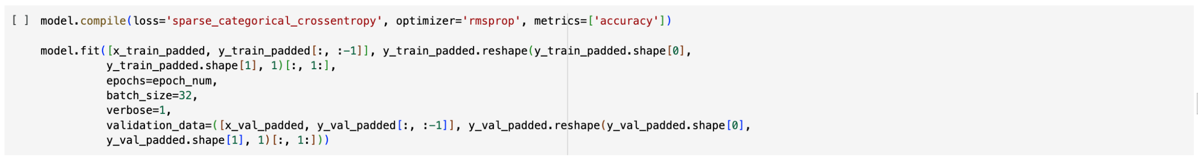
Input text is processed by the encoder (small bidirectional LSTM) forming forward and backward states, which are concatenated into final hidden and cell states, and then the decoder LSTM generates summaries using these states and predicting tokens through a time distributed dense layer with softmax is shown in figure 16.



```
[ ] latent_dim = 128
    # Encoder
    enc_input = Input(shape=(max_text_len, ))
    enc_embed = Embedding(t_max_features, embed_dim, input_length=max_text_len, trainable=True)(enc_input)
    enc_lstm = Bidirectional(LSTM(latent_dim, return_state=True))
    enc_output, enc_fh, enc_fc, enc_bh, enc_bc = enc_lstm(enc_embed)
    enc_h = Concatenate(axis=-1, name='enc_h')([enc_fh, enc_bh])
    enc_c = Concatenate(axis=-1, name='enc_c')([enc_fc, enc_bc])
    #Decoder
    dec_input = Input(shape=(None, ))
    dec_embed = Embedding(s_max_features, embed_dim, trainable=True)(dec_input)
    dec_lstm = LSTM(latent_dim*2, return_sequences=True, return_state=True, dropout=0.3, recurrent_dropout=0.2)
    dec_outputs, _, _ = dec_lstm(dec_embed, initial_state=[enc_h, enc_c])

    dec_dense = TimeDistributed(Dense(s_max_features, activation='softmax'))
    dec_output = dec_dense(dec_outputs)

    model = Model([enc_input, dec_input], dec_output)
    model.summary()

    plot_model(
        model,
        to_file='./seq2seq_encoder_decoder.png',
        show_shapes=True,
        show_layer_names=True,
        rankdir='TB',
        expand_nested=False,
        dpi=96)
```

Figure 16 : Model Architecture

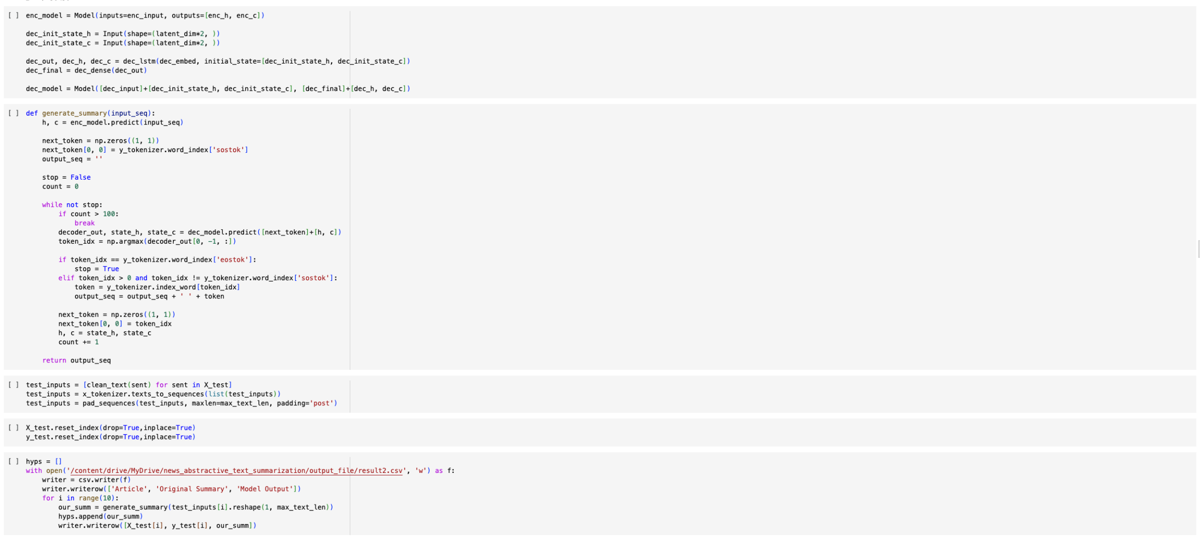
We loaded the code into Colab and compiled the model using sparse categorical crossentropy loss and optimized it with the rmsprop optimizer, trained it on padded input (x train padded) and target (y train padded) sequences for a certain number of epochs with a certain batch size and providing with validation data for evaluation purposes is done in figure 17.



```
[ ] model.compile(loss='sparse_categorical_crossentropy', optimizer='rmsprop', metrics=['accuracy'])

    model.fit([x_train_padded, y_train_padded[:, :-1]], y_train_padded.reshape(y_train_padded.shape[0],
              y_train_padded.shape[1], 1)[:, 1:],
              epochs=epoch_num,
              batch_size=32,
              verbose=1,
              validation_data=([x_val_padded, y_val_padded[:, :-1]], y_val_padded.reshape(y_val_padded.shape[0],
              y_val_padded.shape[1], 1)[:, 1:]))
```

Figure 17 : Training Configuration

Separate encoder and decoder models are constructed for prediction, where the encoder provides concatenated hidden and cell states, and the decoder generates summaries one token at a time using predicted tokens is shown in figure 18.



```
[ ] enc_model = Model(inputs=enc_input, outputs=[enc_h, enc_c])

    dec_init_state_h = Input(shape=(latent_dim*2, ))
    dec_init_state_c = Input(shape=(latent_dim*2, ))

    dec_out, dec_h, dec_c = dec_lstm(dec_embed, initial_state=[dec_init_state_h, dec_init_state_c])
    dec_final = dec_dense(dec_out)

    dec_model = Model([dec_input]+[dec_init_state_h, dec_init_state_c], [dec_final]+[dec_h, dec_c])
```

```
[ ] def generate_summary(input_seq):
        h, c = enc_model.predict(input_seq)

        next_token = np.zeros((1, 1))
        next_token[0, 0] = y_tokenizer.word_index['sostok']
        output_seq = ''

        stop = False
        count = 0

        while not stop:
            if count > 100:
                break
            decoder_out, state_h, state_c = dec_model.predict([next_token]+[h, c])
            token_idx = np.argmax(decoder_out[0, -1, :])

            if token_idx == y_tokenizer.word_index['eostok']:
                stop = True
            elif token_idx > 0 and token_idx != y_tokenizer.word_index['sostok']:
                token = y_tokenizer.index_word[token_idx]
                output_seq = output_seq + ' ' + token

            next_token = np.zeros((1, 1))
            next_token[0, 0] = token_idx
            h, c = state_h, state_c
            count += 1

        return output_seq
```

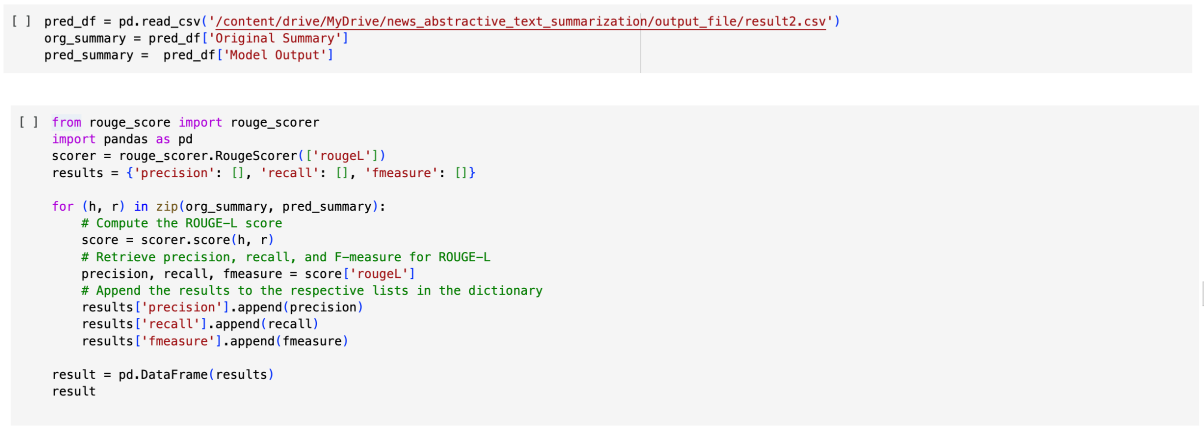
```
[ ] test_inputs = [clean_text(sent) for sent in X_test]
    test_inputs = x_tokenizer.texts_to_sequences(list(test_inputs))
    test_inputs = pad_sequences(test_inputs, maxlen=max_text_len, padding='post')
```

```
[ ] X_test.reset_index(drop=True,inplace=True)
    y_test.reset_index(drop=True,inplace=True)
```

```
[ ] hyps = []
    with open('/content/drive/MyDrive/news_abstractive_text_summarization/output_file/result2.csv', 'w') as f:
        writer = csv.writer(f)
        writer.writerow(['Article', 'Original Summary', 'Model Output'])
        for i in range(10):
            our_summ = generate_summary(test_inputs[i].reshape(1, max_text_len))
            hyps.append(our_summ)
            writer.writerow([X_test[i], y_test[i], our_summ])
```

Figure 18 : Inference setup

Results for precision, recall, and F-measure are recorded in a DataFrame comparing generated summaries against reference summaries using ROUGE-L metrics in figure 19.



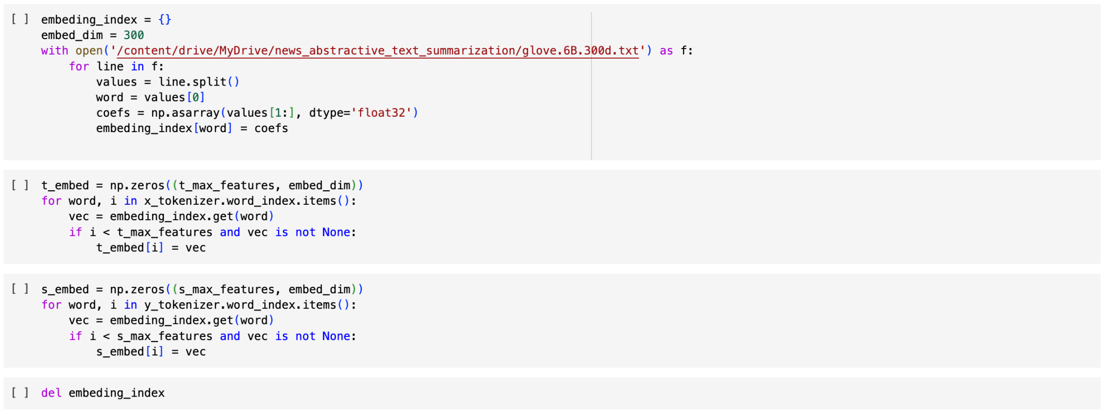
```
[ ] pred_df = pd.read_csv('/content/drive/MyDrive/news_abstractive_text_summarization/output_file/result2.csv')
    org_summary = pred_df['Original Summary']
    pred_summary =  pred_df['Model Output']
```

```
[ ] from rouge_score import rouge_scorer
    import pandas as pd
    scorer = rouge_scorer.RougeScorer(['rougeL'])
    results = {'precision': [], 'recall': [], 'fmeasure': []}

    for (h, r) in zip(org_summary, pred_summary):
        # Compute the ROUGE-L score
        score = scorer.score(h, r)
        # Retrieve precision, recall, and F-measure for ROUGE-L
        precision, recall, fmeasure = score['rougeL']
        # Append the results to the respective lists in the dictionary
        results['precision'].append(precision)
        results['recall'].append(recall)
        results['fmeasure'].append(fmeasure)

    result = pd.DataFrame(results)
    result
```

Figure 19 : Evaluation Metrics

7.3 Study 3 : LSTM with attention layer and glove embeddings

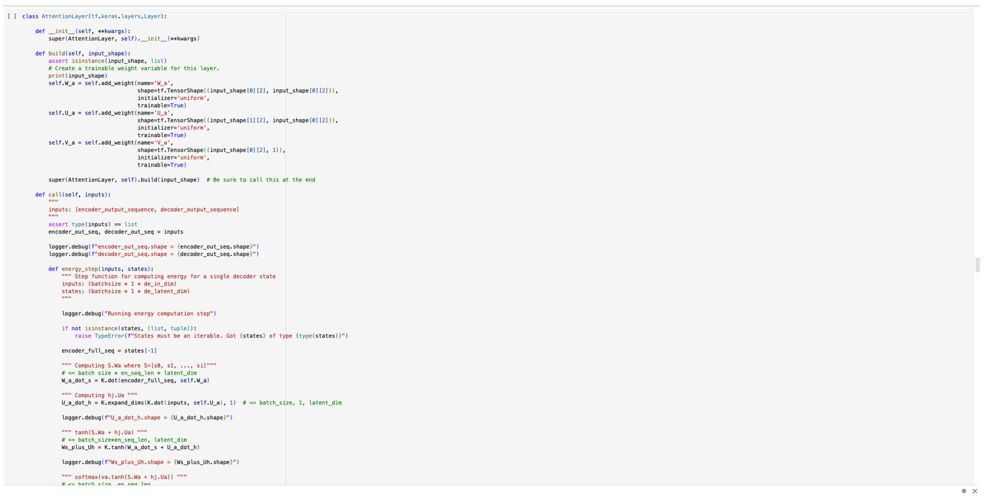
Make sure that GloVe embeddings were correctly preloaded and that it has the same embedding dimension as chosen.For this, the snapshot gives the detailed infomation (embed_dim=300) (figure 20).



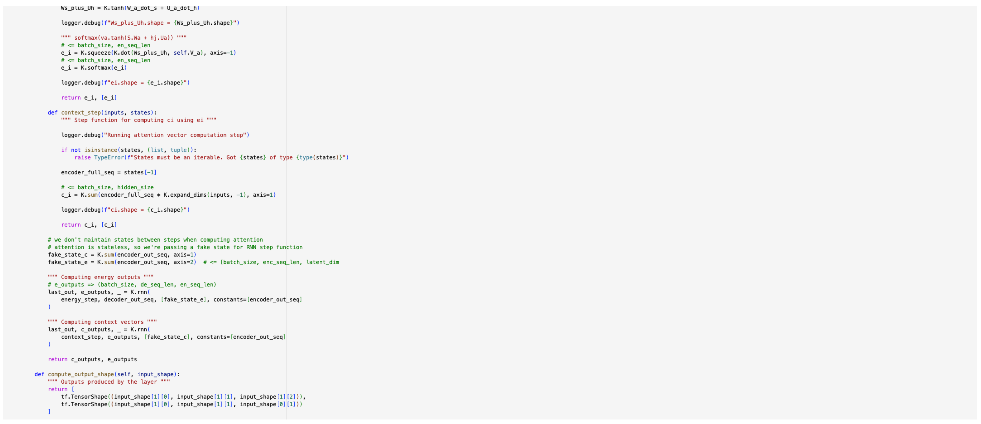
```
[ ] embeding_index = {}
    embed_dim = 300
    with open('/content/drive/MyDrive/news_abstractive_text_summarization/glove.6B.300d.txt') as f:
        for line in f:
            values = line.split()
            word = values[0]
            coefs = np.asarray(values[1:], dtype='float32')
            embeding_index[word] = coefs

[ ] t_embed = np.zeros((t_max_features, embed_dim))
    for word, i in x_tokenizer.word_index.items():
        vec = embeding_index.get(word)
        if i < t_max_features and vec is not None:
            t_embed[i] = vec

[ ] s_embed = np.zeros((s_max_features, embed_dim))
    for word, i in y_tokenizer.word_index.items():
        vec = embeding_index.get(word)
        if i < s_max_features and vec is not None:
            s_embed[i] = vec

[ ] del embeding_index
```

Figure 20 : Pre trained embedding

Ensuring the compatibility between encoder and decoder hidden states.This is explained in figure 21.



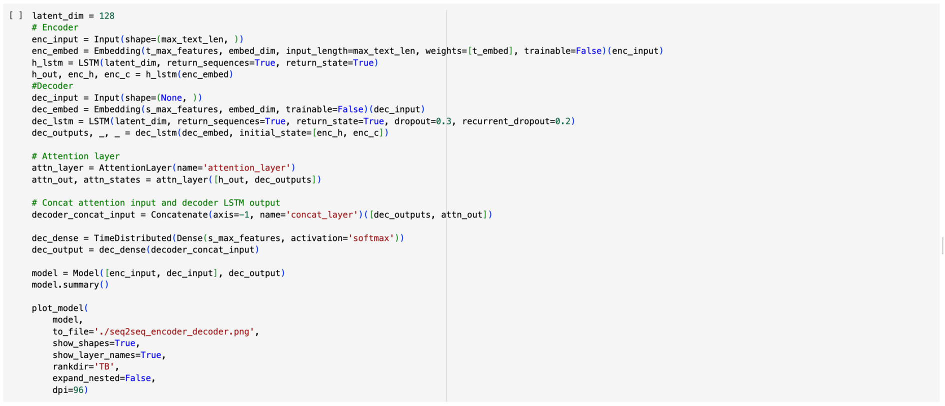
```
[ ] class AttentionLayer(tf.keras.layers.Layer):

        def __init__(self, **kwargs):
            super(AttentionLayer, self).__init__(**kwargs)

        def build(self, input_shape):
            assert isinstance(input_shape, list)
            # Create a trainable weight variable for this layer.
            print(input_shape)
            self.W_a = self.add_weight(name='W_a',
                                       shape=tf.TensorShape((input_shape[0][2], input_shape[0][2])),
                                       initializer='uniform',
                                       trainable=True)
            self.U_a = self.add_weight(name='U_a',
                                       shape=tf.TensorShape((input_shape[1][2], input_shape[0][2])),
                                       initializer='uniform',
                                       trainable=True)
            self.V_a = self.add_weight(name='V_a',
                                       shape=tf.TensorShape((input_shape[0][2], 1)),
                                       initializer='uniform',
                                       trainable=True)

            super(AttentionLayer, self).build(input_shape)  # Be sure to call this at the end

        def call(self, inputs):
            """
            inputs: [encoder_output_sequence, decoder_output_sequence]
            """
            assert type(inputs) == list
            encoder_out_seq, decoder_out_seq = inputs

            logger.debug(f"encoder_out_seq.shape = {encoder_out_seq.shape}")
            logger.debug(f"decoder_out_seq.shape = {decoder_out_seq.shape}")

            def energy_step(inputs, states):
                """ Step function for computing energy for a single decoder state
                inputs: (batchsize * 1 * de_in_dim)
                states: (batchsize * 1 * de_latent_dim)
                """

                logger.debug("Running energy computation step")

                if not isinstance(states, (list, tuple)):
                    raise TypeError(f"States must be an iterable. Got {states} of type {type(states)}")

                encoder_full_seq = states[-1]

                """ Computing S.Wa where S=[s0, s1, ..., si]"""
                # <= batch size * en_seq_len * latent_dim
                W_a_dot_s = K.dot(encoder_full_seq, self.W_a)

                """ Computing hj.Ua """
                U_a_dot_h = K.expand_dims(K.dot(inputs, self.U_a), 1)  # <= batch_size, 1, latent_dim

                logger.debug(f"U_a_dot_h.shape = {U_a_dot_h.shape}")

                """ tanh(S.Wa + hj.Ua) """
                # <= batch_size*en_seq_len, latent_dim
                Ws_plus_Uh = K.tanh(W_a_dot_s + U_a_dot_h)

                logger.debug(f"Ws_plus_Uh.shape = {Ws_plus_Uh.shape}")

                """ softmax(va.tanh(S.Wa + hj.Ua)) """
```



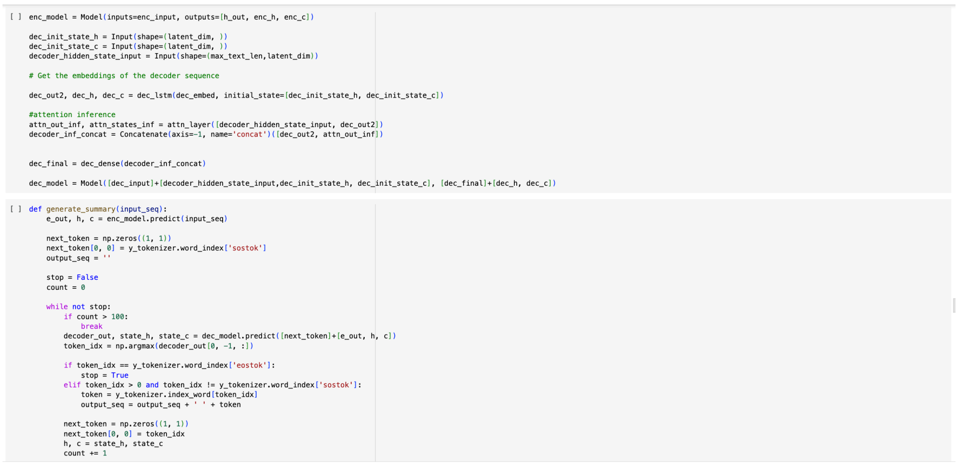
```
                Ws_plus_Uh = K.tanh(W_a_dot_s + U_a_dot_h)

                logger.debug(f"Ws_plus_Uh.shape = {Ws_plus_Uh.shape}")

                """ softmax(va.tanh(S.Wa + hj.Ua)) """
                # <= batch_size, en_seq_len
                e_i = K.squeeze(K.dot(Ws_plus_Uh, self.V_a), axis=-1)
                # <= batch_size, en_seq_len
                e_i = K.softmax(e_i)

                logger.debug(f"ei.shape = {e_i.shape}")

                return e_i, [e_i]

            def context_step(inputs, states):
                """ Step function for computing ci using ei """

                logger.debug("Running attention vector computation step")

                if not isinstance(states, (list, tuple)):
                    raise TypeError(f"States must be an iterable. Got {states} of type {type(states)}")

                encoder_full_seq = states[-1]

                # <= batch_size, hidden_size
                c_i = K.sum(encoder_full_seq * K.expand_dims(inputs, -1), axis=1)

                logger.debug(f"ci.shape = {c_i.shape}")

                return c_i, [c_i]

            # we don't maintain states between steps when computing attention
            # attention is stateless, so we're passing a fake state for RNN step function
            fake_state_c = K.sum(encoder_out_seq, axis=1)
            fake_state_e = K.sum(encoder_out_seq, axis=2)  # <= (batch_size, enc_seq_len, latent_dim

            """ Computing energy outputs """
            # e_outputs => (batch_size, de_seq_len, en_seq_len)
            last_out, e_outputs, _ = K.rnn(
                energy_step, decoder_out_seq, [fake_state_e], constants=[encoder_out_seq]
            )

            """ Computing context vectors """
            last_out, c_outputs, _ = K.rnn(
                context_step, e_outputs, [fake_state_c], constants=[encoder_out_seq]
            )

            return c_outputs, e_outputs

        def compute_output_shape(self, input_shape):
            """ Outputs produced by the layer """
            return [
                tf.TensorShape((input_shape[1][0], input_shape[1][1], input_shape[1][2])),
                tf.TensorShape((input_shape[1][0], input_shape[1][1], input_shape[0][1]))
            ]
```

Figure 21 : Attention Layer

Also, adjust `latent_dim`, dropout, and recurrent dropout for resource efficiency is shown in figure 22.

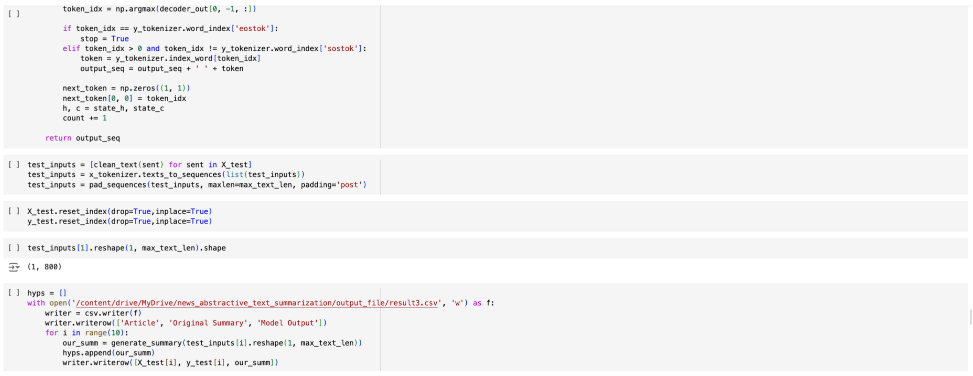


```
[ ] latent_dim = 128
    # Encoder
    enc_input = Input(shape=(max_text_len, ))
    enc_embed = Embedding(t_max_features, embed_dim, input_length=max_text_len, weights=[t_embed], trainable=False)(enc_input)
    h_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)
    h_out, enc_h, enc_c = h_lstm(enc_embed)
    #Decoder
    dec_input = Input(shape=(None, ))
    dec_embed = Embedding(s_max_features, embed_dim, trainable=False)(dec_input)
    dec_lstm = LSTM(latent_dim, return_sequences=True, return_state=True, dropout=0.3, recurrent_dropout=0.2)
    dec_outputs, _, _ = dec_lstm(dec_embed, initial_state=[enc_h, enc_c])

    # Attention layer
    attn_layer = AttentionLayer(name='attention_layer')
    attn_out, attn_states = attn_layer([h_out, dec_outputs])

    # Concat attention input and decoder LSTM output
    decoder_concat_input = Concatenate(axis=-1, name='concat_layer')([dec_outputs, attn_out])

    dec_dense = TimeDistributed(Dense(s_max_features, activation='softmax'))
    dec_output = dec_dense(decoder_concat_input)

    model = Model([enc_input, dec_input], dec_output)
    model.summary()

    plot_model(
        model,
        to_file='./seq2seq_encoder_decoder.png',
        show_shapes=True,
        show_layer_names=True,
        rankdir='TB',
        expand_nested=False,
        dpi=96)
```



```
[ ] enc_model = Model(inputs=enc_input, outputs=[h_out, enc_h, enc_c])

    dec_init_state_h = Input(shape=(latent_dim, ))
    dec_init_state_c = Input(shape=(latent_dim, ))
    decoder_hidden_state_input = Input(shape=(max_text_len,latent_dim))

    # Get the embeddings of the decoder sequence

    dec_out2, dec_h, dec_c = dec_lstm(dec_embed, initial_state=[dec_init_state_h, dec_init_state_c])

    #attention inference
    attn_out_inf, attn_states_inf = attn_layer([decoder_hidden_state_input, dec_out2])
    decoder_inf_concat = Concatenate(axis=-1, name='concat')([dec_out2, attn_out_inf])


    dec_final = dec_dense(decoder_inf_concat)

    dec_model = Model([dec_input]+[decoder_hidden_state_input,dec_init_state_h, dec_init_state_c], [dec_final]+[dec_h, dec_c])
```

```
[ ] def generate_summary(input_seq):
        e_out, h, c = enc_model.predict(input_seq)

        next_token = np.zeros((1, 1))
        next_token[0, 0] = y_tokenizer.word_index['sostok']
        output_seq = ''

        stop = False
        count = 0

        while not stop:
            if count > 100:
                break
            decoder_out, state_h, state_c = dec_model.predict([next_token]+[e_out, h, c])
            token_idx = np.argmax(decoder_out[0, -1, :])

            if token_idx == y_tokenizer.word_index['eostok']:
                stop = True
            elif token_idx > 0 and token_idx != y_tokenizer.word_index['sostok']:
                token = y_tokenizer.index_word[token_idx]
                output_seq = output_seq + ' ' + token

            next_token = np.zeros((1, 1))
            next_token[0, 0] = token_idx
            h, c = state_h, state_c
            count += 1
```



```
[ ]         token_idx = np.argmax(decoder_out[0, -1, :])

            if token_idx == y_tokenizer.word_index['eostok']:
                stop = True
            elif token_idx > 0 and token_idx != y_tokenizer.word_index['sostok']:
                token = y_tokenizer.index_word[token_idx]
                output_seq = output_seq + ' ' + token

            next_token = np.zeros((1, 1))
            next_token[0, 0] = token_idx
            h, c = state_h, state_c
            count += 1

        return output_seq
```

```
[ ] test_inputs = [clean_text(sent) for sent in X_test]
    test_inputs = x_tokenizer.texts_to_sequences(list(test_inputs))
    test_inputs = pad_sequences(test_inputs, maxlen=max_text_len, padding='post')
```

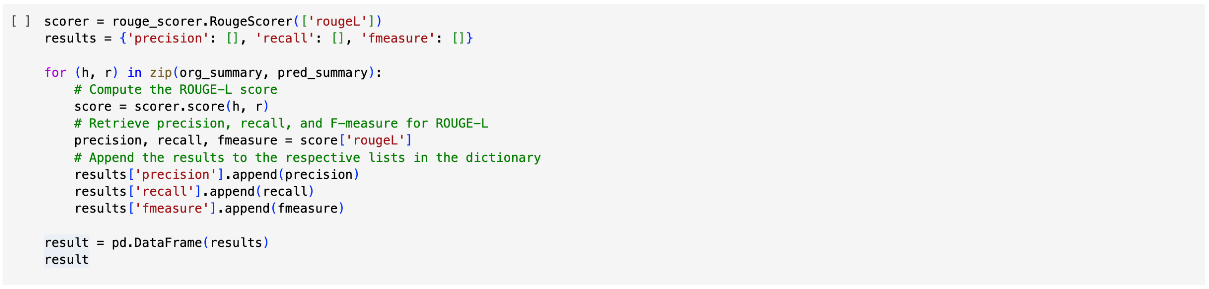
```
[ ] X_test.reset_index(drop=True,inplace=True)
    y_test.reset_index(drop=True,inplace=True)
```

```
[ ] test_inputs[1].reshape(1, max_text_len).shape

    (1, 800)
```

```
[ ] hyps = []
    with open('/content/drive/MyDrive/news_abstractive_text_summarization/output_file/result3.csv', 'w') as f:
        writer = csv.writer(f)
        writer.writerow(['Article', 'Original Summary', 'Model Output'])
        for i in range(10):
            our_summ = generate_summary(test_inputs[i].reshape(1, max_text_len))
            hyps.append(our_summ)
            writer.writerow([X_test[i], y_test[i], our_summ])
```

Figure 22: LSTM Parameters

Results for precision, recall, and F-measure are recorded in a DataFrame comparing generated summaries against reference summaries using ROUGE-L metrics in figure 23.



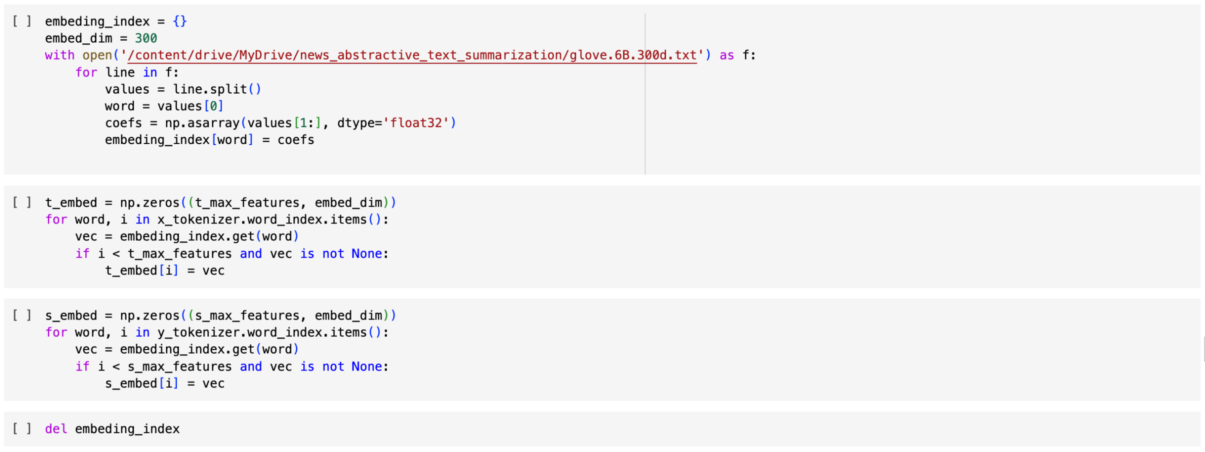
```
[ ]  scorer = rouge_scorer.RougeScorer(['rougeL'])
     results = {'precision': [], 'recall': [], 'fmeasure': []}

     for (h, r) in zip(org_summary, pred_summary):
         # Compute the ROUGE-L score
         score = scorer.score(h, r)
         # Retrieve precision, recall, and F-measure for ROUGE-L
         precision, recall, fmeasure = score['rougeL']
         # Append the results to the respective lists in the dictionary
         results['precision'].append(precision)
         results['recall'].append(recall)
         results['fmeasure'].append(fmeasure)

     result = pd.DataFrame(results)
     result
```

Figure 23: Evaluation Metrices

7.4 : Study 4: BiLSTM with attention layer and Glove Embedding

Pre trained GloVe embeddings are also utilized to give semantic rich word representation. While the decoder produces summaries sequentially with the encoder last output states as input, it is initialized with the final states of the encoder which consists of a bidirectional LSTM, which captures the contextual information in both the forward and backward directions. Context aware generation is facilitated with a custom attention layer that aligns the decoder's focus with the most relevant bits of the encoder's outputs. Finally, token probabilities are obtained from model output via a dense layer with the softmax activation. A system for the abstraction, understanding, and generation of natures of that relationship is presented, parametrized by a summarization relation, and evaluated with ROUGE-L, a metric that has proven effective at measuring summarization performance.

The same step will be followed for this model which is shown in figure 20,21 and 22. There will be slight changes in the model specification, instad of LSTM model, we will be using BiLSTM model as shown in figure 24.

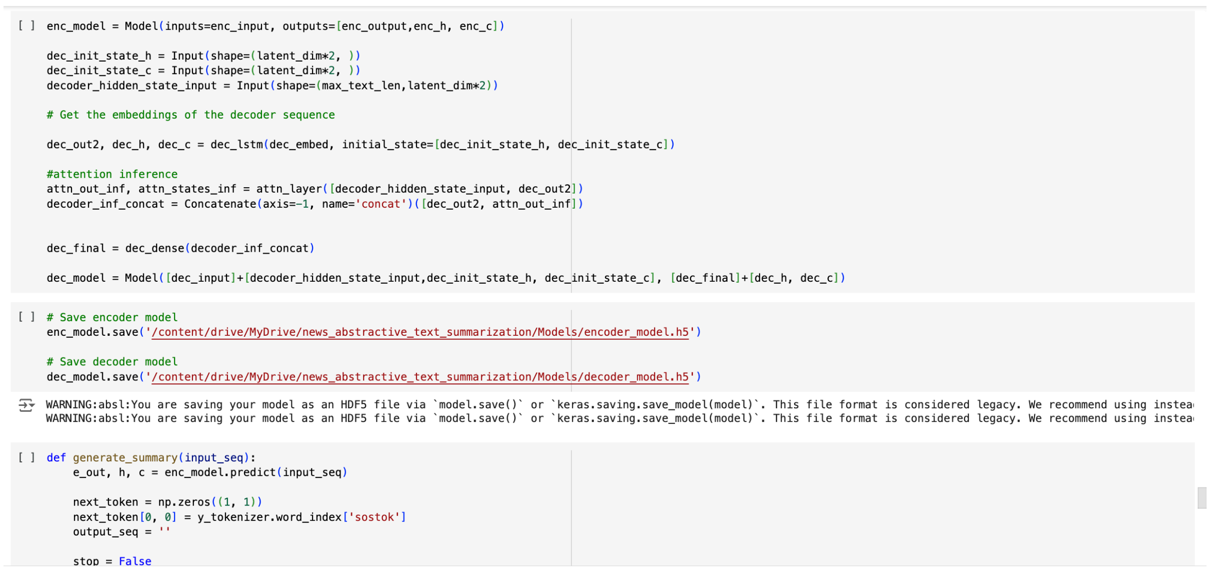


```
[ ]  embeding_index = {}
     embed_dim = 300
     with open('/content/drive/MyDrive/news_abstractive_text_summarization/glove.6B.300d.txt') as f:
         for line in f:
             values = line.split()
             word = values[0]
             coefs = np.asarray(values[1:], dtype='float32')
             embeding_index[word] = coefs
```

```
[ ]  t_embed = np.zeros((t_max_features, embed_dim))
     for word, i in x_tokenizer.word_index.items():
         vec = embeding_index.get(word)
         if i < t_max_features and vec is not None:
             t_embed[i] = vec
```

```
[ ]  s_embed = np.zeros((s_max_features, embed_dim))
     for word, i in y_tokenizer.word_index.items():
         vec = embeding_index.get(word)
         if i < s_max_features and vec is not None:
             s_embed[i] = vec
```

```
[ ]  del embeding_index
```

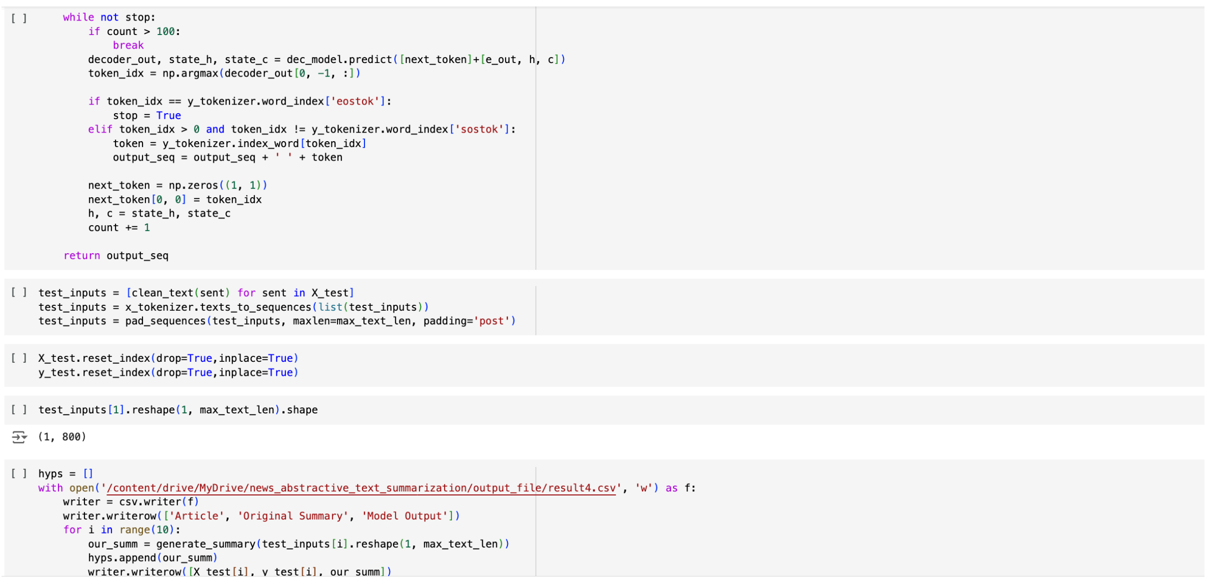


```
[ ] enc_model = Model(inputs=enc_input, outputs=[enc_output,enc_h, enc_c])

    dec_init_state_h = Input(shape=(latent_dim*2, ))
    dec_init_state_c = Input(shape=(latent_dim*2, ))
    decoder_hidden_state_input = Input(shape=(max_text_len,latent_dim*2))

    # Get the embeddings of the decoder sequence

    dec_out2, dec_h, dec_c = dec_lstm(dec_embed, initial_state=[dec_init_state_h, dec_init_state_c])

    #attention inference
    attn_out_inf, attn_states_inf = attn_layer([decoder_hidden_state_input, dec_out2])
    decoder_inf_concat = Concatenate(axis=-1, name='concat')([dec_out2, attn_out_inf])


    dec_final = dec_dense(decoder_inf_concat)

    dec_model = Model([dec_input]+[decoder_hidden_state_input,dec_init_state_h, dec_init_state_c], [dec_final]+[dec_h, dec_c])
```

```
[ ] # Save encoder model
    enc_model.save('/content/drive/MyDrive/news_abstractive_text_summarization/Models/encoder_model.h5')

    # Save decoder model
    dec_model.save('/content/drive/MyDrive/news_abstractive_text_summarization/Models/decoder_model.h5')
```

```
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instea
WARNING:absl:You are saving your model as an HDF5 file via `model.save()` or `keras.saving.save_model(model)`. This file format is considered legacy. We recommend using instea
```

```
[ ] def generate_summary(input_seq):
        e_out, h, c = enc_model.predict(input_seq)

        next_token = np.zeros((1, 1))
        next_token[0, 0] = y_tokenizer.word_index['sostok']
        output_seq = ''

        stop = False
```



```
[ ]     while not stop:
            if count > 100:
                break
            decoder_out, state_h, state_c = dec_model.predict([next_token]+[e_out, h, c])
            token_idx = np.argmax(decoder_out[0, -1, :])

            if token_idx == y_tokenizer.word_index['eostok']:
                stop = True
            elif token_idx > 0 and token_idx != y_tokenizer.word_index['sostok']:
                token = y_tokenizer.index_word[token_idx]
                output_seq = output_seq + ' ' + token

            next_token = np.zeros((1, 1))
            next_token[0, 0] = token_idx
            h, c = state_h, state_c
            count += 1

        return output_seq
```

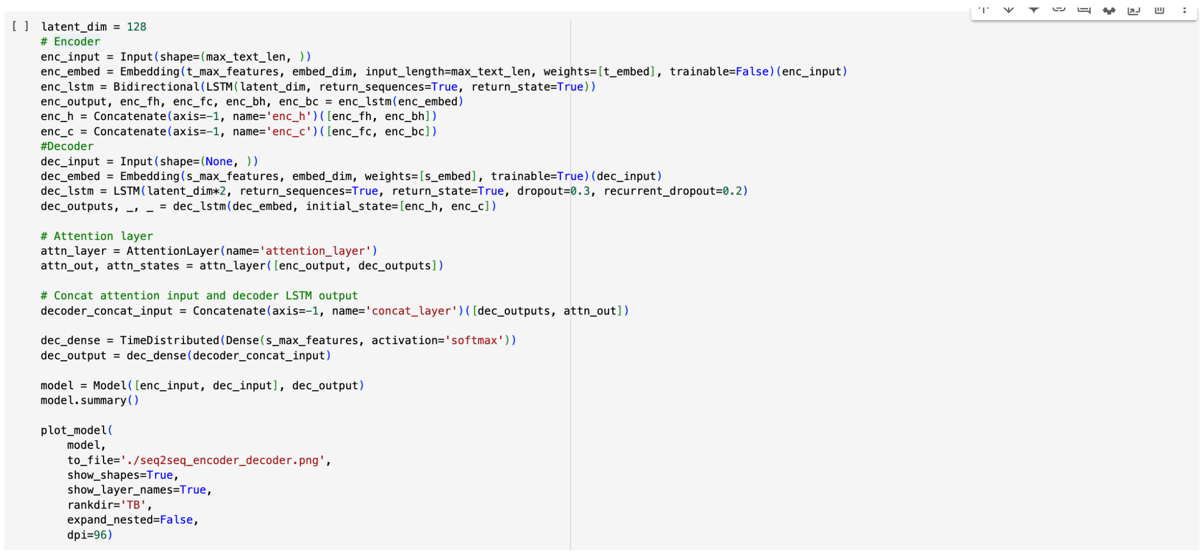
```
[ ] test_inputs = [clean_text(sent) for sent in X_test]
    test_inputs = x_tokenizer.texts_to_sequences(list(test_inputs))
    test_inputs = pad_sequences(test_inputs, maxlen=max_text_len, padding='post')
```

```
[ ] X_test.reset_index(drop=True,inplace=True)
    y_test.reset_index(drop=True,inplace=True)
```

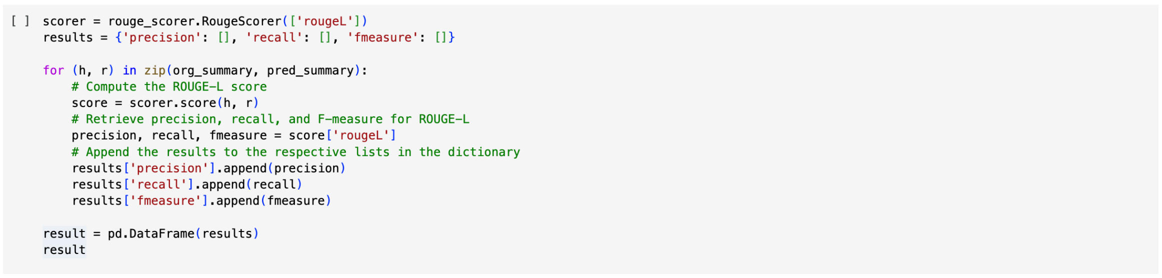
```
[ ] test_inputs[1].reshape(1, max_text_len).shape
```

```
(1, 800)
```

```
[ ] hyps = []
    with open('/content/drive/MyDrive/news_abstractive_text_summarization/output_file/result4.csv', 'w') as f:
        writer = csv.writer(f)
        writer.writerow(['Article', 'Original Summary', 'Model Output'])
        for i in range(10):
            our_summ = generate_summary(test_inputs[i].reshape(1, max_text_len))
            hyps.append(our_summ)
            writer.writerow([X_test[i], y_test[i], our_summ])
```



```
[ ] latent_dim = 128
    # Encoder
    enc_input = Input(shape=(max_text_len, ))
    enc_embed = Embedding(t_max_features, embed_dim, input_length=max_text_len, weights=[t_embed], trainable=False)(enc_input)
    enc_lstm = Bidirectional(LSTM(latent_dim, return_sequences=True, return_state=True))
    enc_output, enc_fh, enc_fc, enc_bh, enc_bc = enc_lstm(enc_embed)
    enc_h = Concatenate(axis=-1, name='enc_h')([enc_fh, enc_bh])
    enc_c = Concatenate(axis=-1, name='enc_c')([enc_fc, enc_bc])
    #Decoder
    dec_input = Input(shape=(None, ))
    dec_embed = Embedding(s_max_features, embed_dim, weights=[s_embed], trainable=True)(dec_input)
    dec_lstm = LSTM(latent_dim*2, return_sequences=True, return_state=True, dropout=0.3, recurrent_dropout=0.2)
    dec_outputs, _, _ = dec_lstm(dec_embed, initial_state=[enc_h, enc_c])

    # Attention layer
    attn_layer = AttentionLayer(name='attention_layer')
    attn_out, attn_states = attn_layer([enc_output, dec_outputs])

    # Concat attention input and decoder LSTM output
    decoder_concat_input = Concatenate(axis=-1, name='concat_layer')([dec_outputs, attn_out])

    dec_dense = TimeDistributed(Dense(s_max_features, activation='softmax'))
    dec_output = dec_dense(decoder_concat_input)

    model = Model([enc_input, dec_input], dec_output)
    model.summary()

    plot_model(
        model,
        to_file='./seq2seq_encoder_decoder.png',
        show_shapes=True,
        show_layer_names=True,
        rankdir='TB',
        expand_nested=False,
        dpi=96)
```



```
[ ] scorer = rouge_scorer.RougeScorer(['rougeL'])
    results = {'precision': [], 'recall': [], 'fmeasure': []}

    for (h, r) in zip(org_summary, pred_summary):
        # Compute the ROUGE-L score
        score = scorer.score(h, r)
        # Retrieve precision, recall, and F-measure for ROUGE-L
        precision, recall, fmeasure = score['rougeL']
        # Append the results to the respective lists in the dictionary
        results['precision'].append(precision)
        results['recall'].append(recall)
        results['fmeasure'].append(fmeasure)

    result = pd.DataFrame(results)
    result
```

Figure 24 : Complete model of BiLSTM with attention layer and glove embeddings

8. Results

After the successful completion of implementation code, you could find results getting displayed. The results are evaluated with the metrics like accuracy, precision, F1 score, and recall.