ELC ACTIVITY

Q1 . Develop a language-independent extractive text summarization system that generates a summary for a given document .

Text summarization means identifying important sections of the text and generating them verbatim producing a subset of the sentences from the original text; while abstractive summarization reproduces important material in a new way after interpretation and examination of the text using advanced natural language techniques to generate a new shorter text that conveys the most critical information from the original one.

Frequency based approach - This approach uses the frequency of words as indicators of importance. The two most common techniques in this category are word probability and TFIDF (Term Frequency Inverse Document Frequency). The probability of a word w is determined as the number of occurrences of the word, f (w), divided by the number of all words in the input (which can be a single document or multiple documents). Words with the highest probability are assumed to represent the topic of the document and are included in the summary.

Text summarization methods aim at producing summary by interpreting the text using advanced natural language techniques in order to generate a new shorter text — parts of which may not appear as part of the original document, that conveys the most critical information from the original text, requiring rephrasing sentences and incorporating information from full text to generate summaries such as a human-written abstract usually does. In fact, an acceptable abstractive summary covers core information in the input and is linguistically fluent.

Abstractive methods take advantage of recent developments in deep learning. Since it can be regarded as a sequence mapping task where the source text should be mapped to the target summary, abstractive methods take advantage of the recent success of the sequence to sequence models. These models consist of an encoder and a decoder, where a neural network reads the text, encodes it, and then generates target text.

By generating automatic summaries, text summarization helps content editors save time and effort, which otherwise is invested in creating summaries of articles manually.

Code for text summarizer –

```
#DataFlair Project
#import all the required libraries import numpy as np import
pandas as pd import pickle from statistics import mode import
nltk from nltk import word_tokenize from nltk.stem import
LancasterStemmer nltk.download('wordnet')
nltk.download('stopwords') nltk.download('punkt') from
nltk.corpus import stopwords from tensorflow.keras.models import
Model from tensorflow.keras import models from tensorflow.keras
import backend as K  from
tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.preprocessing.text import Tokenizer  from
tensorflow.keras.utils import plot_model from
tensorflow.keras.layers import
Input,LSTM,Embedding,Dense,Concatenate,Attention from
sklearn.model_selection import train_test_split from
bs4 import BeautifulSoup

#read the dataset file df=pd.read_csv("labelled
sentences.xlx",nrows=100000) #drop the duplicate and
na values from the records
df.drop_duplicates(subset=['Text'],inplace=True)
df.dropna(axis=0,inplace=True) input_data =
df.loc[:,'Text'] target_data = df.loc[:,'Summary']
target.replace('', np.nan, inplace=True)

input_texts=[] target_texts=[] input_words=[] target_words=[]
contractions=
pickle.load(open("contractions.pkl","rb"))['contractions']
#initialize stop words and LancasterStemmer
stop_words=set(stopwords.words('english'))
```

```
stemm=LancasterStemmer()
 def clean(texts,src):   #remove the html
tags   texts = BeautifulSoup(texts,
"lxml").text
  #tokenize the text into words
words=word_tokenize(texts.lower())
  #filter words which contains \
  #integers or their length is less than or equal to 3   words=
list(filter(lambda w:(w.isalpha() and len(w)>=3),words))
  #contraction file to expand shortened words   words=
[contractions[w] if w in contractions else w for w in words ]
  #stem the words to their root word and filter stop words
if src=="inputs":
    words= [stemm.stem(w) for w in words if w not in stop_words]
else:
    words= [w for w in words if w not in stop_words]
return words

#pass the input records and taret records for
in_txt,tr_txt in zip(input_data,target_data):
  in_words= clean(in_txt,"inputs")
input_texts+= [' '.join(in_words)]
input_words+= in_words
  #add 'sos' at start and 'eos' at end of text
tr_words= clean("sos "+tr_txt+" eos","target")
target_texts+= [' '.join(tr_words)]   target_words+=
tr_words

#store only unique words from input and target list of
words input_words = sorted(list(set(input_words)))
target_words = sorted(list(set(target_words)))
num_in_words = len(input_words) #total number of input words
num_tr_words = len(target_words) #total number of target words
#get the length of the input and target texts which appears most
often   max_in_len = mode([len(i) for i in input_texts]) max_tr_len =
mode([len(i) for i in target_texts])
 print("number of input words :
",num_in_words) print("number of target words
: ",num_tr_words) print("maximum input length
: ",max_in_len) print("maximum target length :
",max_tr_len)
```

```
      _train,x_test,y_train,y_test=train_test_split(input_texts,target_texts,test
_size =0.2,random_state=0)

#train the tokenizer with all the words
in_tokenizer = Tokenizer()
in_tokenizer.fit_on_texts(x_train) tr_tokenizer
= Tokenizer()
tr_tokenizer.fit_on_texts(y_train)

#convert text into sequence of integers
#where the integer will be the index of that word
x_train= in_tokenizer.texts_to_sequences(x_train)
y_train= tr_tokenizer.texts_to_sequences(y_train)
#pad array of 0's if the length is less than the maximum length
en_in_data= pad_sequences(x_train,  maxlen=max_in_len, padding='post')
dec_data= pad_sequences(y_train,  maxlen=max_tr_len, padding='post')
#decoder input data will not include the last word
#i.e. 'eos' in decoder input data dec_in_data
= dec_data[:,:-1]
#decoder target data will be one time step ahead as it will not include
# the first word i.e 'sos' dec_tr_data =
dec_data.reshape(len(dec_data),max_tr_len,1)[:,1:]
K.clear_session()  latent_dim
= 500

#create input object of total number of input words en_inputs
= Input(shape=(max_in_len,))
en_embedding = Embedding(num_in_words+1, latent_dim)(en_inputs)
#create 3 stacked LSTM layer with the shape of hidden dimension
#LSTM 1 en_lstm1= LSTM(latent_dim, return_state=True,
return_sequences=True)  en_outputs1, state_h1, state_c1=
en_lstm1(en_embedding)   #LSTM2 en_lstm2= LSTM(latent_dim,
return_state=True, return_sequences=True)  en_outputs2, state_h2,
state_c2= en_lstm2(en_outputs1)   #LSTM3 en_lstm3=
LSTM(latent_dim,return_sequences=True,return_state=True) en_outputs3
, state_h3 , state_c3= en_lstm3(en_outputs2)
```

```
 #encoder states en_states=
[state_h3, state_c3]

# Decoder.
dec_inputs = Input(shape=(None,))  dec_emb_layer =
Embedding(num_tr_words+1, latent_dim)  dec_embedding
= dec_emb_layer(dec_inputs)

#initialize decoder's LSTM layer with the output states of encoder
dec_lstm = LSTM(latent_dim, return_sequences=True,
return_state=True) dec_outputs, *_ =
dec_lstm(dec_embedding,initial_state=en_states)
#Attention layer attention =Attention()
attn_out =
attention([dec_outputs,en_outputs3])

#Concatenate the attention output with the decoder ouputs
merge=Concatenate(axis=-1, name='concat_layer1')([dec_outputs,attn_out])
#Dense layer (output layer) dec_dense =
Dense(num_tr_words+1, activation='softmax')
dec_outputs = dec_dense(merge)

#Mode class and model summary model = Model([en_inputs,
dec_inputs], dec_outputs)  model.summary() plot_model(model,
to_file='model_plot.png', show_shapes=True,
show_layer_names=True)
 model.compile(
    optimizer="rmsprop",
loss="sparse_categorical_crossentropy", metrics=["accuracy"] )
model.fit(
    [en_in_data, dec_in_data],
dec_tr_data,
batch_size=512,      epochs=10,
    validation_split=0.1,
    )

#Save model
model.save("s2s")

# encoder inference
```

```
latent_dim=500 #load the model
model = models.load_model("s2s")

#construct encoder model from the output of 6 layer i.e.last LSTM layer
en_outputs,state_h_enc,state_c_enc =
model.layers[6].output en_states=[state_h_enc,state_c_enc]
#add input and state from the layer.
en_model = Model(model.input[0],[en_outputs]+en_states)
# decoder inference
#create Input object for hidden and cell state for decoder
#shape of layer with hidden or latent dimension
dec_state_input_h = Input(shape=(latent_dim,))
dec_state_input_c = Input(shape=(latent_dim,))
dec_hidden_state_input = Input(shape=(max_in_len,latent_dim))
# Get the embeddings and input layer from the
model dec_inputs = model.input[1] dec_emb_layer =
model.layers[5] dec_lstm = model.layers[7]
dec_embedding= dec_emb_layer(dec_inputs)

#add input and initialize LSTM layer with encoder LSTM states.
dec_outputs2, state_h2, state_c2 =
dec_lstm(dec_embedding,
initial_state=[dec_state_input_h,dec_state_input_c])
#Attention layer attention = model.layers[8] attn_out2 =
attention([dec_outputs2,dec_hidden_state_input])

merge2 = Concatenate(axis=-1)([dec_outputs2, attn_out2])
#Dense layer
dec_dense = model.layers[10]
dec_outputs2 = dec_dense(merge2)
# Finally define the Model Class
dec_model = Model(
[dec_inputs] + [dec_hidden_state_input,dec_state_input_h,dec_state_input_c],
[dec_outputs2] + [state_h2, state_c2])

#create a dictionary with a key as index and value as words.
reverse_target_word_index = tr_tokenizer.index_word
reverse_source_word_index = in_tokenizer.index_word
```

```
target_word_index = tr_tokenizer.word_index
reverse_target_word_index[0]=' '
 def
decode_sequence(input_seq):
    #get the encoder output and states by passing the input sequence
    en_out, en_h, en_c= en_model.predict(input_seq)
    #target sequence with inital word as 'sos'
target_seq = np.zeros((1, 1))
target_seq[0, 0] = target_word_index['sos']
    #if the iteration reaches the end of text than it will be stop the iteration
stop_condition = False
    #append every predicted word in decoded
sentence     decoded_sentence = ""     while not
stop_condition:
        #get predicted output, hidden and cell state.
output_words, dec_h, dec_c= dec_model.predict([target_seq] +
[en_out,en_h, en_c])

        #get the index and from the dictionary get the word for that
index.         word_index = np.argmax(output_words[0, -1, :])
text_word = reverse_target_word_index[word_index]         decoded_sentence
+= text_word +" "

        # Exit condition: either hit max length
        # or find a stop word or last word.         if text_word ==
"eos" or len(decoded_sentence) > max_tr_len:
          stop_condition = True

        #update target sequence to the current word index.
        target_seq = np.zeros((1, 1))
target_seq[0, 0] = word_index         en_h,
en_c = dec_h, dec_c

    #return the deocded sentence
return decoded_sentence
 inp_review = input("Enter :
") print("Review\\
:",inp_review)

inp_review = clean(inp_review,"inputs") inp_review = '
'.join(inp_review) inp_x=
in_tokenizer.texts_to_sequences([inp_review])  inp_x=
pad_sequences(inp_x,  maxlen=max_in_len, padding='post')
```

```
 summary=decode_sequence(inp_x.reshape(1,max_in_len))
if 'eos' in summary :
  summary=summary.replace('eos','')
print("\nPredicted summary:",summary);print("\n")
```

Q2 . Develop a question answering system that can answer When/Where/Who type questions from a given set of documents.

Question Answering system is a system that gives appropriate answers to questions expressed in natural languages such as English, Chinese, and so on. For example, suppose a user asks “When was Abraham Lincoln assassinated?” In this case, the question answering system is expected to return “Apr 15, 1865”. Below is an example of question-answer pairs.

A question answering system will help you find information efficiently. Generally speaking, we use search engines to search for relevant documents when we look for some information on the Web. However, because they show you documents, you must read the documents and decide whether they contain the information you need. It’s a bother. Thus, commercial search engines have a question answering feature so that you can find information efficiently.

Question answering systems have several paradigms, but two major paradigms have been used: knowledge-based and information retrieval based question answering. When human beings answer a question, they first try to answer the question with their own knowledge. If they can’t answer the question, they look for an answer on the internet or in books. Question answering systems are similar to human beings. The former corresponds to a knowledge-based system, and the latter corresponds to an information retrieval based system. **Step 1: Query Processing** we will generate a search query by using spaCy. First, the question is parsed and tagged with part-ofspeech tags. Next, we remove words based on their part-of-speech tags. Specifically, words other than proper nouns (PROPN), numbers (NUM), verbs (VERB), nouns (NOUN), and adjectives (ADJ) are removed. For example, the question “Who is the founder of Amazon?” generates the query “founder

Amazon”.

Step 2: Document Retrieval we will search for Wikipedia. Wikipedia provides an API called MediaWiki API. We can use the API to search the documents related to the query. The process here consists of two

steps. First, we send a query to get a list of related pages. Then we fetch the contents of the individual pages. **Step 3: Passage Retrieval** we will select passages that are similar to the question. To calculate the similarity, we use BM25 to create vectors of questions and passages. Once we have created the vectors, we can use the dot product and cosine similarity to calculate the similarity between the question and the passage. Because BM25 is powerful, it is often used as a baseline for passage retrieval. **Step 4: Answer Extraction** we formulate the answer extraction as context-aware question answering and solve it with BERT. By inputting the question and passage to the BERT, we can get the offset of the answer. It is known that BERT can solve the answer extraction well and outperforms humans on the SQuAD dataset[2][3].

Code for Q/A system –

Components.py

```
import
concurrent.futures
import itertools import
operator import re

import requests import wikipedia from
gensim.summarization.bm25 import BM25
from transformers import AutoTokenizer, AutoModelForQuestionAnswering,
QuestionAnsweringPipeline


class QueryProcessor:
```

```
    def __init__(self, nlp, keep=None):
        self.nlp = nlp         self.keep = keep or {'PROPN',
'NUM', 'VERB', 'NOUN', 'ADJ'}
     def generate_query(self,
text):
        doc = self.nlp(text)         query = ' '.join(token.text for token in
doc if token.pos_ in self.keep)          return query


class DocumentRetrieval:
     def                                          __init__(self,
url='https://en.wikipedia.org/w/api.php'):
        self.url = url
     def search_pages(self,
query):
        params = {
            'action': 'query',
            'list': 'search',
            'srsearch': query,
            'format': 'json'
        }         res = requests.get(self.url,
params=params)         return res.json()
     def search_page(self,
page_id):
        res = wikipedia.page(pageid=page_id)
return res.content
     def search(self,
query):
        pages = self.search_pages(query)         with
concurrent.futures.ThreadPoolExecutor() as executor:
            process_list = [executor.submit(self.search_page, page['pageid']) for
page in pages['query']['search']]
            docs = [self.post_process(p.result()) for p in process_list]
return docs
     def post_process(self,
doc):         pattern =
'|'.join([
            '== References ==',
            '== Further reading ==',
            '== External links',
            '== See also ==',
            '== Sources ==',
            '== Notes ==',
```

```
            '== Further references ==',
            '== Footnotes ==',
'=== Notes ===',
            '=== Sources ===',
            '=== Citations ===',
        ])          p = re.compile(pattern)
indices = [m.start() for m in p.finditer(doc)]
min_idx = min(*indices, len(doc))          return
doc[:min_idx]


class PassageRetrieval:
    def __init__(self,
nlp):
        self.tokenize = lambda text: [token.lemma_ for token in
nlp(text)]          self.bm25 = None          self.passages = None
    def preprocess(self,
doc):
        passages = [p for p in doc.split('\n') if p and not p.startswith('=')]
return passages
    def fit(self,
docs):
        passages = list(itertools.chain(*map(self.preprocess,
docs)))          corpus = [self.tokenize(p) for p in passages]
self.bm25 = BM25(corpus)          self.passages = passages
    def most_similar(self, question, topn=10):
tokens = self.tokenize(question)          scores =
self.bm25.get_scores(tokens)          pairs = [(s, i)
for i, s in enumerate(scores)]
pairs.sort(reverse=True)
        passages = [self.passages[i] for _, i in pairs[:topn]]
return passages


class AnswerExtractor:
    def __init__(self, tokenizer,
model):
        tokenizer = AutoTokenizer.from_pretrained(tokenizer)
model = AutoModelForQuestionAnswering.from_pretrained(model)
        self.nlp = QuestionAnsweringPipeline(model=model, tokenizer=tokenizer)
```

```
    def extract(self, question, passages):
        answers = []
for passage in passages:
try:
                answer = self.nlp(question=question,
context=passage)                answer['text'] = passage
answers.append(answer)              except KeyError:
                pass
answers.sort(key=operator.itemgetter('score'), reverse=True)
return answers
```

app.py

```
import os
 import spacy from flask import Flask, render_template,
jsonify, request

from src.components import QueryProcessor, DocumentRetrieval, PassageRetrieval,
AnswerExtractor

app = Flask(__name__)
SPACY_MODEL = os.environ.get('SPACY_MODEL', 'en_core_web_sm')
QA_MODEL = os.environ.get('QA_MODEL', 'distilbert-base-cased-distilled-
squad') nlp = spacy.load(SPACY_MODEL, disable=['ner', 'parser', 'textcat'])
query_processor = QueryProcessor(nlp) document_retriever =
DocumentRetrieval() passage_retriever = PassageRetrieval(nlp)
answer_extractor = AnswerExtractor(QA_MODEL, QA_MODEL)

@app.route('/') def
index():
    return render_template('index.html')

@app.route('/answer-question', methods=['POST']) def
analyzer():
```

```
    data = request.get_json()
question = data.get('question')
     query = query_processor.generate_query(question)
docs = document_retriever.search(query)
passage_retriever.fit(docs)     passages =
passage_retriever.most_similar(question)     answers =
answer_extractor.extract(question, passages)     return
jsonify(answers)


if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```