**ELC activity**

**Group member – Sankalp Sharma(102197012)**

**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) |

#split the input and target text into 80:20 ratio or testing size of 20%.

|  |
| --- |
| x\_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.](https://www.mediawiki.org/wiki/API:Main_page) 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) |