WEEK -7

Text Summarization

a. Basic Text Summarization using TF-IDF and Cosine Similarity

```
import nltk

import numpy as np

import re

from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.metrics.pairwise import cosine_similarity


nltk.download("punkt")

text = """

Artificial Intelligence is rapidly transforming the world.

It has applications in healthcare, finance, education, and more.

Ongoing research aims to make AI safer and more transparent.

However, ethical challenges remain in the widespread adoption of AI.

"""

sentences = nltk.sent_tokenize(text)

cleaned = [re.sub(r'[^a-zA-Z ]', '', s.lower()) for s in sentences]

vectorizer = TfidfVectorizer(stop_words='english')

tfidf_matrix = vectorizer.fit_transform(cleaned)

similarity_matrix = cosine_similarity(tfidf_matrix, tfidf_matrix)

scores = similarity_matrix.sum(axis=1)

N = 2

top_sentence_idx = np.argsort(scores)[-N:]

summary = " ".join([sentences[i] for i in sorted(top_sentence_idx)])

print("Summary:")

print(summary)
```

b. Abstractive Text Summarization with Transformers (google collab)

```
from transformers import pipeline
summarizer = pipeline("summarization", model="facebook/bart-large-cnn")
text = """The Apollo 11 mission was the first human spaceflight to land on the Moon.
Commander Neil Armstrong and lunar module pilot Buzz Aldrin landed on July 20,
1969, while Michael Collins orbited above. Armstrong became the first human to step
onto the Moon."""
summary = summarizer(text, max_length=50, min_length=25, do_sample=False)
print("Abstractive Summary:\n", summary[0]['summary_text'])
```

c. Extractive Summarization using BERT and SpaCy (goole collab)

```
import spacy
from sentence_transformers import SentenceTransformer, util
nlp = spacy.load("en_core_web_sm")
model = SentenceTransformer('bert-base-nli-mean-tokens')
text = """Machine learning is a branch of artificial intelligence that focuses on building
systems that learn from data.
It has become essential in applications like recommendation systems, fraud detection, and
autonomous driving.
Despite its success, challenges such as overfitting and data quality issues remain
important considerations."""
doc = nlp(text)
sentences = [sent.text.strip() for sent in doc.sents]
embeddings = model.encode(sentences, convert_to_tensor=True)
cosine_scores = util.cos_sim(embeddings, embeddings)
avg_scores = cosine_scores.mean(dim=1)
summary_sentences = [sentences[0]]
top_other = avg_scores.argsort(descending=True)[0].item()
if top_other != 0:
    summary_sentences.append(sentences[top_other])

summary = ' '.join(summary_sentences)

print("Extractive Summary:\n", summary)
```

WEEK -8

Text Entailment

a. Basic Text Entailment using Simple Rule-Based Methods

```
import nltk

from nltk.corpus import stopwords

from nltk.tokenize import word_tokenize

nltk.download('punkt')

nltk.download('stopwords')

premise = "The cat is sitting on the mat."

hypothesis = "A cat is on the mat."

stop_words = set(stopwords.words('english'))

def preprocess(text):

    tokens = word_tokenize(text.lower())  # tokenize and lowercase

    return [word for word in tokens if word.isalpha() and word not in stop_words]  # remove
punctuation and stopwords

premise_tokens = preprocess(premise)

hypothesis_tokens = preprocess(hypothesis)

common_words = set(hypothesis_tokens).intersection(set(premise_tokens))

if len(common_words) == len(set(hypothesis_tokens)):

    print("Entailment")

else:

    print("Not Entailment")
```

b. Natural Language Inference with BERT (GOOGLE COLLAB)

```
from transformers import pipeline

nli_model = pipeline("text-classification", model="roberta-large-mnli")

premise = "A man is playing a guitar."
```

```
hypothesis = "A person is playing an instrument."
result = nli_model(f"{premise} </s></s> {hypothesis}")
print(result)
```

c. Sentence Pair Classification using Siamese Networks (GOOGLE COLLAB)

```
from sentence_transformers import SentenceTransformer, util
model = SentenceTransformer('paraphrase-MiniLM-L6-v2')
sentence1 = "A man is eating food."
sentence2 = "A person is consuming a meal."
embedding1 = model.encode(sentence1, convert_to_tensor=True)
embedding2 = model.encode(sentence2, convert_to_tensor=True)
similarity = util.pytorch_cos_sim(embedding1, embedding2)
print(f"Cosine Similarity: {similarity.item()}")
if similarity.item() > 0.65:
   print("Entailment")
else:
   print("Not Entailment")
```

WEEK-9

Word and Sentence Embedding

a. Basic Word Embeddings with TF-IDF

```
from sklearn.feature_extraction.text import TfidfVectorizer


corpus = [
   "Deep learning is fun",
   "Word embeddings can be learned",
   "TF-IDF captures word importance",
   "Embeddings represent words as vectors"
]
```

```
vectorizer = TfidfVectorizer()

X = vectorizer.fit_transform(corpus)


features = vectorizer.get_feature_names_out()

tfidf_matrix = X.toarray()


print("TF-IDF Feature Names:")

print(features)

print("\nTF-IDF Matrix:")

print(tfidf_matrix)
```

b. Generating Word Embeddings using Word2Vec and GloVe

```
from gensim.utils import simple_preprocess

from gensim.models import Word2Vec, KeyedVectors

from gensim.scripts.glove2word2vec import glove2word2vec

corpus = [

    "Natural language processing enables computers to understand human language.",

    "Word embeddings capture semantic relationships between words in a vector space.",

    "Deep learning techniques such as Word2Vec and GloVe are widely used in NLP applications.",

    "This is a sample document for generating word embeddings.",

    "Another example document is provided for demonstration purposes."

]

tokenized_corpus = [simple_preprocess(sentence) for sentence in corpus]

print("Sample tokenized sentences:\n", tokenized_corpus)

print("\nTraining Word2Vec model...")

w2v_model = Word2Vec(sentences=tokenized_corpus, vector_size=100, window=5, sg=1,
min_count=1, workers=4)


print("\nWord2Vec: Similar words to 'document'")

print(w2v_model.wv.most_similar("document", topn=5))

print("\nLoading GloVe embeddings...")

glove_file = "glove.6B.100d.txt"
```

```
glove2word2vec(glove_file, "glove.6B.100d.word2vec.txt")
glove_model = KeyedVectors.load_word2vec_format("glove.6B.100d.word2vec.txt", binary=False)
print("\nGloVe: Similar words to 'document'")
print(glove_model.most_similar("document", topn=5))
```

c. Sentence Embeddings with Universal Sentence Encoder(GOOGLE COLLAB)

```
import tensorflow as tf
import tensorflow_hub as hub
import numpy as np
sentences = [
    "This is a sentence.",
    "Another example sentence.",
    "Machine learning is fascinating.",
    "I love natural language processing.",
    "The sky is blue today."
]
print("Loading Universal Sentence Encoder model...")
embed = hub.load("https://tfhub.dev/google/universal-sentence-encoder/4")
print("Model loaded!")
sentence_embeddings = embed(sentences)
print(f"Sentence Embeddings Shape: {sentence_embeddings.shape}\n")
for i, sentence in enumerate(sentences):
    print(f"Sentence: {sentence}")
    print(f"Embedding vector (first 5 values): {sentence_embeddings[i][:5].numpy()}\n")
```

WEEK-10

Question Answering

a. Basic Q&A System using Keyword Matching

```
qa_dataset = {
    "what is python?": "Python is a high-level programming language.",
    "who developed python?": "Python was developed by Guido van Rossum.",
    "what is machine learning?": "Machine Learning is a field of AI that enables computers to learn from data.",
    "what is ai?": "AI (Artificial Intelligence) is the simulation of human intelligence by machines.",
    "what is data science?": "Data Science is the field that uses scientific methods, processes, and algorithms to extract knowledge from data."
}
import json
with open("qa_dataset.json", "w") as f:
    json.dump(qa_dataset, f)
with open("qa_dataset.json", "r") as f:
    qa_data = json.load(f)
def find_answer(question, qa_data):
    question = question.lower()
    for q, a in qa_data.items():
        if all(keyword in q.lower() for keyword in question.split()):
            return a
    return "Sorry, I don't know the answer to that question."
def main():
    print("Welcome to the Q&A System (type 'exit' to quit)")
    while True:
        user_question = input("Ask a question: ")
        if user_question.lower() in ['exit', 'quit']:
            print("Goodbye!")
            break
        answer = find_answer(user_question, qa_data)
        print("Answer:", answer)
if __name__ == "__main__":
    main()
```

b. Building a Q&A System with BERT (Google Collab)

```
from transformers import pipeline
```

```
qa_pipeline = pipeline(
   "question-answering",
   model="bert-large-uncased-whole-word-masking-finetuned-squad",
   tokenizer="bert-large-uncased-whole-word-masking-finetuned-squad"
)
context = "BERT is a pre-trained transformer model for natural language understanding."
question = "What is BERT?"
result = qa_pipeline(question=question, context=context)
print("Question:", question)
print("Answer:", result['answer'])
print("Score:", result['score'])
```

c. Question Answering on SQuAD Dataset using Transformers (GOOGLE COLLAB)

```
import os
os.environ["WANDB_DISABLED"] = "true"
!pip install transformers torch gradio --quiet
from transformers import pipeline
import gradio as gr
qa = pipeline("question-answering", model="distilbert-base-cased-distilled-squad")
def answer_question(context, question):
 result = qa(question=question, context=context)
 return result["answer"]
iface = gr.Interface(
 fn=answer_question,
 inputs=[
 gr.Textbox(label="Context", placeholder="Enter the context paragraph here..."),
 gr.Textbox(label="Question", placeholder="Enter your question here...")
 ],
 outputs=gr.Textbox(label="Answer"),
 title="Question Answering with DistilBERT",
 description="Enter a context and a question, and the model will return the answer.")
```

```
iface.launch()
```

WEEK-11

Machine Translation

a. Basic Machine Translation using Rule-Based Methods

```
dictionary = {
    'hello': 'bonjour',
    'world': 'monde',
    'my': 'mon',
    'name': 'nom',
    'is': 'est'
}
grammar_rules = {
    'SVO': ['subject', 'verb', 'object']
}
def translate(sentence):
    words = sentence.lower().split()
    translated_words = [dictionary.get(word, word) for word in words]
    return ' '.join(translated_words)
sentence = "Hello world"
print(translate(sentence))
```

b. English to French Translation using Seq2Seq with Attention

```
import tensorflow as tf
from tensorflow.keras.layers import Embedding, LSTM, Dense, Attention
import numpy as np


# --- Sample Dataset ---
eng = ["i like apples", "they do not like grapefruit"]
fr  = ["j aime les pommes", "ils n aiment pas le pamplemousse"]
```

```
eng_vocab = {w:i+1 for i,w in enumerate(set(" ".join(eng).split()))}
fr_vocab  = {w:i+1 for i,w in enumerate(set(" ".join(fr).split()))}
inv_fr = {i:w for w,i in fr_vocab.items()}

def encode(s, vocab, max_len=6):
    return [vocab[w] for w in s.split()]+[0]*(max_len-len(s.split()))

X = np.array([encode(s,eng_vocab) for s in eng])
Y = np.array([encode(s,fr_vocab) for s in fr])

# --- Seq2Seq with Attention ---
class Seq2Seq(tf.keras.Model):
    def __init__(self, in_vocab, out_vocab, emb=16, units=32):
        super().__init__()
        self.enc_emb = Embedding(in_vocab+1, emb)
        self.enc_lstm = LSTM(units, return_sequences=True, return_state=True)
        self.dec_emb = Embedding(out_vocab+1, emb)
        self.dec_lstm = LSTM(units, return_sequences=True, return_state=True)
        self.att, self.fc = Attention(), Dense(out_vocab+1, activation="softmax")

    def call(self, inputs):
        x, y = inputs  # unpack
        enc_out,h,c = self.enc_lstm(self.enc_emb(x))
        dec_out,_,_ = self.dec_lstm(self.dec_emb(y), initial_state=[h,c])
        ctx = self.att([dec_out, enc_out])
        return self.fc(tf.concat([dec_out, ctx], -1))

model = Seq2Seq(len(eng_vocab), len(fr_vocab))
model.compile(optimizer="adam", loss="sparse_categorical_crossentropy")
model.fit([X,Y], np.expand_dims(Y,-1), epochs=100, verbose=0)
```

```
# --- Translation ---
test = np.array([encode("they do not like grapefruit", eng_vocab)])
start = np.array([encode("ils", fr_vocab)])  # seed word
pred = np.argmax(model.predict([test,start])[0],-1)
print("Translation:", " ".join(inv_fr.get(i,"") for i in pred))
```

c. Neural Machine Translation with Transformers (English to German)

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch
tokenizer = AutoTokenizer.from_pretrained("microsoft/DialoGPT-medium")
model = AutoModelForCausalLM.from_pretrained("microsoft/DialoGPT-medium")
chat_history_ids = None
print(" Conversational AI Chatbot (type 'quit' to exit)")
while True:
    user_input = input("You: ")
    if user_input.lower() in ["quit", "exit"]:
        print("Bot: Goodbye! 👋")
        break
    new_input_ids = tokenizer.encode(user_input + tokenizer.eos_token, return_tensors='pt')
    if chat_history_ids is not None:
        bot_input_ids = torch.cat([chat_history_ids, new_input_ids], dim=-1)
    else:
        bot_input_ids = new_input_ids
    chat_history_ids = model.generate(
        bot_input_ids,
        max_length=1000,
        pad_token_id=tokenizer.eos_token_id,
        do_sample=True,
        temperature=0.7,
        top_k=50,
        top_p=0.95
    )
    bot_output = tokenizer.decode(
        chat_history_ids[:, bot_input_ids.shape[-1]:][0],
        skip_special_tokens=True
    )
    print("Bot:", bot_output)
```

WEEK-12

Dialogue System

a. Basic Rule-Based Chatbot using Python NLTK

```
import nltk
from nltk.chat.util import Chat, reflections
pairs = [
    (r"my name is (.*)", ["Hello %1, how are you today?"]),
    (r"hi|hey|hello", ["Hello!", "Hey there!"]),
    (r"what is your name?", ["I am a bot created by [Your Name]."]),
    (r"how are you?", ["I'm doing good. How about you?"]),
    (r"sorry (.*)", ["No problem!", "It's okay.", "You don't need to be sorry."]),
    (r"quit", ["Bye! Take care."]),
]
def chatbot():
    print("Hi, I'm the chatbot you created. Type 'quit' to exit.")
    chat = Chat(pairs, reflections)
    chat.converse()
if __name__ == "__main__":
    chatbot()
```

b. Building a Chatbot using Seq2Seq Models (google collab)

```
import tensorflow as tf
import numpy as np
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, LSTM, Dense, Embedding


# --- Toy dataset ---

inputs = ["Hi", "who are you?"]

targets = ["Hello!", "I'm good!"]

chars = sorted(list(set(" ".join(inputs + targets))))

c2i = {c:i for i,c in enumerate(chars)}

i2c = {i:c for c,i in c2i.items()}

max_in, max_out = max(len(s) for s in inputs), max(len(s) for s in targets)


def encode(s, l): return [c2i[c] for c in s] + [0]*(l-len(s))

X_enc = np.array([encode(s, max_in) for s in inputs])

X_dec = np.array([encode(s, max_out) for s in targets])

Y = np.expand_dims(X_dec, -1)


# --- Seq2Seq Model ---

latent=32

enc_inp = Input((max_in,))

enc_emb = Embedding(len(chars), latent)(enc_inp)

enc_out, h, c = LSTM(latent, return_state=True)(enc_emb)


dec_inp = Input((max_out,))

dec_emb = Embedding(len(chars), latent)(dec_inp)

dec_out, _, _ = LSTM(latent, return_sequences=True, return_state=True)(dec_emb, initial_state=[h,
c])

dec_out = Dense(len(chars), activation='softmax')(dec_out)


model = Model([enc_inp, dec_inp], dec_out)

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy')

model.fit([X_enc, X_dec], Y, epochs=300, verbose=0)
```

```
# --- Inference ---
def respond(s):
    seq = np.array([encode(s, max_in)])
    dec = np.zeros((1, max_out))
    for i in range(max_out):
        p = model.predict([seq, dec], verbose=0)[0]
        dec[0,i] = np.argmax(p[i])
    return "".join([i2c[int(i)] for i in dec[0]]).strip()


# Test
print("Input: How are you?")
print("Bot:", respond("How are you?"))
```

c. Conversational AI with Transformer-based Models (google Coolab)

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch
tokenizer = AutoTokenizer.from_pretrained("microsoft/DialoGPT-medium")
model = AutoModelForCausalLM.from_pretrained("microsoft/DialoGPT-medium")
chat_history_ids = None
print(" Conversational AI Chatbot (type 'quit' to exit)")
while True:
    user_input = input("You: ")
    if user_input.lower() in ["quit", "exit"]:
        print("Bot: Goodbye! 👋 ")
        break
    new_input_ids = tokenizer.encode(user_input + tokenizer.eos_token, return_tensors='pt')
    if chat_history_ids is not None:
        bot_input_ids = torch.cat([chat_history_ids, new_input_ids], dim=-1)
    else:
        bot_input_ids = new_input_ids
    chat_history_ids = model.generate(
        bot_input_ids,
        max_length=1000,
        pad_token_id=tokenizer.eos_token_id,
        do_sample=True,
        temperature=0.7,
        top_k=50,
        top_p=0.95
    )
    bot_output = tokenizer.decode(
        chat_history_ids[:, bot_input_ids.shape[-1]:][0],
        skip_special_tokens=True
    )
    print("Bot:", bot_output)
```