National College of Ireland

A Smart Cloud – Based Document Search Engine for Query Retrieval Using Large Learning Models (LLM's)

MSc Research Project
MSc in Cloud Computing

Rohith Konan Ravi
Student ID: 23195983

School of Computing
National College of Ireland

Supervisor: Abubakr Siddig

Student ID: 23195983

Programme: MSc in Cloud computing **Year:** 2024.

Module: MSc Research Project

Supervisor: Abubakr Siddig

Submission Due Date: 12/12/2024

Project Title: A Smart Cloud – Based Document Search Engine for Fast Query Retrieval Using Large Learning Models (LLM's)

Word Count: 1163 **Page Count :** 18

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: Rohith Konan Ravi

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):	

Introduction:

The configuration manual is the manual for the step by step instruction to run the implementation of cloud based document retrieval engine , this mainly explains the software tools and the methods which are mainly used in the implementation of the research , it mainly explains the code which we employed while training the model with different datasets to achieve the goal, and main aim of this manual is to help researchers and practitioners identifying and following the steps and resources which will be helpful for further research and development.

System requirements

All the implementation and research performed on the MacBook M1 air laptop with Mac operating system With M1 chip

1. Hardware: Quad-Core CPU, 16GB RAM, 50GB storage, NVIDIA GPU with CUDA support.
2. Software: Python 3.8+, OS (Windows 10/11, macOS, or Linux), libraries (`pandas`, `torch`, `transformers`, etc.).
3. Dependencies: Pre-trained models (`facebook/bart-large`,), ~10GB storage for models/embeddings.
4. Tools: Jupyter/VSCode, internet for downloads, cloud (Colab/AWS).

```
import tkinter as tk
from tkinter import messagebox
import re
import pickle
import pandas as pd
from sentence_transformers import SentenceTransformer, CrossEncoder, util
import string
from tqdm.autonotebook import tqdm
import nltk

#import json
#import gzip
#import os
import torch
#import numpy as np
#from rank_bm25 import BM25Okapi
```

- Imports essential libraries for GUI (Tkinter), data handling (Pandas), and NLP tasks (SentenceTransformers, NLTK).

```
"""here one more function for the pdf extraction will be there"""

def encode_passages(df, bi_encoder):
    bi_encoder.max_seq_length = 512#256
    top_k = 128
    passages_with_context = []
    for i, row in df.iterrows():
        pdf_name_f = row['pdf_name']#.split('/')[-1]
        passage_context = {
            'pdf_name': pdf_name_f,
            'page_num': row['page_num'],
            'para_num': row['para_num']
        }
        passages_with_context.append((row['documents'], passage_context))
    corpus_embeddings = bi_encoder.encode([p[0] for p in passages_with_context], convert_to_tensor=True, show_progress_bar=True) ###
    passages = [p[0] for p in passages_with_context]
    return corpus_embeddings, passages, passages_with_context
```

- Extracts passages from a DataFrame and encodes them into embeddings using a bi-encoder model for semantic similarity tasks.

```
def search(query, passages, passages_with_context, bi_encoder, cross_encoder, corpus_embeddings):
    question_embedding = bi_encoder.encode(query, convert_to_tensor=True)
    # question_embedding = question_embedding.cuda()
    hits = util.semantic_search(question_embedding, corpus_embeddings, top_k=top_k)
    hits = hits[0]
    cross_inp = [[query, passages[hit['corpus_id']]] for hit in hits]
    cross_scores = cross_encoder.predict(cross_inp)

    for idx in range(len(cross_scores)):
        hits[idx]['cross-score'] = cross_scores[idx]
    print("\n-------------------------\n")
    print("Top-3 Cross-Encoder Re-ranker hits")
    hits = sorted(hits, key=lambda x: x['cross-score'], reverse=True)
    results_lstt = []
    for hit in hits[0:10]:
      passage_index = hit['corpus_id']
      passage_text = passages_with_context[passage_index][0].replace("\n", " ")
      passage_context = passages_with_context[passage_index][1]
      pdf_name = passage_context['pdf_name']
      page_num = passage_context['page_num']
      print("[PDF: {}, Page: {}]\t{}".format(pdf_name, page_num, passage_text))
      #results_lstt.append({"Pdf_name":pdf_name, "Page_name": page_num, "Content":passage_text})
    return results_lstt
```

- Performs semantic search by encoding a query, retrieving relevant passages, re-ranking them with a cross-encoder, and returning the top results.

```
BASE_DRIVE= 'C:\\Users\\nimai\\OneDrive\\Documents\\code testing\\AbleTech\\Testing\\'
csv_path = BASE_DRIVE + "aleltech_mega.csv"
df = pd.read_csv(csv_path, encoding='latin-1')#albatross.csv')

df.documents = df['documents'].astype(str)
df.documents = df['documents'].apply(lambda x: x.strip())
df1 = df
```

- Reads a CSV file, processes the `documents` column to remove whitespace, and converts it to string type.

```
# Load the biencoder model
model_path1 = BASE_DRIVE + 'biencoder.pkl'
#bi_encoder = SentenceTransformer('efederici/mmarco-sentence-BERTino')
#with open(model_path1, 'wb') as f:
#    pickle.dump(bi_encoder,f)


with open(model_path1, 'rb') as f:
    loaded_model_bi_encoder = pickle.load(f)

# Load the crossencoder model
model_path = BASE_DRIVE + 'crossencoder.pkl'
with open(model_path, 'rb') as file:
    loaded_model_cross_encoder = pickle.load(file)
```

- Loads pre-trained bi-encoder and cross-encoder models from pickle files for use in semantic search.

```
#corpus_embeddings1, passages, passages_with_context = encode_passages(df1, loaded_model_bi_encoder)

embed_path = BASE_DRIVE+'embedding.pkl'
with open(embed_path,'rb') as fp:
   corpus_embeddings1 = pickle.load(fp)

with open(BASE_DRIVE+'passages.pkl','rb') as fp:
   passages = pickle.load(fp)

with open(BASE_DRIVE+'passages_with_context.pkl','rb') as fp:
   passages_with_context = pickle.load(fp)

loaded_model_bi_encoder.max_seq_length = 512#256
top_k = 128

Run Cell | Run Below | Debug Cell
#%%
"""Search the Questions"""
query = input("Enter Your Question: ")
results = search(query, passages, passages_with_context, loaded_model_bi_encoder, loaded_model_cross_encoder, corpus_embeddings1)
print(results)
```

- Loads precomputed embeddings, passages, and context from files. Takes user input for a query and performs semantic search, printing the top results.

recursive summarization fine-tuning

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

- Mounts your Google Drive to the Colab environment, enabling access to files stored in your Drive.

```
!pip install datasets
!pip install rouge_score
# !pip install accelerate -U
```

- Installs the `datasets` library for handling datasets and `rouge_score` for evaluating text summarization. The commented line suggests upgrading the `accelerate` library if needed.

```
from datasets import load_dataset
import nltk
import json
from nltk.tokenize import word_tokenize
import pickle
from transformers import pipeline
from datasets import load_metric
nltk.download('punkt')
!pip install huggingface-hub
```

- Loads essential libraries for dataset handling, tokenization (`nltk`), and model pipelines (`transformers`). `punkt` is downloaded for text tokenization, and the `huggingface-hub` library is installed for accessing Hugging Face tools.

```
import numpy as np
import transformers
from rouge_score import rouge_scorer
from datasets import load_dataset
from tqdm.auto import tqdm

import torch
```

- Imports libraries for numerical operations (`numpy`), model handling (`transformers`), evaluation metrics (`rouge_scorer`), progress bars (`tqdm`), and PyTorch (`torch`).

```
dataset1 = load_dataset("abisee/cnn_dailymail" , "3.0.0")
```

- Downloads the CNN/DailyMail dataset version 3.0.0 for summarization tasks using the `datasets` library.

```
def clean_dataset(data):

  cleaned_data = {}
  for split, data_points in data.items():
    cleaned_data[split] = []
    for data_point in data_points:
      if count_words(data_point['article']) >= 50 and count_words(data_point['highlights']) >= 5:
        cleaned_data[split].append(data_point)

  return cleaned_data

cleaned_dataset = clean_dataset(dataset1)


print("Dataset cleaning complete!")
```

- Filters the dataset by ensuring articles have at least 50 words and summaries ("highlights") have at least 5 words. Returns a cleaned dataset.

```
cleaned_dataset['train'][0]
```

- Accesses the first record from the `train` split of the cleaned dataset.

```
with open('/content/drive/MyDrive/Sumbot /datasets/cnn_dailymail_preoccesd.pkl', 'wb') as f:
    pickle.dump(cleaned_dataset, f)
```

- Opens a file in write-binary mode to save the `cleaned_dataset`.
- The `pickle.dump` function serializes the dataset and writes it to a `.pkl` file for future use.

```
cnn = pickle.load(open('/content/drive/MyDrive/Sumbot /datasets/cnn_dailymail_preoccesd.pkl', 'rb'))
```

- This snippet uses the `pickle` library to load a preprocessed CNN/DailyMail dataset (`cnn_dailymail_preccesd.pkl`) in binary read mode (`rb`) from the specified file path.
- The loaded dataset is assigned to the variable `cnn` for further use in summarization tasks.

The Second Dataset XSum

```
dataset = load_dataset("EdinburghNLP/xsum")
```

```
dict
```

```
def clean_dataset(data):

  cleaned_data = {}
  for split, data_points in data.items():
    cleaned_data[split] = []
    for data_point in data_points:
      if count_words(data_point['document']) >= 50 and count_words(data_point['summary']) >= 5:
        cleaned_data[split].append(data_point)

  return cleaned_data

cleaned_dataset = clean_dataset(dataset)

cleaned_train_data = cleaned_dataset['train']
cleaned_test_data = cleaned_dataset['test']
cleaned_validation_data = cleaned_dataset['validation']

print("Dataset cleaning complete!")
```

- Loads the XSum dataset for summarization tasks using the `datasets` library.
- Filters the dataset to retain only documents with at least 50 words and summaries with at least 5 words.
- Segregates the cleaned data into train, test, and validation splits.

```
with open('/content/drive/MyDrive/Sumbot /datasets/xsum.pkl', 'wb') as f:
    pickle.dump(dataset, f)
```

- Serializes the `dataset` and saves it as a `.pkl` file.

```
xsum = pickle.load(open('/content/drive/MyDrive/Sumbot /datasets/xsum.pkl', 'rb'))
```

- Loads the serialized `xsum` dataset from the `.pkl` file for use in the code.

The Third Dataset

```
def jsonl_to_dict(filename):
  """
  Converts a JSON Lines file to a dictionary.

  Args:
      filename (str): The path to the JSON Lines file.

  Returns:
      dict: A dictionary containing each JSON object from the file.
  """
  data = {}
  with open(filename, 'r') as f:
    for i, line in enumerate(f):
      item = json.loads(line.strip())
      data[i] = item  # Use an index or unique identifier as the key
  return data
data_dict = jsonl_to_dict(filename = "/content/drive/MyDrive/Sumbot /datasets/wikisum/WikiSumDataset/WikiSumDataset.jsonl")
```

- Converts a JSON Lines file into a Python dictionary.
- Each line of the file is treated as a JSON object and added to the dictionary with an index as the key.

```
new_dict = {
    "train": [],
    "test": [],
    "validation": []
}

for i  in tqdm(range (len(data_dict))):
  if count_words(data_dict[i]['summary']) >= 5 and count_words(data_dict[i]['article']) >= 50:
    temp_dict = {'article' : data_dict[i]['article'] , 'summary' : data_dict[i]['summary']}
    if data_dict[i]['fold'] == 'train' or data_dict[i]['fold'] == 'test':
      new_dict[data_dict[i]['fold']].append(temp_dict)
    else:
      new_dict['validation'].append(temp_dict)
```

- Segregates data into `train`, `test`, and `validation` splits based on the `fold` field in `data_dict`.
- Retains only records with articles having at least 50 words and summaries with at least 5 words.

```
with open('/content/drive/MyDrive/Sumbot /datasets/wikihow_preoccesd.pkl', 'wb') as f:  # 'wb' for write binary
    pickle.dump(new_dict, f)
```

- Saves the `new_dict` containing preprocessed WikiHow data as a `.pkl` file for future use.

```
import pickle
wikihow = pickle.load(open('/content/drive/MyDrive/intern 7/dataset/wikihow_preoccesd.pkl', 'rb'))
```

- Loads the previously saved WikiHow dataset (`wikihow_preprocessed.pkl`) using `pickle`.

Model 1

```
from tqdm import tqdm
```

```
pip install torch
```

- Imports `tqdm` for progress bars during loops and installs `torch` for PyTorch-based operations.

```
wikihow['test'][0]
```

- Accesses the first data record in the `test` split of the `wikihow` dataset.

```
wikihow = wikihow["test"]
```

- Assigns the `test` split of the `wikihow` dataset to the `wikihow` variable.

```
wikihow[0]
```

- Accesses the first record from the current `wikihow` dataset variable, which is now likely the test split.

```python
device = "cuda" if torch.cuda.is_available() else "cpu"

model_name = 'facebook/bart-large-cnn'
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name).to(device)
summarizer = pipeline("summarization", model=model, tokenizer=tokenizer, device=0 if device == "cuda" else -1  )

rouge = rouge_scorer.RougeScorer(['rouge1', 'rougeL', 'rougeLsum'], use_stemmer=True)

def summarize_in_chunks(article, summarizer, tokenizer, chunk_size=512, overlap=25):
    inputs = tokenizer(article, return_tensors="pt", truncation=False)
    input_ids = inputs['input_ids'][0]
    chunks = [input_ids[i:i + chunk_size] for i in range(0, len(input_ids), chunk_size - overlap) ]
    summaries = []
    for chunk in chunks:
        chunk_text = tokenizer.decode(chunk, skip_special_tokens=True)
        summary = summarizer(chunk_text, max_length=127, min_length=4, do_sample=False)[0]['summary_text']
        print(summary)
        summaries.append(summary)

    combined_summary = " ".join(summaries)
    final_summary = summarizer(combined_summary, max_length=512, min_length=4, do_sample=False)[0]['summary_text']
    return final_summary
sample_size = 1
results = []
for i in tqdm(np.random.randint(0, len(wikihow), size=sample_size).tolist()):
    article = wikihow[i]["article"]
    print(article)
    article = article.replace("\n", "")
    reference_summary = wikihow[i]["summary"]
    generated_summary = summarize_in_chunks(article, summarizer, tokenizer)
    print("genereted_summary")
    print(generated_summary)
    print(reference_summary)
    rouge_scores = rouge.score(generated_summary, reference_summary)
    results.append(rouge_scores)
print(results)
```

This code:

1. **Device and Model Initialization**:
    - Sets computation device (`cuda` or `cpu`) and loads the `facebook/bart-large-cnn` model with tokenizer.
2. **Pipeline Setup**:
    - Creates a summarization pipeline and initializes the ROUGE scorer.
3. **Summarization in Chunks**:
    - Splits large text into chunks for summarization and combines chunk summaries into a final summary.
4. **Processing Articles**:
    - Randomly selects articles from the dataset, generates summaries, and evaluates them using ROUGE scores.
5. **Result Storage**:
    - Appends ROUGE scores of each summary to a list and prints the results.

```
import json
with open('/content/drive/MyDrive/bart-large-cnn_wikihow.json', 'w') as f:
    json.dump(results, f, indent=4)
```

```
import numpy as np
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer, pipeline
from rouge_score import rouge_scorer
from datasets import load_dataset
from tqdm.auto import tqdm
import torch
from transformers import AutoModelForSeq2SeqLM, AutoTokenizer, pipeline
from rouge_score import rouge_scorer
import torch
```

- Saves the summarization results and evaluation metrics as a formatted JSON file in Google Drive.
- Imports essential libraries for text summarization, evaluation, dataset handling, and hardware acceleration.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

model_name = 'facebook/bart-large-cnn'
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name).to(device)
rouge = rouge_scorer.RougeScorer(['rouge1', 'rougeL', 'rougeLsum'], use_stemmer=True)

def summarize(text, maxSummarylength=500):
    global model, tokenizer
    summarizer = pipeline("summarization", model=model, tokenizer=tokenizer, device=0 if device.type == "cuda" else -1)
    summary = summarizer(text, max_length=maxSummarylength, min_length=int(maxSummarylength/5),
                         length_penalty=10.0, num_beams=4, early_stopping=True)[0]['summary_text']

    return summary
```

- Configures the Torch device and loads the BART-large CNN model and tokenizer.
- Performs text summarization using the model pipeline with fine-tuned parameters like `num_beams`, `length_penalty`, and `max_length`.

```python
def split_text_into_pieces(text,
                           max_tokens=900,
                           overlapPercent=0):
    tokens = tokenizer.tokenize(text)

    overlap_tokens = int(max_tokens * overlapPercent / 100)


    pieces = [tokens[i:i + max_tokens]
              for i in range(0, len(tokens),
                             max_tokens - overlap_tokens)]

    text_pieces = [tokenizer.decode(
        tokenizer.convert_tokens_to_ids(piece),
        skip_special_tokens=True) for piece in pieces]

    return text_pieces
```

- Splits a long text into manageable chunks based on `max_tokens` with optional overlapping tokens.
- Converts tokens back into text for further processing.

```python
def recursive_summarize(text, max_length=200, recursionLevel=0):
    recursionLevel=recursionLevel+1
    tokens = tokenizer.tokenize(text)
    expectedCountOfChunks = len(tokens)/max_length
    max_length=int(len(tokens)/expectedCountOfChunks)+2
    pieces = split_text_into_pieces(text, max_tokens=max_length)
    summaries=[]
    k=0
    for k in range(0, len(pieces)):
        piece=pieces[k]
        summary =summarize(piece, maxSummarylength=int(max_length//3*2))
        summaries.append(summary)
    concatenated_summary = ' '.join(summaries)

    tokens = tokenizer.tokenize(concatenated_summary)

    if len(tokens) > max_length:

        return recursive_summarize(concatenated_summary,
                                   max_length=max_length,
                                   recursionLevel=recursionLevel)
    else:
        final_summary=concatenated_summary
        if len(pieces)>1:
            final_summary = summarize(concatenated_summary,
                                      maxSummarylength=max_length)
        return final_summary
```

1.

- Recursively processes large text by splitting it into smaller pieces using the `split_text_into_pieces` function.
- Summarizes chunks, concatenates them, and re-summarizes if the result exceeds the `max_length`.
- Ensures iterative summarization for long texts until the desired length is met.

```
wikihow = wikihow['test']
```

+ Code + Text

```
wikihow = wikihow['test'].sample_size = 1
results = []
for i in tqdm(np.random.randint(0, len(wikihow), size=sample_size).tolist()):
    article = wikihow[i]["article"]
    article = article.replace("\n", "")
    reference_summary = wikihow[i]["summary"]
    generated_summary = recursive_summarize(article,256)
    rouge_scores = rouge.score(generated_summary, reference_summary)
    results.append(rouge_scores)
print(results)
```

- Processes a random sample of articles from the WikiHow `test` set.
- Generates summaries using the `recursive_summarize` function, evaluates them using ROUGE scores, and appends the results to a list.

```
with open('/content/drive/MyDrive/intern 7/results/bart-large-cnn_wikihow-0percent.json', 'w') as f:
    json.dump(results, f, indent=4)
```

- Serializes and saves the summarization results, including ROUGE scores, into a JSON file with proper formatting.

```
[ ] with open('/content/drive/MyDrive/intern 7/results/bart-large-xsum_wikihow-0percent.json', 'r') as f:
        rouge_scores = json.load(f)
    len(rouge_scores)
```

- Loads precomputed ROUGE scores from a JSON file and calculates the total number of summaries evaluated.

```
import random
wikihow =  random.sample(wikihow['train'], k=10000)
```

```
import pandas as pd
from datasets import Dataset, DatasetDict
from transformers import BartTokenizer, BartForConditionalGeneration, TrainingArguments, Trainer
from transformers.integrations import TensorBoardCallback
import matplotlib.pyplot as plt
import torch
import numpy as np
```

- Randomly samples 10,000 articles from the WikiHow training dataset for analysis or processing.
- Imports essential libraries for dataset handling, tokenization, training, and visualization.

```python
from transformers import BartTokenizer



tokenizer = BartTokenizer.from_pretrained('facebook/bart-large')

def find_articles_with_token_size(wikihow, max_token_size=1024):
    train_data = wikihow.get('train', [])

    filtered_articles = []
    for article in tqdm(train_data):
        tokens = tokenizer(article)['input_ids']
        if len(tokens) <= max_token_size:
            filtered_articles.append(article)

    return filtered_articles

articles_with_acceptable_token_size = find_articles_with_token_size(wikihow, 1200)


len(articles_with_acceptable_token_size)
```

- Filters articles based on the number of tokens, ensuring they meet the model's input limit.

```python
from datasets import Dataset, DatasetDict
import json


file_path = "/content/drive/MyDrive/intern 7/cleaned/filtered_articles.json"

with open(file_path, "r") as json_file:
    train = json.load(json_file)
```

- Loads filtered articles for training from a JSON file.

```python
file_path = "/content/drive/MyDrive/intern 7/cleaned/validation_articles.json"

# Read the JSON file
with open(file_path, "r") as json_file:
    val = json.load(json_file)

train_dataset = Dataset.from_list(train)
val_dataset = Dataset.from_list(val)
```

- Converts loaded JSON files into Hugging Face `Dataset` objects for training and validation.

```python
from transformers import BartTokenizer

tokenizer = BartTokenizer.from_pretrained('facebook/bart-large')

def preprocess_function(examples):
    inputs = examples['article']
    targets = examples['summary']
    model_inputs = tokenizer(inputs, max_length=1024, truncation=True, padding="max_length")

    # Tokenize targets
    with tokenizer.as_target_tokenizer():
        labels = tokenizer(targets, max_length=256, truncation=True, padding="max_length")

    model_inputs['labels'] = labels[ 'input_ids']
    return model_inputs

tokenized_train_dataset = train_dataset.map(preprocess_function, batched=True)
tokenized_val_dataset = val_dataset.map(preprocess_function, batched=True)
```

- Preprocesses articles and summaries by tokenizing them and maps the preprocessing function over the datasets for training and validation.

Finetuning

```python
model = BartForConditionalGeneration.from_pretrained('facebook/bart-large')

seq2seq_data_collator = DataCollatorForSeq2Seq(tokenizer, model=model)

training_args = TrainingArguments(
    output_dir='./results',
    num_train_epochs=3,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir='./logs',
    logging_steps=10,
    evaluation_strategy='steps',
    eval_steps=500,
    gradient_accumulation_steps=16,
)

trainer = Trainer(
    model=model,
    data_collator=seq2seq_data_collator,
    args=training_args,
    train_dataset=tokenized_train_dataset,
    eval_dataset=tokenized_val_dataset,
)

trainer.train()
```

- Initializes the `BartForConditionalGeneration` model and fine-tunes it using the `Trainer` class with specified training arguments (e.g., batch size, epochs, weight decay).

```
trainer.save_model("fineguned_model")
```

- Saves the fine-tuned model to the specified directory for later use.

```
pipe = pipeline ('summarization' , model = 'fineguned_model')

save_directory = '/content/drive/MyDrive/intern 7/cleaned'

# Save the fine-tuned model
model.save_pretrained(save_directory)

# Save the tokenizer
tokenizer.save_pretrained(save_directory)
```

- Sets up a summarization pipeline with the fine-tuned model and saves both the model and tokenizer to the specified directory.

```
load_directory = '/content/drive/MyDrive/intern 7/cleaned'

fine_tuned_model = BartForConditionalGeneration.from_pretrained(load_directory)
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model_name = 'facebook/bart-large'
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name).to(device)
rouge = rouge_scorer.RougeScorer(['rouge1', 'rougeL', 'rougeLsum'], use_stemmer=True)

def summarize(text, maxSummarylength=256):
    global model, tokenizer
    summarizer = pipeline("summarization", model=model, tokenizer=tokenizer, device=0 if device.type == "cuda" else -1)
    summary = summarizer(text, max_length=maxSummarylength, min_length=int(maxSummarylength/5),
                         length_penalty=10.0, num_beams=4, early_stopping=True)[0]['summary_text']

    return summary
```

- Loads the fine-tuned model and defines a summarization function with parameters like `max_length`, `min_length`, and `num_beams` for evaluation.

```python
def split_text_into_pieces(text,
                           max_tokens=1024,
                           overlapPercent=25):
    tokens = tokenizer.tokenize(text)

    overlap_tokens = int(max_tokens * overlapPercent / 100)


    pieces = [tokens[i:i + max_tokens]
              for i in range(0, len(tokens),
                             max_tokens - overlap_tokens)]

    text_pieces = [tokenizer.decode(
        tokenizer.convert_tokens_to_ids(piece),
        skip_special_tokens=True) for piece in pieces]

    return text_pieces
```

- Splits long text into chunks based on token limits with overlapping sections for context preservation.

```python
def recursive_summarize(text, max_length=200, recursionLevel=0):
    recursionLevel=recursionLevel+1
    tokens = tokenizer.tokenize(text)
    expectedCountOfChunks = len(tokens)/max_length
    max_length=int(len(tokens)/expectedCountOfChunks)+2
    pieces = split_text_into_pieces(text, max_tokens=max_length)

    summaries=[]
    k=0
    for k in range(0, len(pieces)):
        piece=pieces[k]

        summary =summarize(piece, maxSummarylength=int(max_length//3*2))
        summaries.append(summary)

    concatenated_summary = ' '.join(summaries)

    tokens = tokenizer.tokenize(concatenated_summary)

    if len(tokens) > max_length:
        return recursive_summarize(concatenated_summary,
                                   max_length=max_length,
                                   recursionLevel=recursionLevel)
    else:
        final_summary=concatenated_summary
        if len(pieces)>1:
            final_summary = summarize(concatenated_summary,
                                      maxSummarylength=max_length)
        return final_summary
```

- Uses recursive logic to generate summaries for large text chunks and re-summarizes concatenated outputs if required.

```
sample_size = 1
results = []
for i in tqdm(t.tolist()):
    article = wikihow[i]["article"]
    article = article.replace("\n", "")
    reference_summary = wikihow[i]["summary"]
    generated_summary = recursive_summarize(article)

    results.append([reference_summary ,generated_summary ])
print(results)
```

- Iterates over a sample of articles, generates summaries using the recursive summarizer, and stores results.

```
] with open('/content/drive/MyDrive/intern 7/results/bart-large-cnn_wikihow-finetuned.json', 'w') as f:
      json.dump(results, f, indent=4)

] finetuned = results

] merged_list = [[finetuned[i][0] , results[i][1] , finetuned[i][1]] for i in range(5)]

  df = pd.DataFrame(merged_list, columns=['Reference', 'bart large', 'finetuned'])
```

- Saves summarization results as a JSON file and creates a DataFrame comparing the reference, base model, and fine-tuned model summaries.