Project III | Business Case: Building a Multimodal AI ChatBot for YouTube Video QA

1. Introduction

1.1 Project Overview

The goal of the project is to build a chatbot that can summarize and translate YouTube videos into text allowing for natural language querying.

The project focuses on the following goals:

- Exploration of the open-source solutions.
- Optimization for deployment on CPU.
- Abstractive question answering.
- Real-life processing of videos provided by a user.

1.2 Data

The Stanford Question Answering Dataset (SQuAD) and ELI5-Category dataset were used for fine-tuning. The former is a collection of question-answer pairs derived from Wikipedia articles. The latter is an English-language dataset of questions and answers gathered from the r/explainlikeimfive subreddit where users ask factual questions requiring paragraph-length or longer answers. The results of fine-tuning are discussed in the dedicated section

1.3 Tools and Technologies

- Python
- Google.Colab, Jupyter Notebook
- Docker, PuTTY, WinSCP
- Libraries and Frameworks: tensorflow, NumPy, Scikit-Learn, Matplotlib, Seaborn, Flask, yt-dlp, youtube_transcript_api, Langchain, Pandas, Sentence-Transformers, Pinecone, PyTorch, Transformers (Hugging Face), NLTK, PEFT (Parameter-Efficient Fine-Tuning), Tqdm.
- Additional tools and libraries: dotenv, TfidfVectorizer, ConversationBufferMemory, HuggingFacePipeline

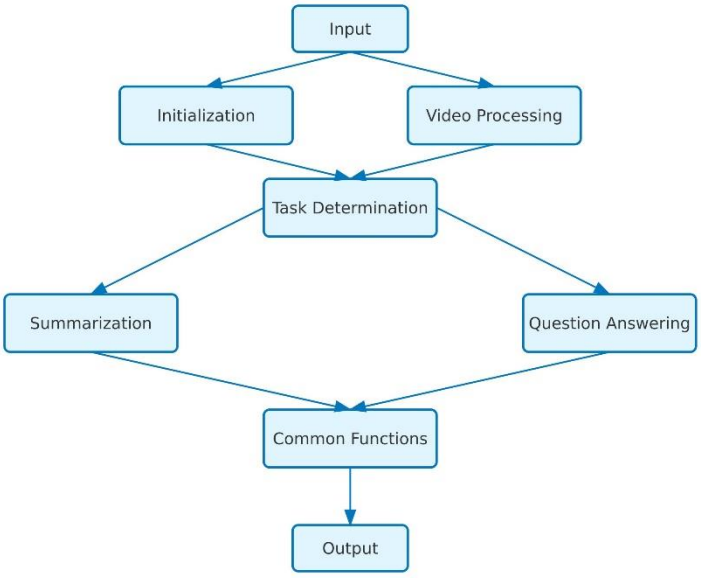
2. App

2.1 App overview

An app was developed for summarization and abstractive question-answering (See Fig. 1). The app has structured and modular architecture. Key aspects of the implementation include:

1. **Initialization**: Efficient model loading and index clearing for a clean start.
2. **Video Processing**: Robust extraction and chunking of video content.
3. **Task Determination**: Simple yet effective classification of user inputs.
4. **Summarization**: A multi-step process involving context retrieval, abstractive summarization, and post-processing for high-quality summaries.
5. **Question-Answering**: Utilizes hybrid context retrieval and relevance checking for accurate answers.
6. **Common Functions**: Shared utilities for text cleaning and formatting, ensuring consistent output quality.

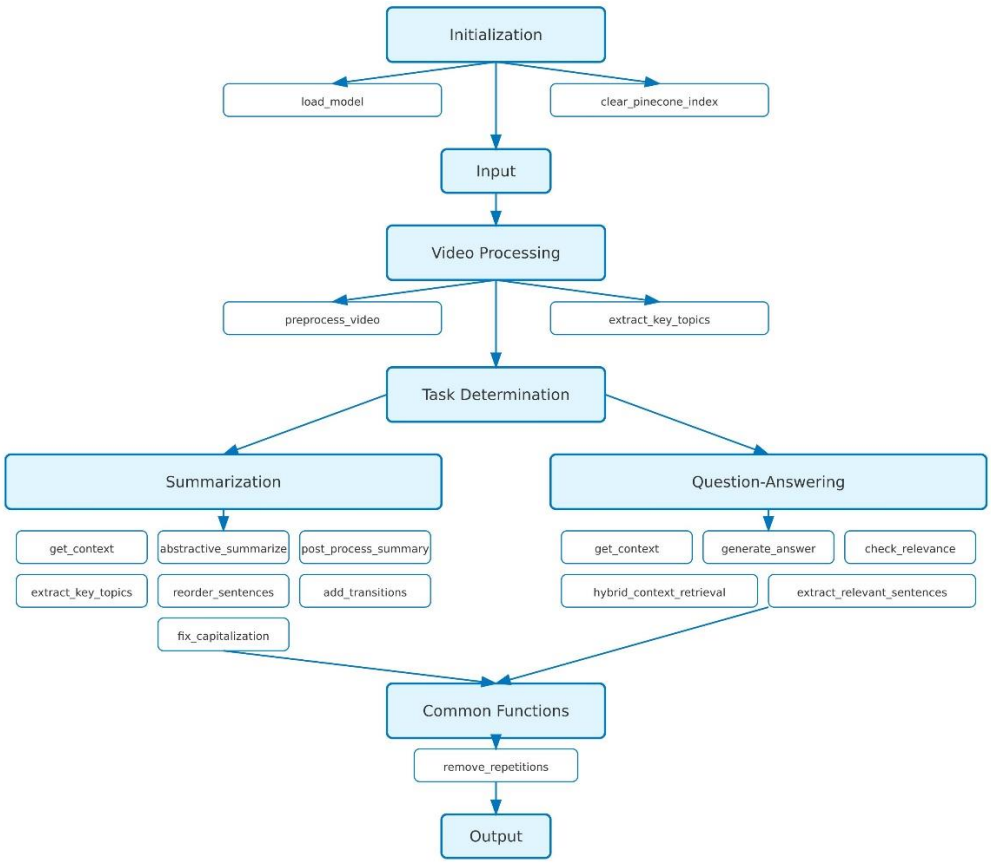
Figure 1. Simplified structure of the app.



Source: own elaboration.

The architecture aims to achieve a balance between leveraging pre-trained models and custom processing steps, as well maintains consideration for both efficiency (e.g., in context retrieval) and output quality (through various post-processing steps). Various features were added to enhance the app's performance (See Fig. 2). These features and their functionality are discussed in the next subsection.

Figure 2. Detailed structure of the app.



Source: own elaboration.

2.2 App functions and components

2.2.1 Initialization

2.2.1.1 load_model()

Purpose: Initializes the model used for text generation tasks.

Explanation: By loading the model at the start, we ensure it's ready for use throughout the application, improving response times. The function uses the Hugging Face Transformers library to load a pre-trained model and tokenizer. The specific model (HF_MODEL_NAME) is defined as a global variable, allowing for easy switching between different models if needed.

Code Snippet:

```
def load_model():
    print(f"Loading model from Hugging Face: {HF_MODEL_NAME}")
    model = AutoModelForSeq2SeqLM.from_pretrained(HF_MODEL_NAME)
    tokenizer = AutoTokenizer.from_pretrained(HF_MODEL_NAME)
    print("Successfully loaded Hugging Face model")
    return model, tokenizer
```

I selected VBlagoje/BART-LFQA model for the question answering tasks for the following reasons:

1. Advanced Question-Answering Capabilities: the model is designed for long-form question answering making it highly suitable for chatbot applications that require generating detailed and informative responses.

2. State-of-the-Art Language Model: Based on the BART (Bidirectional and Auto-Regressive Transformers) architecture, this model leverages the strengths of both bidirectional and auto-regressive transformers.

3. Pre-trained on Large Datasets: The model has been pre-trained on extensive datasets, allowing it to understand and generate text with high accuracy and fluency.

4. Customization and Fine-Tuning Potential: The BART-LFQA model is highly adaptable, allowing for further fine-tuning.

5. Handling Long-Form Answers: The model is optimized for long-form answers.

2.2.1.2 clear_pinecone_index()

Purpose: Clears the Pinecone index before processing a new video.

Explanation: This ensures a clean slate for each new video, preventing contamination from previous data. The function uses the Pinecone client to retrieve all namespaces in the index and then deletes all vectors within each namespace. This approach allows for efficient clearing of the entire index while maintaining the index structure.

Code Snippet:

```
def clear_pinecone_index():
    global pinecone_index
    try:
        namespaces =
pinecone_index.describe_index_stats()['namespaces'].keys()
        for namespace in namespaces:
            pinecone_index.delete(delete_all=True, namespace=namespace)
        print("Pinecone index cleared successfully")
    except Exception as e:
        print(f"Error clearing Pinecone index: {str(e)}")
```

2.2.2 Video Processing

2.2.2.1 preprocess_video()

Purpose: Extracts video information and transcript, then splits the content into manageable chunks.

Explanation: Preprocessing allows for efficient storage and retrieval of video content. The function uses yt-dlp to extract video information and YouTubeTranscriptApi to get the transcript. It then employs a RecursiveCharacterTextSplitter to break the content into chunks, ensuring that each chunk is semantically meaningful and of an appropriate size for processing.

Code Snippet:

```
def preprocess_video(video_id):
    ydl_opts = {'skip_download': True}
    video_data = []
    with yt_dlp.YoutubeDL(ydl_opts) as ydl:
```

```
        info =
ydl.extract_info(f"https://www.youtube.com/watch?v={video_id}",
download=False)

    transcript = YouTubeTranscriptApi.get_transcript(video_id)

    text_splitter = RecursiveCharacterTextSplitter(chunk_size=2000,
chunk_overlap=400)

     # ... (see app for more details)

  return video_data, info
```

2.2.2.2 extract_key_topics()

Purpose: Identifies the main topics from the video title and description.

Explanation: Key topics help in generating more focused and relevant summaries. The function combines the title and description, tokenizes the text, and uses a Counter to identify the most common words. It then filters out stop words using NLTK's stopwords list to focus on meaningful content words.

Code Snippet:

```
def extract_key_topics(title, description):

    text = f"{title} {description}".lower()

    words = re.findall(r'\w+', text)

    word_counts = Counter(words)

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

    key_topics = [word for word, count in word_counts.most_common(5) if
word not in stop_words]

    return key_topics
```

2.2.3 Task Determination

2.2.3.1 determine_task()

Purpose: Decides whether the user input requires summarization or question-answering.

Explanation: This function allows the app to handle different types of user queries appropriately. The function uses a list of predefined phrases associated with summarization tasks. It checks if any of these phrases are present in the user's input (case-insensitive) to determine the task type. This approach allows for easy expansion of recognized phrases in the future.

Code Snippet:

```
def determine_task(user_input):
    summary_phrases = ["summarize", "summary", "overview", "main points",
"key points"]
    if any(phrase in user_input.lower() for phrase in summary_phrases):
        return "summarization"
    else:
        return "question_answering"
```

2.2.4 Summarization

2.2.4.1 get_context()

Purpose: Retrieves relevant context from the Pinecone index for summarization.

Explanation: Efficient context retrieval is crucial for generating accurate summaries. For summarization, the function retrieves all vectors from the Pinecone index for the given video. It uses a zero vector as the query to avoid biasing the retrieval towards any specific content. This ensures a comprehensive representation of the video content for summarization.

Code Snippet:

```
def get_context(task, video_id, max_contexts=20, is_summarization=False):
    namespace = f"video_{video_id}"
    if is_summarization:
        query_response = pinecone_index.query(
            vector=[0]*768,
            top_k=max_contexts,
            include_metadata=True,
            namespace=namespace
        )
    else:
        query_embedding = retriever.encode([task]).tolist()
        query_response = pinecone_index.query(
            vector=query_embedding[0],
            top_k=max_contexts * 2,
            include_metadata=True,
```

```
            namespace=namespace
        )
        # ... (see app for more details)
    return contexts
```

2.2.4.2 abstractive_summarize()

Purpose: Generates an abstractive summary of the video content.

Explanation: Abstractive summarization provides concise, human-like summaries of the video content. The function uses a pre-trained BART model for summarization. It prepends key topics to the input text to guide the summary towards important content. The function employs various generation parameters like beam search, length penalties, and sampling to produce diverse and coherent summaries.

Code Snippet:

```
def abstractive_summarize(text, key_topics, max_length=150, min_length=50):
    topic_text = " ".join(key_topics)
    full_text = f"{topic_text}. {text}"
    inputs = summarization_tokenizer([full_text], max_length=1024, return_tensors="pt", truncation=True)
    summary_ids = summarization_model.generate(
        inputs["input_ids"],
        num_beams=4,
        max_length=max_length,
        min_length=min_length,
        length_penalty=1.5,
        early_stopping=True,
        no_repeat_ngram_size=3,
        do_sample=True,
        top_k=50,
        top_p=0.95
    )
    summary = summarization_tokenizer.decode(summary_ids[0], skip_special_tokens=True)
    return summary
```

2.2.4.3 post_process_summary()

Purpose: Refines the generated summary for better readability and coherence.

Explanation: Post-processing ensures that the final summary is polished and user-friendly. This function applies a series of refinements to the raw summary. It removes redundant information, reorders sentences for better flow, adds transition phrases, and fixes capitalization. These steps are implemented as separate helper functions, allowing for modular improvements and easy testing of each step.

Code Snippet:

```
def post_process_summary(summary, key_topics):
    summary = re.sub(r'Video title:.*?\.\s*', '', summary,
flags=re.IGNORECASE)
    sentences = sent_tokenize(summary)
    sentences = [s for s in sentences if len(s.split()) > 5]
    sentences = reorder_sentences(sentences)
    sentences = add_transitions(sentences)
    processed_summary = ' '.join(sentences)
    processed_summary = fix_capitalization(processed_summary)
    processed_summary = re.sub(r'\s+\.', '.', processed_summary)
    processed_summary = re.sub(r'\s+,', ',', processed_summary)
    return processed_summary
```

2.2.4.4 reorder_sentences()

Purpose: Reorders sentences in the summary for better flow.

Explanation: Improves the logical progression of ideas in the summary. The current implementation simply sorts sentences by length, assuming shorter sentences are more general and should appear first. This is a simplistic approach that could be improved with more sophisticated natural language processing techniques in future iterations.

Code Snippet:

```
def reorder_sentences(sentences):
    return sorted(sentences, key=len)
```

2.2.4.5 add_transitions()

Purpose: Adds transition phrases between sentences for smoother reading.

Explanation: Enhances the coherence and readability of the summary. The function randomly inserts transition phrases at the beginning of sentences. It uses a probability check to determine whether to add a transition to each sentence, ensuring that not every sentence is modified. This approach maintains a balance between coherence and naturalness in the summary.

Code Snippet:

```
def add_transitions(sentences):
    transitions = ["Additionally, ", "Furthermore, ", "Moreover, ", "In addition, ", "Also, "]
    for i in range(1, len(sentences)):
        if random.random() < 0.3:
            sentences[i] = random.choice(transitions) + sentences[i][0].lower() + sentences[i][1:]
    return sentences
```

2.2.4.6 fix_capitalization()

Purpose: Ensures proper capitalization in the summary.

Explanation: Improves the grammatical correctness and readability of the summary. This function uses NLTK's part-of-speech tagging to identify proper nouns and capitalize them. It also ensures that the first word of each sentence is capitalized. This approach combines rule-based and NLP-based methods for accurate capitalization.

Code Snippet:

```
def fix_capitalization(text):
    sentences = sent_tokenize(text)
    capitalized_sentences = [s.capitalize() for s in sentences]
    text = ' '.join(capitalized_sentences)
    words = word_tokenize(text)
    tagged = pos_tag(words)
```

```
    capitalized_words = [word.capitalize() if tag.startswith('NNP') else
word for word, tag in tagged]
    return ' '.join(capitalized_words)
```

2.2.5 Question-Answering

2.2.5.1 get_context()

Purpose: Retrieves relevant context from the Pinecone index for answering questions.

Explanation: Efficient context retrieval is crucial for generating accurate answers. For question-answering, the function encodes the query using a SentenceTransformer model and uses this encoding to retrieve similar vectors from the Pinecone index. It retrieves more contexts than needed, allowing for further refinement in subsequent steps. This two-stage approach improves the relevance of the retrieved context. This function is shared with summarization and was described earlier.

2.2.5.2 generate_answer()

Purpose: Generates an answer to the user's question based on the retrieved context.

Explanation: Utilizes the loaded model to produce coherent and relevant answers. The function constructs a prompt that includes the context and the question. It then uses the pre-trained model to generate an answer, applying various generation parameters to control the output. The function also includes a fallback response for cases where the question cannot be answered based on the given context.

Code Snippet:

```
def generate_answer(question, context):
    max_input_length = 1024
    input_text = f"""Answer the question based on the following context. If
the question cannot be answered based on the context, respond only with "I
don't have information about that in the context of this video."


Context: {context[:max_input_length]}


Question: {question}


Answer:"""
```

```
    inputs = tokenizer([input_text], max_length=1024, return_tensors="pt",
truncation=True)


    with torch.no_grad():
        outputs = model.generate(
            inputs.input_ids,
            max_length=300,
            min_length=50,
            do_sample=True,
            num_beams=5,
            top_p=0.95,
            temperature=0.7,
            early_stopping=True,
            no_repeat_ngram_size=3,
             num_return_sequences=1,
             eos_token_id=tokenizer.eos_token_id,
            pad_token_id=tokenizer.pad_token_id,
        )


    answer = tokenizer.decode(outputs[0], skip_special_tokens=True)
    return split_into_sentences(answer)
```

2.2.5.3 check_relevance()

Purpose: Verifies if the generated answer is relevant to the question and context.

Explanation: Ensures the quality and relevance of the answers provided to the user. This function uses the local language model to assess the relevance of the generated answer. It constructs a prompt that includes the question, answer, and a snippet of the context, then asks the model to determine relevance. This approach leverages the model's understanding of language and context to perform a quality check on the generated answer.

Code Snippet:

```
def check_relevance(question, answer, context):
    relevance_prompt = f"""
    Question: {question}
```

```
    Answer: {answer}

    Context: {context[:500]}


    Is the answer relevant to the question and based on the given context?
Respond with Yes or No.

    """

    relevance_check = local_llm(relevance_prompt)

    return relevance_check.strip().lower() == "yes"
```

2.2.5.4 hybrid_context_retrieval()

Purpose: Combines TF-IDF and semantic similarity for more accurate context retrieval.

Explanation: Improves the relevance of retrieved context by considering both lexical and semantic similarities. The function uses both TF-IDF and SentenceTransformer embeddings to compute similarities between the query and contexts. It then combines these scores using a weighted average, allowing for fine-tuning of the retrieval process. This hybrid approach helps balance between exact keyword matching and semantic understanding.

Code Snippet:

```
def hybrid_context_retrieval(query, contexts, top_k=7, tfidf_weight=0.4):

    vectorizer = TfidfVectorizer(stop_words='english')

    tfidf_matrix = vectorizer.fit_transform([query] + contexts)

    tfidf_scores = cosine_similarity(tfidf_matrix[0:1],
tfidf_matrix[1:]).flatten()


    query_embedding = retriever.encode([query])

    context_embeddings = retriever.encode(contexts)

    semantic_scores = cosine_similarity(query_embedding,
context_embeddings)[0]


    combined_scores = tfidf_weight * tfidf_scores + (1 - tfidf_weight) *
semantic_scores


    top_indices = combined_scores.argsort()[-top_k:][::-1]

    return [contexts[i] for i in top_indices]
```

2.2.5.5 extract_relevant_sentences()

Purpose: Extracts the most relevant sentences from the context for answering the question.

Explanation: Focuses the answer generation on the most pertinent information. The function tokenizes both the question and context into words, removes stop words, and then scores each sentence in the context based on its word overlap with the question. It then selects the top-scoring sentences. This approach balances simplicity with effectiveness, providing a good starting point for more advanced relevance extraction in future iterations.

Code Snippet:

```
def extract_relevant_sentences(question, context, num_sentences=2):
    sentences = sent_tokenize(context)
    stop_words = set(stopwords.words('english'))
    question_words = [w.lower() for w in word_tokenize(question) if
w.lower() not in stop_words]

    scores = []
    for sentence in sentences:
        words = [w.lower() for w in word_tokenize(sentence) if w.lower()
not in stop_words]
        score = sum(1 for w in question_words if w in words)
        scores.append(score)

    top_indices = sorted(range(len(scores)), key=lambda i: scores[i],
reverse=True)[:num_sentences]
    top_sentences = [sentences[i] for i in sorted(top_indices)]

    return " ".join(top_sentences)
```

2.2.6 Common Functions

2.2.6.1 remove_repetitions()

Purpose: Removes duplicate sentences from the generated text.

Explanation: Improves the conciseness and readability of both summaries and answers. The function splits the text into sentences, then iterates through them, adding each unique sentence to a new list. This approach preserves the original order

of sentences while removing exact duplicates. The implementation uses string stripping to ignore minor differences in whitespace.

Code Snippet:

```
def remove_repetitions(text):
    sentences = text.split('.')
    unique_sentences = []
    for sentence in sentences:
        if sentence.strip() and sentence.strip() not in unique_sentences:
            unique_sentences.append(sentence.strip())
    return '. '.join(unique_sentences) + '.'
```

2.2.6.2 split_into_sentences()

Purpose: Splits the generated text into properly formatted sentences.

Explanation: Ensures that the output text is well-structured and readable, with proper sentence boundaries. This function uses a regular expression to split text into sentences more accurately than simple period-based splitting. It then applies the `clean_and_format_sentence` function to each sentence, ensuring consistent formatting. This two-step approach allows for more nuanced sentence boundary detection and individual sentence cleanup.

Code Snippet:

```
def split_into_sentences(text):
    # Use a regex to split text into sentences more accurately
    sentences = re.split(r'(?<=[.!?])\s+(?=[A-Z])', text)
    cleaned_sentences = [clean_and_format_sentence(s) for s in sentences if
len(s.strip()) > 10]

    # Ensure sentences are joined properly without unnecessary commas
    return ' '.join(cleaned_sentences)
```

2.2.6.3 clean_and_format_sentence()

Purpose: Cleans and formats individual sentences for consistency and readability.

Architecture Motivation: Provides a standardized way to format sentences, ensuring consistency across different parts of the application.

Implementation Details: The function removes extra spaces, trailing commas, and [Music] tags (which may appear in video transcripts). It also ensures that sentences end with proper punctuation and start with a capital letter. This detailed cleaning process helps maintain a professional and consistent output quality.

Code Snippet:

```
def clean_and_format_sentence(sentence):
    # Remove extra spaces and trailing commas, and [Music] tags
    sentence = re.sub(r'\s+', ' ', sentence).strip().rstrip(',')
    sentence = re.sub(r'\[Music\]', '', sentence)

    # Ensure the sentence ends with proper punctuation
    if not sentence.endswith(('.', '?', '!')):
        sentence += '.'

    # Capitalize the first letter
    if len(sentence) > 0:
        sentence = sentence[0].upper() + sentence[1:]

    return sentence
```

3. Further steps

Further steps were taken to improve the app's performance, but were not completed due to the time constraints. These steps are not implemented in the deployed app.

3.1 Memory

ConversationBufferWindowMemory was implemented on Jypiter Notebook in the before working on the app (see Processing any video from youtube/Any_video Working example with Bart.ipynb).

Code Snippet:

```
memory = ConversationBufferWindowMemory(k=3)
local_llm = HuggingFacePipeline(pipeline=pipe)
conversation = ConversationChain(
    llm=local_llm,
    memory=memory,
    verbose=True
)
```

3.2 Fine-tuning and evaluation

I attempted to fine tune the model on two datasets, the Stanford Question Answering Dataset (SQuAD) and ELI5-Category dataset, that both seemed promising fit. Due to time constraints, computational limits, and the size of the model being fine-tuned, I had to drastically limit training parameters, which resulted in a poor performance (see Table 1).

Table 1. Model evaluation results.

	vblagoje/bart_lfqa	Model fine-tuned on SQUAD
Faithfulness	0.653	0.484
Context Recall	0.578	0.578
Relevance	0.110	0.110
ROUGE-1	0.110	0.110
ROUGE-2	0.056	0.056
ROUGE-L	0.110	0.110

Source: own elaboration.

For this reason, the fine-tuned models were used in the deployed app. Nevertheless, I acquired skills of applying PEFT optimization and model quantization, as well loading and utilizing the quantized model.

4. Conclusion and Future Work

4.1 Conclusion

The project successfully developed and deployed a chatbot that can summarize and translate YouTube videos into text allowing for natural language querying.

4.2 Key Takeaways

The project taught me a more cautious approach to defining its scope.

4.3 Future Work

Areas for potential future improvement include:

- Allowing processing video without youtube transcript
- Fine-tuning both question answering and summarization models
- More advanced sentence reordering in summarization
- Integration of more sophisticated NLP techniques for context extraction and answer generation