PDF Answering AI

Question and Answering System

for extensive pdf documents

Submitted by:

Akula Koushik

21114010

CSE 4th Year

1. Introduction

Problem Statement

The rapid advancement of artificial intelligence has opened up new possibilities in natural language processing (NLP), particularly in the development of question-answering (QA) systems. The goal of this project is to create a QA system that can accurately interpret and respond to questions based on the content of a given document. This system is particularly useful for applications such as automated customer service, academic research assistance, and content summarization.

Objectives

- **Text Extraction:** Develop a reliable method to extract text from PDF documents.
- **Preprocessing:** Clean and preprocess the extracted text to ensure it is suitable for NLP tasks.
- **Embedding:** Implement a method to convert text into embeddings for similarity search.
- **QA System Implementation:** Use a pre-trained language model to build a QA system.
- **Evaluation:** Measure the performance of the QA system using various metrics.

2. Approach

Methodology

The project follows a structured methodology to address the objectives outlined above. The approach can be divided into the following phases:

Text Extraction

The first step involves extracting text from PDF documents. This is achieved using the pdfplumber library, which allows for precise extraction of text, preserving the original formatting and structure.

```
import pdfplumber
import re
import torch
```

```
def extract_text_from_pdf(pdf_path):
    with pdfplumber.open(pdf_path) as pdf:
        text = ""
        for page in pdf.pages:
            text += page.extract_text()
    return text

pdf_text = extract_text_from_pdf("blade_runner_2049.pdf")
```

Preprocessing

The extracted text often contains special characters, newlines, and other non-alphanumeric characters that need to be removed. Additionally, converting the text to lowercase standardizes the input for the language model.

```
def preprocess_text(text):
    # Remove special characters, newlines, and other non-alphanumeric characters
    text = re.sub(r'[^a-zA-Z0-9\s]', '', text)

    # Lowercasing
    text = text.lower()

    return text
```

```
preprocessed_text = preprocess_text(pdf_text)
```

Embedding and Vector Store

To enable efficient retrieval of relevant text passages, we use embeddings. Hugging Face embeddings and the FAISS library are used for this purpose. FAISS (Facebook AI Similarity Search) allows for efficient similarity search, which is essential for large-scale text retrieval.

```
from langchain.vectorstores import FAISS
from langchain.embeddings import HuggingFaceEmbeddings
```

```
embeddings = HuggingFaceEmbeddings(model_name='sentence-transformers/all-MiniLM-L6-v2',model_kwargs={'device': 'cuda'})
```

```
text_splitter=CharacterTextSplitter(separator='\n',
                                    chunk_size=1000,
                                    chunk_overlap=200)
text_chunks=text_splitter.split_documents(documents)
vectorstore=FAISS.from_documents(text_chunks, embeddings)
```

Question-Answering Chain

The core of the QA system is the QA chain, which utilizes a pre-trained language model from Hugging Face. This involves loading the model, tokenizer, and setting up the QA chain to handle user queries.

```
from langchain.chains.question_answering import load_qa_chain
from transformers import AutoTokenizer, AutoModelForCausalLM

tokenizer = AutoTokenizer.from_pretrained(model_name,token=token,trust_remote_code=True, padding_side="left")
```

```
model = AutoModelForCausalLM.from_pretrained(model_name,token=token,quantization_config=bnb_config,device_map='auto',torch_dtype=torch.float16,trust_remc
model.config.use_cache = False
```

```
Loading checkpoint shards:   0%|          | 0/2 [00:00<?, ?it/s]
```

```
pipe = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    max_length=3000,
    num_return_sequences=1,
    repetition_penalty=1.2,
)
```

```
llm=HuggingFacePipeline(pipeline=pipe)
qa =  RetrievalQA.from_chain_type(llm=llm, chain_type = "stuff",return_source_documents=False, retriever=vectorstore.as_retriever(k=2))
```

3. Failed Approaches

Initial Model Selection

The project initially used a smaller language model for the QA system. However, this model failed to provide accurate answers due to its limited understanding and capacity. This failure highlighted the importance of selecting a robust model that can handle complex queries and large volumes of text.

Text Chunking Issues

Early attempts at chunking the text for embedding purposes led to a loss of context, which negatively impacted the QA system's performance. This issue was resolved by adopting a more sophisticated text splitting strategy that preserved the context.

4.Results

1. Accuracy Metrics

The performance of the QA system was evaluated using BERTScore and Cosine Similarity. These metrics were chosen to measure the semantic similarity between the true answers and the predicted answers, rather than relying on exact matches.

BERTScore

BERTScore leverages pre-trained language models like BERT to compute similarity scores between sentences. It captures the semantic similarity more effectively than traditional n-gram-based methods. The evaluation results are presented below:

- **Precision**: 0.857
- **Recall**: 0.868
- **F1-Score**: 0.862

```python
import bert_score

def bert_score_evaluation(true_answers, predicted_answers):
    # Calculate BERTScore
    P, R, F1 = bert_score.score(predicted_answers, true_answers, lang="en", verbose=True)

    # Return average precision, recall, and F1-score
    precision = P.mean().item()
    recall = R.mean().item()
    f1 = F1.mean().item()

    return precision, recall, f1
```

These metrics provide a robust measure of the QA system's effectiveness in capturing the meaning of the answers.

Cosine Similarity

Cosine Similarity was used to evaluate the semantic closeness of the embeddings of true and predicted answers. This method helps in quantifying the similarity between the two sets of answers.

The cosine similarity scores are as follows:

- **Similarity Scores**: [0.72, 0.61, 0.48, 0.57, 0.61]

```python
# Calculate the cosine similarity between true and predicted answers
def cosine_similarity_evaluation(true_answers, predicted_answers):
    # Convert answers to embeddings
    true_embeddings = model.encode(true_answers, convert_to_tensor=True)
    predicted_embeddings = model.encode(predicted_answers, convert_to_tensor=True)

    # Calculate cosine similarities
    similarities = util.pytorch_cos_sim(true_embeddings, predicted_embeddings)

    # Determine matches based on highest similarity
    matched_scores = similarities.diag()

    # Return similarity scores
    return matched_scores.tolist()

similarity_scores = cosine_similarity_evaluation(true_answers, predicted_answers)
print(similarity_scores)
```

```
[0.7243386507034302, 0.6191291213035583, 0.4859050512313843, 0.5794343948364258, 0.6040864586830139]
```

These scores indicate the degree of similarity between the true and predicted answers, with higher scores representing more similar answers.

5. Discussion

Significance of Results

The results demonstrate the importance of model selection and preprocessing in developing an effective QA system. The improved model, along with advanced preprocessing techniques, significantly enhanced the system's accuracy and reliability.

Insights Gained

- **Model Selection:** Choosing the right model is crucial for handling complex queries.
- **Preprocessing:** Effective preprocessing, including cleaning and chunking, improves the system's performance.
- **Context Maintenance:** Preserving context during text chunking is essential for accurate QA.

6. Conclusion

Summary of Findings

The project successfully developed an automated QA system capable of processing and answering questions based on document content. Key findings include the importance of robust model selection, advanced preprocessing techniques, and efficient embedding strategies.

Future Improvements

- **Advanced Models:** Explore more sophisticated models with higher capacity and better contextual understanding.
- **Feedback Loop:** Implement a feedback mechanism to continuously improve the system based on user interactions.
- **Complex Documents:** Enhance preprocessing steps to handle more complex and varied document structures.

7. References

- LangChain Documentation: LangChain
- Hugging Face Transformers: Hugging Face
- FAISS Documentation: FAISS