

Basic of RAG

```
###Document Structure

from langchain_core.documents import Document

✓ 0.0s

doc=Document(
  page_content="this is the main text content I am using to create RAG",
  metadata={
    "source":"exmaple.txt" (page_content: str, **kwargs: Any) -> Document
    "pages":1, Pass page_content in as positional or named arg.
    "author":"Krish Naik",
    "date_created":"2024-01-01"
  )
)
```

here the document looks like this, that contains the page_contents, metadata which contains the source pages etc . this meta data helps for thw filtering of the data from other data , so retrival may be fast. means if u ask what is the source content written by " Krish Naik " , then it filters all the krishnaik page docs

```
import os
os.makedirs("../data/text_files",exist_ok=True)

✓ 0.0s

sample_texts={
  "../data/text_files/python_intro.txt":"Python Programming Introduction

Python is a high-level, interpreted programming language known for its simplicity and readability
Created by Guido van Rossum and first released in 1991, Python has become one of the most popular
programming languages in the world.

Key Features:
- Easy to learn and use
- Extensive standard library
- Cross-platform compatibility
"
```

creating the manual text for the files , files are creating using the os.makedirs

```
### Textloader
from langchain.document_loaders import TextLoader

from langchain_community.document_loaders import TextLoader

loader=TextLoader("../data/text_files/python_intro.txt",encoding="utf-8")
document=loader.load()
print(document)

✓ 0.0s

Python
intro.txt'}, page_content='Python Programming Introduction\n\nPython is a high-level, interpreted program
```

so using the langchain TextLoader , we are reading the manual text we created in .txt file

```
## load all the text files from the directory
dir_loader=DirectoryLoader(
    "../data/text_files",
    glob="**/*.txt", ## Pattern to match files
    loader_cls= TextLoader, ##loader class to use
    loader_kwargs={'encoding': 'utf-8'},
    show_progress=False
)

I
documents=dir_loader.load()
documents

✓ 0.0s Python

[Document(metadata={'source': '..\\data\\text_files\\machine_learning.txt'}, page_content='Machine Learn
Document(metadata={'source': '..\\data\\text_files\\python_intro.txt'}, page_content='Python Programmin
```

same like above here also we are reading the text from .txt files , but all the *.txt

```
from langchain_community.document_loaders import PyPDFLoader, PyMuPDFLoader

## load all the text files from the directory
dir_loader=DirectoryLoader(
    "../data/pdf",
    glob="**/*.pdf", ## Pattern to match files
    loader_cls= PyMuPDFLoader, ##loader class to use
    show_progress=False
)

pdf_documents=dir_loader.load()
pdf_documents
```

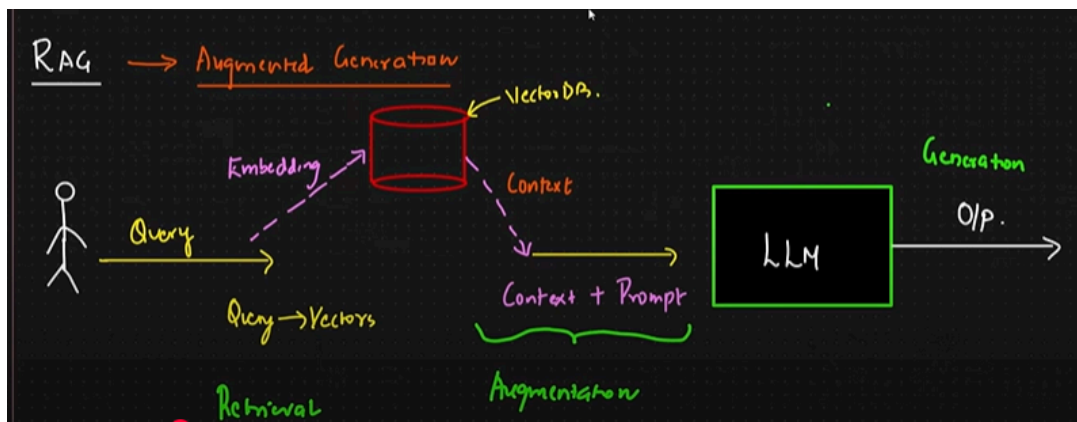
here reading the data from the PDF

```
Document(metadata={'producer': 'pdfTeX-1.40.25', 'creator': 'LaTeX with hyperref', 'creationdate': '2024
Document(metadata={'producer': 'pdfTeX-1.40.25', 'creator': 'LaTeX with hyperref', 'creationdate': '2024
Document(metadata={'producer': 'pdfTeX-1.40.25', 'creator': 'LaTeX with hyperref', 'creationdate': '2024
Document(metadata={'producer': 'pdfTeX-1.40.25', 'creator': 'LaTeX with hyperref', 'creationdate': '2024
Document(metadata={'producer': 'pdfTeX-1.40.25', 'creator': 'LaTeX with hyperref', 'creationdate': '2024
Document(metadata={'producer': 'pdfTeX-1.40.25', 'creator': 'LaTeX with hyperref', 'creationdate': '2024
Document(metadata={'producer': 'pdfTeX-1.40.25', 'creator': 'LaTeX with hyperref', 'creationdate': '2024
Document(metadata={'producer': 'pdfTeX-1.40.25', 'creator': 'LaTeX with hyperref', 'creationdate': '2024
```

the data looks likes this , here it creates like "data content" and "metadata"

1. then we need to create the embeddings using any openAi/HuggingFace model
2. build vectorStore either using Pinecone,ChromaDb,Fiass
3. then RAG retriever

- the query we give will be converted to embeddings
- then similarity search score
- if similarity search score is greater than threshold then retrieve that as ans



→ RAGS implementation types

- we can do RAG using the normal python functions , but large code
- using simple RAG pipeline

```

### Simple RAG pipeline with Groq LLM
from langchain_groq import ChatGroq
import os
from dotenv import load_dotenv
load_dotenv()

### Initialize the Groq LLM (set your GROQ_API_KEY in environment)
groq_api_key = "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"

llm=ChatGroq(groq_api_key=groq_api_key,model_name="gemma2-9b-it",temperature=0.1,max_tokens=1024)

## 2. Simple RAG function: retrieve context + generate response
def rag_simple(query,retriever,llm,top_k=3):
    ## retriever the context
    results=retriever.retrieve(query,top_k=top_k)
    context="\n\n".join([doc['content'] for doc in results]) if results else ""
    if not context:
        return "No relevant context found to answer the question."

    ## generate the answer using GROQ LLM
    prompt=f""Use the following context to answer the question concisely.

```

- loading the groq llm model , used its API key
- we are retrieving the context first (simple Retrieve RAG)

```

## generate the answer using GROQ LLM
prompt=f"""Use the following context to answer the question concisely.
Context:
{context}

Question: {query}

Answer: """
response=llm.invoke([prompt.format(context=context,query=query)])
return response.content

```

- then we gave the prompt inside that function only
- so when the question is asked it sends to the function, there some answers are retrieved from the vector database
- and then sends that to the llm model to get the correct o/p based on the retrievers o/p

```

answer=rag_simple("What is attention mechanism?",rag_retriever,llm)
print(answer)
✓ 0.4s
Retrieving documents for query: 'What is attention mechanism?'
Top K: 3, Score threshold: 0.0
Generating embeddings for 1 texts...
Batches: 100%|██████████| 1/1 [00:00<00:00, 76.39it/s]
Generated embeddings with shape: (1, 384)
Retrieved 3 documents (after filtering)

An attention mechanism is a function that maps a query and a set of key-value pairs to an output vector, using a weighted

```

we are calling the function with the query as the parameter send to the function where we have retriever,LLM model

3. Enhanced RAG Pipeline Features

```

Enhanced RAG Pipeline Features

# --- Enhanced RAG Pipeline Features ---
def rag_advanced(query, retriever, llm, top_k=5, min_score=0.2, return_context=False):
    """
    RAG pipeline with extra features:
    - Returns answer, sources, confidence score, and optionally full context.
    """
    results = retriever.retrieve(query, top_k=top_k, score_threshold=min_score)
    if not results:
        return {'answer': 'No relevant context found.', 'sources': [], 'confidence': 0.0, 'context': ''}

    # Prepare context and sources
    context = "\n\n".join([doc['content'] for doc in results])
    sources = [
        {
            'source': doc['metadata'].get('source_file', doc['metadata'].get('source', 'unknown')),
            'page': doc['metadata'].get('page', 'unknown'),
            'score': doc['similarity_score'],
            'preview': doc['content'][:120] + '...'
        } for doc in results
    ]

```

```

        'preview': doc['content'][:300] + '...'
    } for doc in results]
    confidence = max([doc['similarity_score'] for doc in results])

    # Generate answer
    prompt = f"""Use the following context to answer the question concisely.\nContext:\n{context}\n\nQuestion: {query}\n\nAns
response = llm.invoke([prompt.format(context=context, query=query)])

    output = {
        'answer': response.content,
        'sources': sources,
        'confidence': confidence
    }
    if return_context:
        output['context'] = context
    return output

```

Defining the RAG and LLM model function

```

        'sources': sources,
        'confidence': confidence
    }
    if return_context:
        output['context'] = context
    return output

# Example usage:
result = rag_advanced("What is attention mechanism?", rag_retriever, llm, top_k=3, min_score=0.1, return_context=True)
print("Answer:", result['answer'])
print("Sources:", result['sources'])
print("Confidence:", result['confidence'])
print("Context Preview:", result['context'][:300])

```

this result contains the answer(what we get after passing the context to LLM model) , sources, confidence score , context that passes through LLM

```

Retrieving documents for query: 'What is attention mechanism?'
Top K: 3, Score threshold: 0.1
Generating embeddings for 1 texts...
Batches: 100%|██████████| 1/1 [00:00<00:00, 109.81it/s]
Generated embeddings with shape: (1, 384)
Retrieved 3 documents (after filtering)

Answer: An attention mechanism is a function that maps a query and a set of key-value pairs to an output vector, where the output
Sources: [{'source': 'attention.pdf', 'page': 2, 'score': 0.2714172601699829, 'preview': '3.2 Attention\nAn attention function ca
Confidence: 0.2714172601699829
Context Preview: 3.2 Attention
An attention function can be described as mapping a query and a set of key-value pairs to an output,
where the query, keys, values, and output are all vectors. The output is computed as a weighted sum
3
3.2 Attention
An attention function can be described as mapping a query and a set

```

actually in second type we saw the simple code of the RAG where we get the contents only, but here we tried to get the Answer as well as the "Sources, Confidence Score" also

4. TypeSense-Lightning Fast

- <https://www.youtube.com/watch?v=MMS04bku3FE&list=PLZoTAELRMXVM8Pf4U67L4UuDRgV4TNX9D&index=4>
- this is the link of the video

- here instead of using the PineCone/ChromaDb/Fiass we used Typesense which has ability for fast retrieval of the data , It is a Open source model only , here we need to create a collections of data