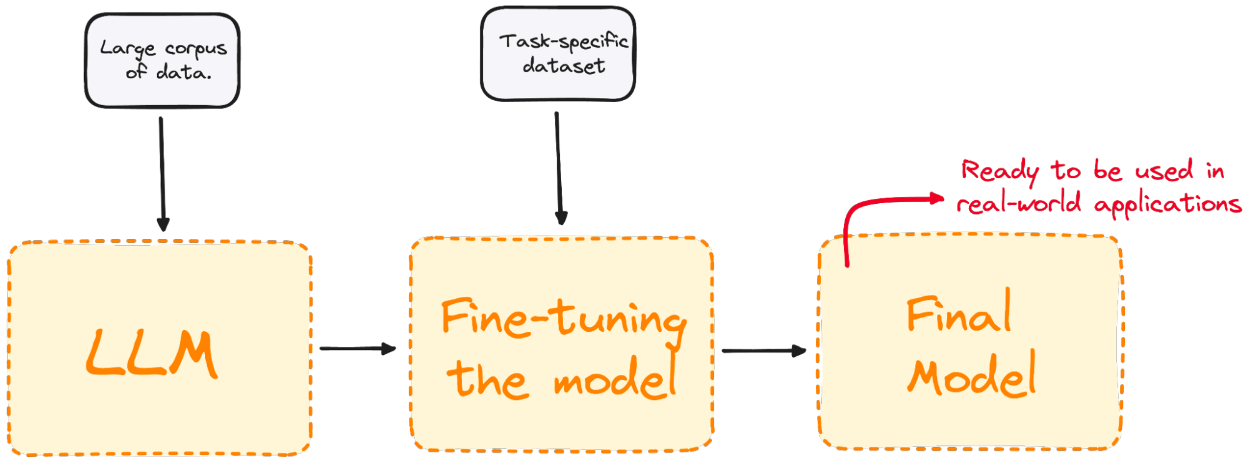
FINE TUNING OF LARGE LANGUAGE MODELS

Fine Tuning in general refers to the process of adjusting the parameters of a pretrained model here, specifically those of a LLM for a particular use case or domain .



Here we trained the pre trained LLM for our specific usecase

ADVANTAGES OF FINETUNING

- Fine Tuning allows the model to acquire a deeper understanding of the nuances of the domain by allowing it to learn from domain-specific data to make it more accurate and effective for targeted applications.

- Organizations can ensure their model adheres to data compliance standards by fine-tuning the LLM on proprietary or regulated data.

- Fine Tuning helps in dealing with data scarcity since fine tuning llms on limited labeled data is helpful too.

APPROACHES TO LLM FINE TUNING

There are two major approaches to Fine Tune a Large Language Model.

1) Feature Extraction
2) Full Fine Tuning

In the first approach only the final layers of the model are trained on the task-specific data while the rest of the model remains frozen.It is also the primary approach to fine-tuning LLMs. Here the pre-trained LLM is treated as a fixed feature extractor

On the other hand Full Fine Tuning changes all the weights of LLM in order to customize the model on the dataset.
Full Fine Tuning is particularly beneficial when the task-specific dataset is large and significantly different from the pre-training data.

METHODS OF LLM FINE TUNING

There are again two major methods of fine tuning a LLM . These are -
1)Supervised fine-tuning
2)Reinforcement learning from human feedback (RLHF)
In Supervised Fine Tuning, the model is trained on a task-specific labeled dataset, where each input data point is associated with a correct answer or label. The model learns to adjust its parameters to predict these labels as accurately as possible.

On the other hand Reinforcement learning from human feedback involves training language models through interactions with human feedback.

TECHNIQUES FOR SUPERVISED FINE TUNING

The major techniques under supervised fine tuning are as follows -

1) **Basic hyperparameter tuning** - It involves manually adjusting the model hyperparameters, such as the learning rate, batch size, and the number of epochs, until you achieve the desired performance.
2) **Transfer learning -** In this approach, a model pre-trained on a large, general dataset is used as a starting point and then fine-tuned on the task-specific data
3) **Multi-task learning-**Here the model is fine-tuned on multiple related tasks simultaneously.
4) **Few-shot learning -** In this technique, the model is given a few examples during inference time to learn a new task.It enables a model to adapt to a new task with little task-specific data
5) **Task-specific fine-tuning -** It allows the model to adapt its parameters to the nuances and requirements of the targeted task. It optimizes the model's performance for a single, well-defined task.

TECHNIQUES FOR RLHF

The major techniques under RLHF Fine Tuning are as follows -

1) **Reward modeling** - here the model generates several possible outputs or actions, and human evaluators rank or rate these outputs based on their quality.

This method enables the model to learn and adapt based on human-provided incentives.

2) **Proximal policy optimization -** It updates the language model's policy to maximize the expected reward.It introduces a constraint on the policy update that prevents harmful large updates while still allowing beneficial small updates.
3) **Comparative ranking-** In Comparative Ranking model learns from relative rankings of multiple outputs provided by human evaluators, focusing more on the comparison between different outputs**.**
4) **Preference learning (reinforcement learning with preference feedback) -** In this technique, the model generates multiple outputs, and human evaluators indicate their preference between pairs of outputs.The model then learns to adjust its behavior to produce outputs that align with the human evaluators' preferences
5) **Parameter efficient fine-tuning -** improves the performance of pre-trained LLMs on specific downstream tasks while minimizing the number of trainable parameters. It offers a more efficient approach by updating only a minor fraction of the model parameters during fine-tuning.

DATASET PREPARATION-

We need to perform a few steps on our dataset to align it with LLM . These are -
1)Data Cleaning(for custom)
2)Data Augmentation(for custom then uploading on hugging face)
3) Dataset Loading
4)Tokenization

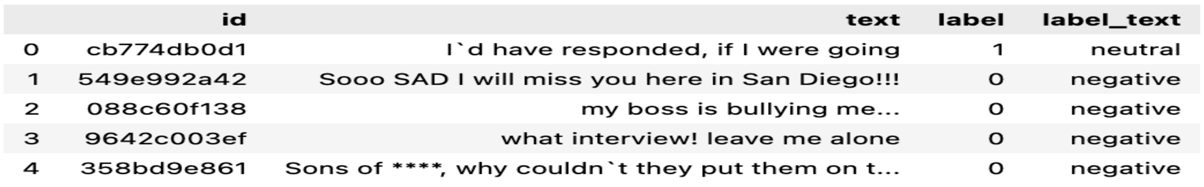
DATASET LOADING

We can load existing datasets from Hugging Face Library and upload our own dataset using datasets library-
1)Loading existing datasets from Datasets Library-
We can load and convert these data into dataframe format.For example below we load a dataset on sentiments and extract its train column to train the model -
CODE

```
from datasets import load_dataset

dataset = load_dataset("mteb/tweet_sentiment_extraction")
df = pd.DataFrame(dataset['train'])
```

OUTPUT



	id	text	label	label_text
0	cb774db0d1	I`d have responded, if I were going	1	neutral
1	549e992a42	Sooo SAD I will miss you here in San Diego!!!	0	negative
2	088c60f138	my boss is bullying me...	0	negative
3	9642c003ef	what interview! leave me alone	0	negative
4	358bd9e861	Sons of ****, why couldn`t they put them on t...	0	negative

DATASET LOADING

We need a tokenizer to prepare our training data to be parsed by our model.
To process our dataset in one step, we can use the **Datasets map** method to apply a preprocessing function over the entire dataset.

CODE

```
from transformers import GPT2Tokenizer

# Loading the dataset to train our model
dataset = load_dataset("mteb/tweet_sentiment_extraction")

tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
tokenizer.pad_token = tokenizer.eos_token
def tokenize_function(examples):
   return tokenizer(examples["text"], padding="max_length", truncation=True)

tokenized_datasets = dataset.map(tokenize_function, batched=True)
)
```

To improve the model's output we can divide the entire dataset into smaller subsets and fine tune the model on these subsets.

INITIALIZATION OF BASE MODEL AND ADDING EVALUATE METHOD

Initialisation of models could be different for different models and datasets. These models could be easily loaded using the **Transformer Library.** For the above example we can initialize the model as follows - :

CODE

```
from transformers import GPT2ForSequenceClassification
model = GPT2ForSequenceClassification.from_pretrained("gpt2", num_labels=3)
```

Transformer library does not have an inbuilt evaluator that could test the model. However we can add this via the **'evaluate'** library.

CODE

```
import evaluate
metric = evaluate.load("accuracy")
def compute_metrics(eval_pred):
   logits, labels = eval_pred
   predictions = np.argmax(logits, axis=-1)
   return metric.compute(predictions=predictions, references=labels)
```

FINE TUNING THE MODEL

The Transformers library contains the Trainer class, which supports a wide range of training options and features such as logging, gradient accumulation, and mixed precision.For the above example we first define the training arguments together with the evaluation strategy. Once everything is defined, we train the model using the train() command.

CODE

```
from transformers import TrainingArguments, Trainer

training_args = TrainingArguments(
   output_dir="test_trainer",
   #evaluation_strategy="epoch",
   per_device_train_batch_size=1,  # Reduce batch size here
   per_device_eval_batch_size=1,    # Optionally, reduce for evaluation as well
   gradient_accumulation_steps=4
   )


trainer = Trainer(
   model=model,
   args=training_args,
   train_dataset=small_train_dataset,
   eval_dataset=small_eval_dataset,
   compute_metrics=compute_metrics,

)

trainer.train()

import evaluate

trainer.evaluate()
```

PROBLEMS IN LLM FINE TUNING

1) Overfitting
2) Underfitting
3) Data Leakage

FINE TUNING LLMS IN MEMORY CONSTRAINTS

Following techniques can help in fine tuning LLMs in limited GPU capacity. - :

1) **AdaFactor** - Its a stochastic optimization method based on AdamW. AdaFactor is an optimization algorithm like Adam.

Adafactor Class is the optimiser that implements the Adafactor algorithm.

CODE

```
keras.optimizers.Adafactor(
    learning_rate=0.001,
    beta_2_decay=-0.8,
    epsilon_1=1e-30,
    epsilon_2=0.001,
    clip_threshold=1.0,
    relative_step=True,
    weight_decay=None,
    clipnorm=None,
    clipvalue=None,
    global_clipnorm=None,
    use_ema=False,
    ema_momentum=0.99,
    ema_overwrite_frequency=None,
    loss_scale_factor=None,
    gradient_accumulation_steps=None,
    name="adafactor",
    **kwargs
)
```

Arguments

- **learning_rate**: A float, a keras.optimizers.schedules.LearningRateSchedule instance, or a callable that takes no arguments and returns the actual value to use. The learning rate. Defaults to 0.001.
- **beta_2_decay**: float, defaults to -0.8. The decay rate of beta_2.
- **epsilon_1**: float, defaults to 1e-30. A small offset to keep the denominator away from 0.
- **epsilon_2**: float, defaults to 1e-3. A small offset to avoid learning rate becoming too small by time.
- **clip_threshold**: float, defaults to 1.0. Clipping threshold. This is a part of Adafactor algorithm, independent from clipnorm, clipvalue, and global_clipnorm.

- **relative_step**: bool, defaults to True. If learning_rate is a constant and relative_step=True, learning rate will be adjusted based on current iterations. This is a default learning rate decay in Adafactor.
- **name**: String. The name to use for momentum accumulator weights created by the optimizer.
- **weight_decay**: Float. If set, weight decay is applied.
- **clipnorm**: Float. If set, the gradient of each weight is individually clipped so that its norm is no higher than this value.
- **clipvalue**: Float. If set, the gradient of each weight is clipped to be no higher than this value.
- **global_clipnorm**: Float. If set, the gradient of all weights is clipped so that their global norm is no higher than this value.
- **use_ema**: Boolean, defaults to False. If True, exponential moving average (EMA) is applied. EMA consists of computing an exponential moving average of the weights of the model (as the weight values change after each training batch), and periodically overwriting the weights with their moving average.

2) **LORA Low Rank Adaptation of LLMs)-LoRA** works by freezing the majority of the original LLM's trainable parameters at each layer, it further introduces a much smaller set of learnable "low-rank" matrices at each layer. This significantly reduces the trainable parameters by freezing them, resulting in less memory consumption

PEFT (Parameter Efficient Fine Tuning is a method of LORA too)

Things to keep in mind while Using LORA are -
a)Post Fine Tuning models will be in adapter format, and loading the adapter requires using the `PeftModel` class and its specific loading approach

CODE

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
from peft import LoraConfig, get_peft_model, prepare_model_for_int8_training,
TaskType

model = AutoModelForSeq2SeqLM.from_pretrained("google/flan-t5-xxl",
load_in_8bit=True, device_map="auto")
lora_config = LoraConfig(
 r=16,
 lora_alpha=32,
 target_modules=["q", "v"],
 lora_dropout=0.05,
 bias="none",
 task_type=TaskType.SEQ_2_SEQ_LM
)
model = prepare_model_for_int8_training(model)
model = get_peft_model(model, lora_config)

model.print_trainable_parameters()
# trainable params: 18874368 || all params: 11154206720 || trainable%:
0.16921300163961817
```

3) **Gradient Checkpointing** - It can significantly reduce memory consumption (by 50–60%) by strategically managing how activations are handled during training.

   CODE

```
training_args = TrainingArguments(
    per_device_train_batch_size=1, gradient_accumulation_steps=4,
gradient_checkpointing=True, **default_args
)

trainer = Trainer(model=model, args=training_args, train_dataset=ds)
result = trainer.train()
print_summary(result)
```

4) **Gradient Accumulation** - Gradient Accumulation first divides data into mini batches and each mini batch further into subbatches. For each sub-batch, the gradients are calculated independently. This allows the system to avoid storing all gradients at once, reducing memory consumption.ter processing all sub-batches within the mini-batch, the individual gradients are accumulated. This accumulated gradient is then used to update the model's weights, effectively mimicking a full mini-batch update.

   CODE

```
training_args = TrainingArguments(per_device_train_batch_size=1,
gradient_accumulation_steps=4, **default_args)

trainer = Trainer(model=model, args=training_args, train_dataset=ds)
result = trainer.train()
print_summary(result)
```

```
# Training loop logic

for i in range(num_iterations):
    accumulated_gradients = 0
    for j in range(accumulation_steps):
        batch = next(training_batch)
        gradients = compute_gradients(batch)
        accumulated_gradients += gradients
    update_weights(accumulated_gradients)
```

5) **Quantization-Quantization and Low-Rank Adaptation (QLoRA)** - It significantly reduces model size, leading to lower memory requirements. For example Deepspeed s uses quantization techniques to convert model weights from higher precision (e.g., float32) to lower precision formats (e.g., fp16).
6) **Parallel Computation-Fully Sharded Data Parallel (FSDP) -** Deepspeed leverages data parallelism to distribute training across multiple nodes in a cluster environment (HPC).It harness the combined power of multiple GPUs, significantly accelerating the fine-tuning process

CREATING DATASET FROM JSON FILES - :

- Different model types require a different format of training data.
  For most chat completion models The training and validation data in use must be formatted as a JSON Lines (JSONL) document.
  the fine-tuning dataset must be formatted in the conversational format that is used by the Chat completions API

- Is the Dataset Compressed ? Decompress is first since it would be more efficient as for each training pass data won't be decompressed

- Convert Json File into Dataframe to get insights and select relevant features

EXAMPLE CODE

```
import pandas as pd
json file = "99 000000000000"
df = pd. read_ json(os. path. join (decompressed dir, json_file), lines=True)
print (df . head ())
```

- Structuring Data
- Uploading on Hugging Face
- Tokenize the text and leave the labels

```
from transformers import DistilBertTokenizerFast
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')
train_encodings = tokenizer(train_texts, truncation=True, padding=True)
val_encodings = tokenizer(val_texts, truncation=True, padding=True)
test_encodings = tokenizer(test_texts, truncation=True, padding=True)
```

we pass our texts to the tokenizer. pass truncation=True and padding=True, which will ensure that all of our sequences are padded to the same length and are truncated to be no longer than the model's maximum input length.

Converting tokens and labels into dataset objects-:
this is done by subclassing a torch.utils.data.Dataset object and implementing __len__ and __getitem__.

EXAMPLE CODE

```
import torch

class IMDbDataset(torch.utils.data.Dataset):
    def __init__(self, encodings, labels):
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        return len(self.labels)

train_dataset = IMDbDataset(train_encodings, train_labels)
val_dataset = IMDbDataset(val_encodings, val_labels)
test_dataset = IMDbDataset(test_encodings, test_labels)
```

Post that fine tuning

```
from transformers import DistilBertForSequenceClassification, Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir='./results',          # output directory
    num_train_epochs=3,              # total number of training epochs
    per_device_train_batch_size=16,  # batch size per device during training
    per_device_eval_batch_size=64,   # batch size for evaluation
    warmup_steps=500,                # number of warmup steps for learning rate scheduler
    weight_decay=0.01,               # strength of weight decay
    logging_dir='./logs',            # directory for storing logs
    logging_steps=10,
)

model = DistilBertForSequenceClassification.from_pretrained("distilbert-base-uncased")

trainer = Trainer(
    model=model,                         # the instantiated  Transformers model to be trained
    args=training_args,                  # training arguments, defined above
    train_dataset=train_dataset,         # training dataset
    eval_dataset=val_dataset             # evaluation dataset
)

trainer.train()
```

- IF THE JSON FILE HAS MULTIPLE COLUMNS THAT WOULD GO INTO TRAINING WE WOULD LOAD EACH OF THESE AND FURTHER TOKENIZE AND CREATE ENCODINGS

Example we could get insights from a task wherein the model was to be tuned on conversation .The dataset here is Stanford Question Answering Dataset (SQuAD) 2.0.

In this dataset each split is in a structured json file with a number of questions and answers for each passage (or context).
We will divide this dataset into three lists answers questions and context

EXAMPLE CODE
```
import json
from pathlib import Path

def read_squad(path):
    path = Path(path)
    with open(path, 'rb') as f:
        squad_dict = json.load(f)

    contexts = []
    questions = []
    answers = []
    for group in squad_dict['data']:
        for passage in group['paragraphs']:
            context = passage['context']
            for qa in passage['qas']:
                question = qa['question']
                for answer in qa['answers']:
                    contexts.append(context)
                    questions.append(question)
                    answers.append(answer)

    return contexts, questions, answers

train_contexts, train_questions, train_answers = read_squad('squad/train-v2.0.json')
val_contexts, val_questions, val_answers = read_squad('squad/dev-v2.0.json')
```

The contexts and questions are just strings. The answers are dicts containing the subsequence of the passage with the correct answer as well as an integer indicating the character at which the answer begins. In order to train a model on this data we need the tokenized context/question pairs, and integers indicating at which *token* positions the answer begins and ends.

We can manually write the code to get the integer positions for answer starting and ending
Now we can tokenize these answers, questions etc.
Tokenizers can accept parallel lists of sequences and encode them together as sequence pairs.

Example code

```
from transformers import DistilBertTokenizerFast
tokenizer = DistilBertTokenizerFast.from_pretrained('distilbert-base-uncased')

train_encodings = tokenizer(train_contexts, train_questions, truncation=True, padding=True)
val_encodings = tokenizer(val_contexts, val_questions, truncation=True, padding=True)
```

Here the tokenizer accepts train_contexts, train_questions two different lists parallelly and return as sequence pair

We now have text converted to tokens and starting ending position of answer in paragraph. We can convert this word position into tokens position
This could be done through a built in function char_to_token() present in Fast Tokenizers,

Now this data can be uploaded to Datasets
This could be done in two ways

1) Pytorch - : we can define a custom Dataset class which inherits from torch.utils.data.Dataset
2) Tensorflow -: , we can pass a tuple of (inputs_dict, labels_dict) to the from_tensor_slices method.

We should then choose and load a model . Once that is done we can train the model with Trainer / TFTrainer.

MULTI TURN CHAT FILE FORMAT
Multiple turns of a conversation in a single line of the jsonl training file is also supported.
To skip fine-tuning on specific assistant messages add the optional `weight` key value pair. Currently weight can be set to 0 or 1.

Retrieval-Augmented Generation

It aims to optimize the output of LLMs , so it references an authoritative knowledge base outside of the training data before generating the response.
We do not need to retrain the model on our dataset.

Underlying Principle

With RAG there is an information retrieval component in which the data is stored and the user input first pulls information from a new data source.
User Query and relevant information is given to the llm.
The LLM uses the new knowledge and its training data to create better responses.

Storing Data

Firstly we gather the external data . This may come from multiple data sources, such as APIs, databases, or document repositories. The data may exist in various formats like files, database records, or long-form text.

Data Embeddings

Converting Data into numerical embeddings and then store it into a vector Database.

Retrieve relevant Information

The user query is converted to a vector representation and matched with the vector databases. the system will retrieve relevant records.
The relevancy is calculated and established using mathematical vector calculations and representations

Augment the LLM Prompt

RAG model augments the user input (or prompts) by adding the relevant retrieved data in context. This step uses prompt engineering techniques to communicate effectively with the LLM. The augmented prompt allows the large language models to generate an accurate answer to user queries.

WHEN TO CHOSE RAG

Method	Requirements	Advantages	Cons
Prompt Engineering	None	Fast, cost-effective, no training required	Less control than fine-tuning
Retrieval augmented generation (RAG)	External knowledge base or database (e.g., vector database)	Dynamically updated context, enhanced accuracy	Increases prompt length and inference computation
Fine-tuning	Thousands of domain-specific or instruction examples	Granular control, high specialization	Requires labeled data, computational cost
Pretraining	Large datasets (billions to trillions of tokens)	Maximum control, tailored for specific needs	Extremely resource-intensive

Vector Database vs Vector Library - :

The difference between vector databases and vector libraries is that vector libraries are mostly used for static data, where the index data is immutable.
It is because vector libraries only store vector embeddings and not the associated objects they were generated from
Therefore they don't have CURD feature in them
Example of Vector Libraries are- FAISS ANNOY
Vector library with some CURD Feature but lacking dependent ecosystem - HNSWLib

SQL Database with Vector Support

Example - pgvector
It is a SQL database with vector support extensions
Cons -

- not scalable and efficient
- misalignment between the traditional relational structure of SQL databases and the nature of unstructured vector data.

Other Alternatives

- Full-text Search databases (ElasticSearch, OpenSearch)
- NoSQL databases with vector support (Redis, MongoDB)

Dedicated Vector Databases

- PineCone- closed-sourced offering limited scalability with its free plan
- Chroma- tailored for audio data and lacks textual data optimization
- Vearch and Vald

Best options

- Weaviate
- Milvus
- Qdrant
- Vespa
- Vector DB

Leading RAG Algorithms - :

REALM -REtrieval Augmented Language Model,

- It augments a standard T5 language model which serves as the core generative module, by enabling it to incorporate evidence passages retrieved from Wikipedia in a lightweight, efficient manner.
  Workflow-
  1. Input Question
  2. Retrieve Relevant Passages-Sparse vector index retrieval using BM25 to fetch top k Wikipedia passages given question embeddings.
  3. Encoding Question and Answer Independently by RoBERTa

4. Joint Contextualization-Encoded vectors interact via cross-attention layers to produce final contextualized representations that power output text generation.

ORQA: Optimized Retrieval Question Answering

- It uses BERT encoder optimized for question representation and REALM encoder optimized for evidence passages
- This results in significantly more efficient and accurate encoding of questions and evidence passages

RAG TOKEN

It reformulates rag as a single sequence-to-sequence task rather than retrieving full passages, RAG Token distills external knowledge into succinct subject-relation-object triplets. This allows scaling to huge knowledge graphs as the retrieval source.

RAG via Langchain

Required Libraries and models

- Loading Data- Document Loader and Text Loader
- Collecting data from a url - Request Library
- Chunking the data/documents- CharacterTextSplitter
- Embed and store chunks - using OpenAI Embeddings and different vector Database
- .from_documents() - Populate chunks with Vector Database

EXAMPLE CODE

```
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Weaviate
import weaviate
from weaviate.embedded import EmbeddedOptions

client = weaviate.Client(
 embedded_options = EmbeddedOptions()
)

vectorstore = Weaviate.from_documents(
   client = client,
   documents = chunks,
   embedding = OpenAIEmbeddings(),
   by_text = False
)
```

- Retrieval- Define the populated vector Database as retriever .
- Augment- define a prompt template and augment the prompt with the additional context

EXAMPLE CODE

```
from langchain.prompts import ChatPromptTemplate
```

```
template = """You are an assistant for question-answering tasks.
Use the following pieces of retrieved context to answer the question.
If you don't know the answer, just say that you don't know.
Use three sentences maximum and keep the answer concise.
Question: {question}
Context: {context}
Answer:
"""
prompt = ChatPromptTemplate.from_template(template)

print(prompt)
```

- Generate- build a chain for the RAG pipeline, chaining together the retriever, the prompt template and the LLM. after that invoke it to generate output

  EXAMPLE CODE

```
from langchain.chat_models import ChatOpenAI
from langchain.schema.runnable import RunnablePassthrough
from langchain.schema.output_parser import StrOutputParser

llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)

rag_chain = (
   {"context": retriever,  "question": RunnablePassthrough()}
   | prompt
   | llm
   | StrOutputParser()
)

query = "What did the president say about Justice Breyer"
rag_chain.invoke(query)
```

ADVANCED RAG TECHNIQUES

Adaptive RAG Techniques-
It dynamically selects the most suitable method for large language models (LLMs) based on query complexity.
Adaptive-RAG selects the most appropriate strategy, whether it is iterative and single-step retrieval-augmented procedures or completely bypassing retrieval.

LangChain ReACT

- It enables LLMs to interface with external tools, hence improving decision-making processes.
- ReAct is ideal for scenarios where LLM has to rely on external tools and agent and have to interact with them to fetch information for various reasoning steps.
- In scenarios like QA systems, where LLMs might not always provide correct answers, the interaction with an external search engine becomes crucial, and ReAct proves invaluable.

EXAMPLE IMPLEMENTATION OF ADAPTIVE RAG USING COHERE LLM AND TAVILY SERP

```
! pip install --quiet langchain langchain_cohere tiktoken chromadb pymupdf
### Set API Keys
import os

os.environ["COHERE_API_KEY"] = "Cohere API Key"
os.environ["TAVILY_API_KEY"] = "Tavily API Key"
from langchain_community.tools.tavily_search import TavilySearchResults

internet_search = TavilySearchResults()
internet_search.name = "internet_search"
internet_search.description = "Returns a list of relevant document snippets for a textual query
retrieved from the internet."

from langchain_core.pydantic_v1 import BaseModel, Field

class TavilySearchInput(BaseModel)
    query: str = Field(description="Query to search the internet with")


internet_search.args_schema = TavilySearchInput

from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_cohere import CohereEmbeddings
#from langchain_community.document_loaders import WebBaseLoader
from langchain_community.document_loaders import PyMuPDFLoader
from langchain_community.vectorstores import Chroma
# Set embeddings
embd = CohereEmbeddings()
# Load Docs to Index
loader = PyMuPDFLoader('/content/cleartax-in-s-income-tax-slabs.pdf') #PDF Path
from langchain.tools.retriever import create_retriever_tool
vectorstore_search = create_retriever_tool(retriever=vectorstore_retriever,
name="vectorstore_search",description="Retrieve relevant info from a vectorstore that contains
documents related to Income Tax of India New and Old Regime Rules")
from langchain.agents import AgentExecutor
from langchain_cohere.react_multi_hop.agent import create_cohere_react_agent
from langchain_core.prompts import ChatPromptTemplate
```

```
# LLM
from langchain_cohere.chat_models import ChatCohere
chat = ChatCohere(model="command-r-plus", temperature=0.3)
# Preamble
preamble = """
You are an expert who answers the user's question with the most relevant datasource.
You are equipped with an internet search tool and a special vectorstore of information about
Income Tax Rules and Regulations of India.
If the query covers the topics of Income tax old and new regime India Rules and regulations
then use the vectorstore search.
"""

# Prompt
prompt = ChatPromptTemplate.from_template("{input}")

# Create the ReAct agent
agent = create_cohere_react_agent(
    llm=chat,
    tools=[internet_search, vectorstore_search],
    prompt=prompt,
)
# Agent Executor

agent_executor = AgentExecutor(
    agent=agent, tools=[internet_search, vectorstore_search], verbose=True
)
output = agent_executor.invoke(
    {
        "input": "What is the general election schedule of India 2024?",
        "preamble": preamble,
    }
)

print(output)

print(output['output'])
```

OUTPUT PARSERS

Output parsers are responsible for taking the output of an LLM and transforming it to a more suitable format. The major types of Output Parsers are as follows - :

1) StrOutputParser

2) OpenAI Functions Parsers - These parsers handle outputs from OpenAI's function calls, focusing on the function_call and arguments parameters to generate structured responses.

StrOutputParser

In Langchain StrOutputParser Is used to convert the output of a language model, whether from an LLM or a ChatModel, into a string format. This conversion is crucial for applications that require a uniform output format for further processing or display to end-users.

It creates a new model by parsing and validating input data from keyword arguments. Raises ValidationError if the input data cannot be parsed to form a valid model.

Outputs of StrOutputParser

- If the model's output is already a string, the StrOutputParser simply passes this string through without modification.
- In cases where the output is a ChatModel message, the StrOutputParser extracts the .content attribute of the message

EXAMPLE CODE

```
from langchain_core.output_parsers import StrOutputParser
output_parser = StrOutputParser()
chain = prompt | llm | output_parser
chain.invoke({"input": "how can langsmith help with testing?"})
```

PARAMETERS AND OUTPUT

Parameters
- inputs (*List[Input]*) –
- config (*Optional[Union[RunnableConfig, List[RunnableConfig]]]*) –
- return_exceptions (*bool*) –
- kwargs (*Optional[Any]*) –

Return type
List[*Output*]

MultiQueryRetriever

MultiQueryRetriever automates the process of prompt tuning by using an LLM to generate multiple queries from different perspectives for a given user input query.

It retrieves a set of relevant documents and takes the unique union across all queries to get a larger set of potentially relevant documents.By generating multiple perspectives on the same question, the MultiQueryRetriever overcomes limitations of distance-based retrieval and gets a richer set of results.

This method doesn't rely on a singular set of documents retrieved for an initial query to produce the final output

EXAMPLE CODE

```
from langchain.retrievers.multi_query import MultiQueryRetriever
from langchain_openai import ChatOpenAI

question = "What are the approaches to Task Decomposition?"
llm = ChatOpenAI(temperature=0)
retriever_from_llm = MultiQueryRetriever.from_llm(
    retriever=vectordb.as_retriever(), llm=llm
)
```

OTHER ADVANCED RETRIEVERS -

Contextual Compression -

Contextual compression in LangChain is a technique used to compress and filter documents based on their relevance to a given query.

LLM ChainFilter

The LLMChainFilter in LangChain is a component used for filtering and processing documents based on their relevance to a given query.

EmbeddingsFilter

The EmbeddingsFilter provides a cheaper and faster option by embedding the documents and query and only returning those documents which have sufficiently similar embeddings to the query.

DocumentCompressorPipeline

It helps in compressing and transforming documents in a contextual manner. The pipeline can include compressors like EmbeddingsRedundantFilter to remove redundant documents based on embedding similarity, and EmbeddingsFilter to filter documents based on their similarity to the query. Document transformers like TextSplitter can be used to split documents into smaller pieces.

Ensemble Retriever

It is used to improve the performance of retrieval by leveraging the strengths of different algorithms. . It is particularly useful when combining a sparse retriever (e.g., BM25) with a dense retriever (e.g., embedding similarity) because their strengths are complementary.

MultiVector Retriever

It allows to store multiple vectors per document

Parent Document Retriever

It splits and stores small chunks of data. During retrieval, it first fetches the small chunks but then looks up the parent ids for those chunks and returns those larger documents.

Time-weighted vector store

This retriever uses a combination of semantic similarity and a time decay.

Vector store-backend retriever

It is a lightweight wrapper around the vector store class to make it conform to the retriever interface. It uses the search methods implemented by a vector store, like similarity search and MMR, to query the texts in the vector store.

WebSearchRetriever

RunnablePassthrough

RunnablePassthrough allows to pass inputs unchanged. This typically is used in conjunction with RunnableParallel to pass data through to a new key in the map.

This runnable behaves almost like the identity function, except that it can be configured to add additional keys to the output, if the input is a dict.

allows to pass inputs unchanged or with the addition of extra keys

EXAMPLE CODE

```
from langchain_core.runnables import RunnableParallel, RunnablePassthrough

runnable = RunnableParallel(
  passed=RunnablePassthrough(),
  modified=lambda x: x["num"] + 1,
)

runnable.invoke({"num": 1})
```

OUTPUT

```
{'passed': {'num': 1}, 'extra': {'num': 1, 'mult': 3}, 'modified': 2}
```

PARAMETERS AND RETURN TYPE

Parameters

- inputs (*List[Input]*) –
- config (*Optional[Union[RunnableConfig, List[RunnableConfig]]]*) –
- return_exceptions (*bool*) –
- kwargs (*Optional[Any]*) –

Return type

List[*Output*]