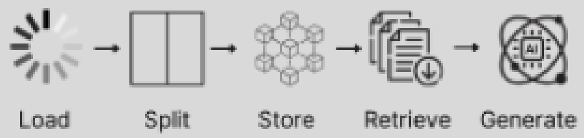
A Complete Cheatsheet LangChain

RAG Pipeline



Quickstart: Loading

```
loader = PyMuPDFLoader("example_data/layout-parser-paper.pdf")
docs = loader.load()
```

Quickstart: Splitting

```
text_splitter = RecursiveCharacterTextSplitter
                (chunk_size=1000, chunk_overlap=200)
splits = text_splitter.split_documents(docs)
```

Loading

Document loaders provide a "load" method for loading data as documents from a configured source

```
import bs4

from langchain_community.document_loaders import WebBaseLoader

bs4_strainer = bs4.SoupStrainer(class_=("post-title", "post-header", "post-content"))

loader = WebBaseLoader(
 web_paths=("https://lilianweng.github.io/posts/2023-06-23-agent/",),
 bs_kwargs={"parse_only": bs4_strainer},)

docs = loader.load()

from langchain_community.document_loaders import PyMuPDFLoader

loader = PyMuPDFLoader("example_data/layout-parser-paper.pdf")
data = loader.load()

loader = DirectoryLoader('../', glob="**/*.md", use_multithreading=True)
docs = loader.load()
```

Quickstart: Storing

```
vectorstore = Chroma.from_documents
              (documents=splits,
               embedding=OpenAIEmbeddings())
```

Quickstart: Retrieving

```
retriever = vectorstore.as_retriever()

prompt = hub.pull("rlm/rag-prompt")

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

def format_docs(docs):
   return "\n\n".join(doc.page_content for doc in docs)
```

Quickstart: Generation

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

rag_chain.invoke("What is Task Decomposition?")
```

Split

Text splitters are tools designed to segment long pieces of text into smaller, manageable chunks while maintaining semantic coherence


```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
 chunk_size=1000, chunk_overlap=200,
add_start_index=True)

all_splits = text_splitter.split_documents(docs)

python_splitter =
RecursiveCharacterTextSplitter.from_language(
 language=Language.PYTHON, chunk_size=50,
chunk_overlap=0)

python_docs =
python_splitter.create_documents([PYTHON_CODE])
python_docs

text_splitter = SemanticChunker(OpenAIEmbeddings())
docs = text_splitter.create_documents([File])
```

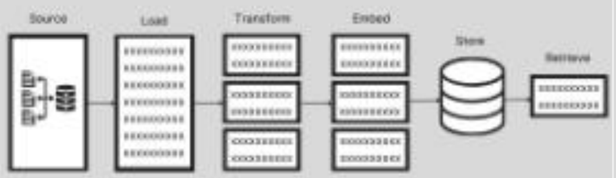
Vector Store

Vector stores are databases designed to efficiently store and retrieve high-dimensional vector embeddings of text for fast similarity search and information retrieval tasks.


```
from langchain_community.vectorstores import Chroma
from langchain_openai import OpenAIEmbeddings

vectorstore =
Chroma.from_documents(documents=all_splits,
embedding=OpenAIEmbeddings())

docs = vectorstore.similarity_search(query)
embedding_vector =
OpenAIEmbeddings().embed_query(query)
docs =
vectorstore.similarity_search_by_vector(embedding_vector)

from langchain_community.vectorstores import Qdrant
found_docs = await
qdrant.amax_marginal_relevance_search(query, k=2,
fetch_k=10)
```

Retrieve

Retrievers are components that fetch relevant information from a database based on a query, to find the most pertinent documents.




```
retriever = db.as_retriever(search_type="mmr")

retriever = db.as_retriever(
search_type="similarity_score_threshold",
search_kwargs={"score_threshold": 0.5})
retriever = db.as_retriever(search_kwargs={"k": 3})

# parent document retriever

retriever = ParentDocumentRetriever(
 vectorstore=vectorstore,
 docstore=InMemoryStore(),
 child_splitter=child_splitter)

# Time-weighted retriever

retriever = TimeWeightedVectorStoreRetriever(
 vectorstore=vectorstore,
decay_rate=0.0000000000001, k=1)
```

Generate

Generators are models that produce text content, to create responses, summaries, or other forms of generated text based on input data or prompts.


```
llm = Ollama(model="llama2")

llm.invoke("Tell me a joke")

llm = HuggingFaceEndpoint(
 repo_id="mistralai/Mistral-7B-Instruct-v0.2",
max_length=128, temperature=0.5,
token=HUGGINGFACEHUB_API_TOKEN)

llm_chain = LLMChain(prompt=prompt, llm=llm)

print(llm_chain.run(question))

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

for chunk in chat.stream("Write me a song about goldfish on the moon"):
    print(chunk.content, end="", flush=True)
```

Adding Sources

```
rag_chain_from_docs = (
 RunnablePassthrough.assign(context=(lambda x:
format_docs(x["context"])))
 | prompt
 | llm
 | StrOutputParser()
)

rag_chain_with_source = RunnableParallel(
 {"context": retriever, "question":
RunnablePassthrough()}
).assign(answer=rag_chain_from_docs)
rag_chain_with_source.invoke("What is Task
Decomposition")
```

Memory

```
conversation_with_summary = ConversationChain(
 llm=OpenAI(temperature=0),
 memory=ConversationBufferWindowMemory(k=5),
 verbose=True)
conversation_with_summary.predict(input="Hi, what's up?")
conversation_with_kg = ConversationChain(
 llm=llm, verbose=True, prompt=prompt,
memory=ConversationKGMemory(llm=llm))

conversation_with_kg.predict(input="Hi, what's up?")
```

LangChain Expression Language

LCEL makes it easy to build complex chains from basic components

```
from langchain_core.output_parsers import StrOutputParser

from langchain_core.prompts import ChatPromptTemplate

from langchain_openai import ChatOpenAI

prompt = ChatPromptTemplate.from_template("tell me a short joke about {topic}")

model = ChatOpenAI(model="gpt-4")

output_parser = StrOutputParser()

chain = prompt | model | output_parser

chain.invoke({"topic": "ice cream"})
```

Agents and Tools

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent

tools = load_tools(["wikipedia", "llm-math"], llm=llm)
agent = initialize_agent(tools, llm, agent="zero-shot-react-description", verbose=True)

agent.run("Can you tell me the distance between Earth and the moon? And could you please convert it into miles? Thank you.")
```

LangChain onepager

import langchain

Open in Colab

Models | Prompts | Indexes | Memory | Chains | Agents & Tools

Calling OpenAI large language model — Models

```
from langchain.llms import OpenAI
llm = OpenAI( model_name="text-davinci-003",  temperature=0.01)
llm("Suggest 3 bday gifts for a data scientist")
>>> 1. A subscription to a data science magazine
>>> 2. A set of data science books
>>> 3. A data science-themed mug or t-shirt
```

Conversation schemas: History and Instructions

```
from langchain.chat_models import ChatOpenAI
from langchain.schema import HumanMessage,AIMessage,SystemMessage
chat = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0.01)
conversation_history = [
  HumanMessage(content="Suggest 3 bday gifts for a data scientist"),
  AIMessage(content="What is your price range?"),
  HumanMessage(content="Under 100$") ]
chat(conversation_history).content
>>> 1. A data science book: Consider gifting a popular and highly ...
>>> 2. Data visualization tool: A data scientist often deals with ...
>>> 3. Subscription to a data science platform: Give them access to ....
system_instruction = SystemMessage(content = """You work as an assistant
in an electronics store. Your income depends on the items you sold""")
user_message = HumanMessage(content="3 bday gifts for a data scientist")
chat([system_instruction, user_message]).content
>>> 1. Laptop: A high-performance laptop with a powerful processor ....
>>> 2. External Hard Drive: Data scientists deal with large datasets ....
>>> 3. Data Science Books: Books related to data science can be ....
```

Open-source models

```
from auto_gptq import AutoGPTQForCausalLM, BaseQuantizeConfig
from transformers import AutoTokenizer, AutoModelForCausalLM
model_name = "TheBloke/llama-2-13B-Guanaco-QLoRA-GPTQ"
tokenizer = AutoTokenizer.from_pretrained(model_name, use_fast=True)
# Initialize the AutoGPTQForCausalLM model with appropriate parameters
model = AutoGPTQForCausalLM.from_quantized(
  model_name, use_safetensors=True, trust_remote_code=True,
  device_map="auto", quantize_config=None )
# Tokenize the query and convert to CUDA tensor
input_ids = tokenizer(query, return_tensors="pt").input_ids.cuda()
# Generate text using the model with specified settings
output = model.generate(inputs=input_ids, temperature=0.1)
```

Text generation parameters

The temperature parameter affects the randomness of the token generation
Top-k sampling limits token generation to the top k most likely at each step
Top-p (nucleus) sampling limits token generation to cumulative probability *p*
The length of generated tokens can be specified by max_tokens parameter

```
llm = OpenAI(temperature=0.5, top_k=10, top_p=0.75, max_tokens=50)
```

Quantization

```
from transformers import BitsAndBytesConfig
# Configure BitsAndBytesConfig for 4-bit quantization
bnb_config = BitsAndBytesConfig(
  load_in_4bit=True,  bnb_4bit_compute_dtype=torch.bfloat16,
  bnb_4bit_quant_type="nf4", bnb_4bit_use_double_quant=True)
model_4bit = AutoModelForCausalLM.from_pretrained(
  model_name_or_path, quantization_config=bnb_config,
  device_map="auto", trust_remote_code=True)
```

Fine-tuning — Models

```
from peft import LoraConfig, get_peft_model, prepare_model_for_kbit_training
pretrained_model = AutoModelForCausalLM.from_pretrained(...)
pretrained_model.gradient_checkpointing_enable()
model = prepare_model_for_kbit_training(pretrained_model)
# Specify LoRa configuration
config = LoraConfig(r=16, lora_alpha=32, lora_dropout=0.05, bias="none",
target_modules=["query_key_value"], task_type="CAUSAL_LM")
model = get_peft_model(model, config)
# Set training parameters
trainer = transformers.Trainer(
  model=model, train_dataset=train_dataset,
  args=transformers.TrainingArguments(
    num_train_epochs=10, per_device_train_batch_size=8, ...),
  data_collator=transformers.DataCollatorForLanguageModeling(tokenizer))
model.config.use_cache = False
trainer.train()
```

Prompt Templates — Prompts

```
from langchain.prompts import PromptTemplate
# Define the template for SEO description
template = "Act as an SEO expert. Provide a SEO description for {product}"
# Create the prompt template
prompt = PromptTemplate(input_variables=["product"], template=template)
# Pass in an input to return a formatted prompt
formatted_prompt = prompt.format(product="Electric Scooter")
llm(formatted_prompt)
>>> The Electric Scooter is the perfect way to get around town quickly ...
formatted_prompt = prompt.format(product="Perpetuum Mobile")
llm(formatted_prompt)
>>> Perpetuum Mobile is an innovative product that provides a ...
```

```
from langchain.prompts import FewShotPromptTemplate
# Define three examples for the 3-shot learning
examples = [
  {"email_text": "Win a free iPhone!", "category": "Spam"},
  {"email_text": "Next Sprint Planning Meeting.", "category": "Meetings"},
  {"email_text": "Version 2.1 of Y is now live", "category": "Project Updates"}]
# Create a PromptTemplate for classifying emails
prompt_template = PromptTemplate(template="Classify the email:
{email_text}/n{category}", input_variables=["email_text", "category"])
# Create a FewShotPromptTemplate using PromptTemplate and examples
few_shot_prompt = FewShotPromptTemplate(example_prompt =
prompt_template, examples = examples, suffix = "Classify the email:
{email_text}", input_variables=["email_text"])
```

Document loaders — Indexes

```
from langchain.document_loaders import csv_loader, DirectoryLoader,
  WebBaseLoader, JSONLoader, UnstructuredPDFLoader, .......
loader = DirectoryLoader('../', glob="**/*.md")
loader = csv_loader.CSVLoader(...)
loader = WebBaseLoader(...)
loader = JSONLoader(...)
loader = UnstructuredPDFLoader(...)
loaded_documents = loader.load()
```

Retrievers and Vectorstores

```
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.vectorstores import FAISS, Chroma, Pinecone, ...
# Split docs into texts
splitter = RecursiveCharacterTextSplitter(chunk_size=800, chunk_overlap=50)
texts = splitter.split_documents(loaded_documents)
# Embed your texts and store them in a vectorstore
db = FAISS.from_documents(texts, embeddings)
db = FAISS.from_texts(["some_string_abc", "some_string_xyz"], embeddings)
# Perform similarity search
db.similarity_search(query)
# Initialize retriever and ask for relevant documents back
retriever = db.as_retriever()
docs = retriever.get_relevant_documents(some_query)
```

Setup Memory — Memory

```
from langchain.memory import ConversationBufferMemory
memory = ConversationBufferMemory(memory_key="chat_history")
# Setup predefined memories
memory.chat_memory.add_user_message("Hi!")
memory.chat_memory.add_ai_message("Welcome! How can I help you?")
memory_variables = memory.load_memory_variables({...})
# Add response to memory
memory.add_ai_message(chat_response.content)
```

Chains — Chains

```
from langchain.chains import ConversationChain, summarize, question_answering
from langchain.schema import StrOutputParser
# Templates for summarizing customer feedback and drafting email response
feedback_summary_prompt = PromptTemplate.from_template(
  """You are a customer service manager. Summarize the customer feedback.
Customer Feedback: {feedback}
Summary:""")
email_prompt = PromptTemplate.from_template(
  """You are a customer service representative. Given the summary of
customer feedback, it is your job to write a professional email response.
Feedback Summary: {summary}
Email Response:""" )
feedback_chain = feedback_summary_prompt | llm | StrOutputParser()
summary_chain = ({"summary": feedback_chain} | email_prompt | llm | StrOutputParser())
summary_chain.invoke({"feedback": "Incorrect item has arrived"})
```

```
# Predefined chains: summarization and Q&A
chain = summarize.load_summarize_chain(llm, chain_type="stuff")
chain.run(loaded_documents)
chain = question_answering.load_qa_chain(llm, chain_type="stuff")
chain.run(input_documents=loaded_documents, question = <input>)
# Use memory
conversation=ConversationChain(llm=llm,memory=ConversationBufferMemory())
conversation.run("Name the tallest mountain in the world")   >>> Everest
conversation.run("How high is it?")                          >>> 8848 m
```

Tools — Agents and Tools

```
from langchain.agents import load_tools
tools = load_tools(["serpapi", "llm-math", ...], llm=llm)
from langchain.tools import StructuredTool, BaseTool
def multiply_two_numbers(a: float, b: float) -> float:
  """multiply two numbers"""
  return a * b
multiplier_tool = StructuredTool.from_function(multiply_two_numbers)
```

Agents

```
from langchain.agents import initialize_agent, AgentType, BaseSingleActionAgent
agent = initialize_agent(
      tools, llm, agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION )
agent.run({"input": "How old would Harry Potter be when Daniel
Radcliffe was born?"})                    >>>9
# create own agents and tools
class UnitConversionTool(BaseTool):
  name = "Unit Conversion Tool"
  description = "Converts American units to International units"
  def _run(self, text: str):
    def miles_to_km(match):
      miles = float(match.group(1))
      return f"{miles * 1.60934:.2f} km"
    return re.sub(r'\b(\d+(\.\d+)?)\s*(miles|mile)\b', miles_to_km, text)
  def _arun(self, text: str):
    raise NotImplementedError("No async yet")
agent = initialize_agent(
  agent='chat-conversational-react-description',
  tools=[UnitConversionTool()],
  llm=llm,
  memory=memory
)
agent.run("five miles")
>>> 8.05 kilometers
```

v1.1.1 – 12.11.2023
latest version (click) →
by Ivan Reznikov
medium.com/@ivanreznikov
linkedin.com/in/reznikovivan