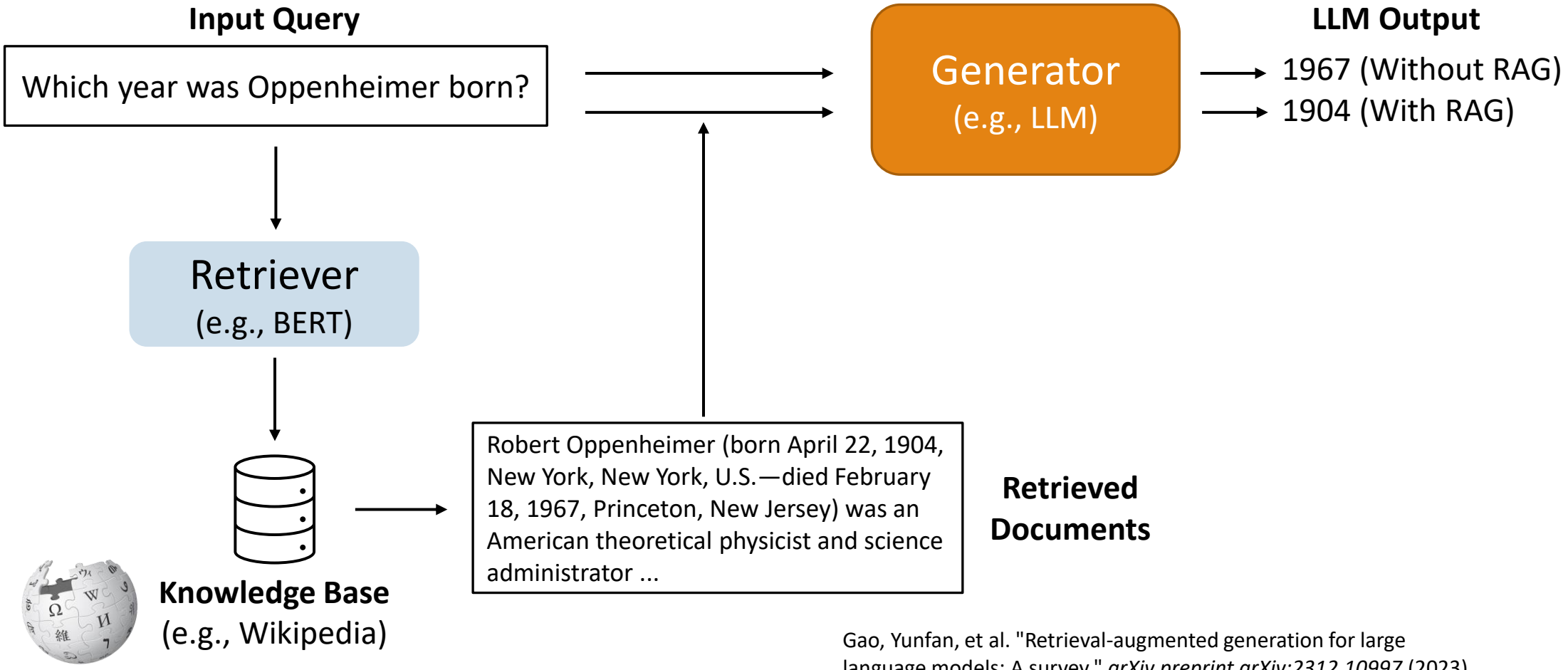
Natural Language Processing

RAG Tutorial 1 (on Colab) 2024/11/28

Outline

- Introduction to LangChain, Ollama

Retrieval-Augmented Generation (RAG)



Gao, Yunfan, et al. "Retrieval-augmented generation for large language models: A survey." *arXiv preprint arXiv:2312.10997* (2023).

Introduction to LangChain

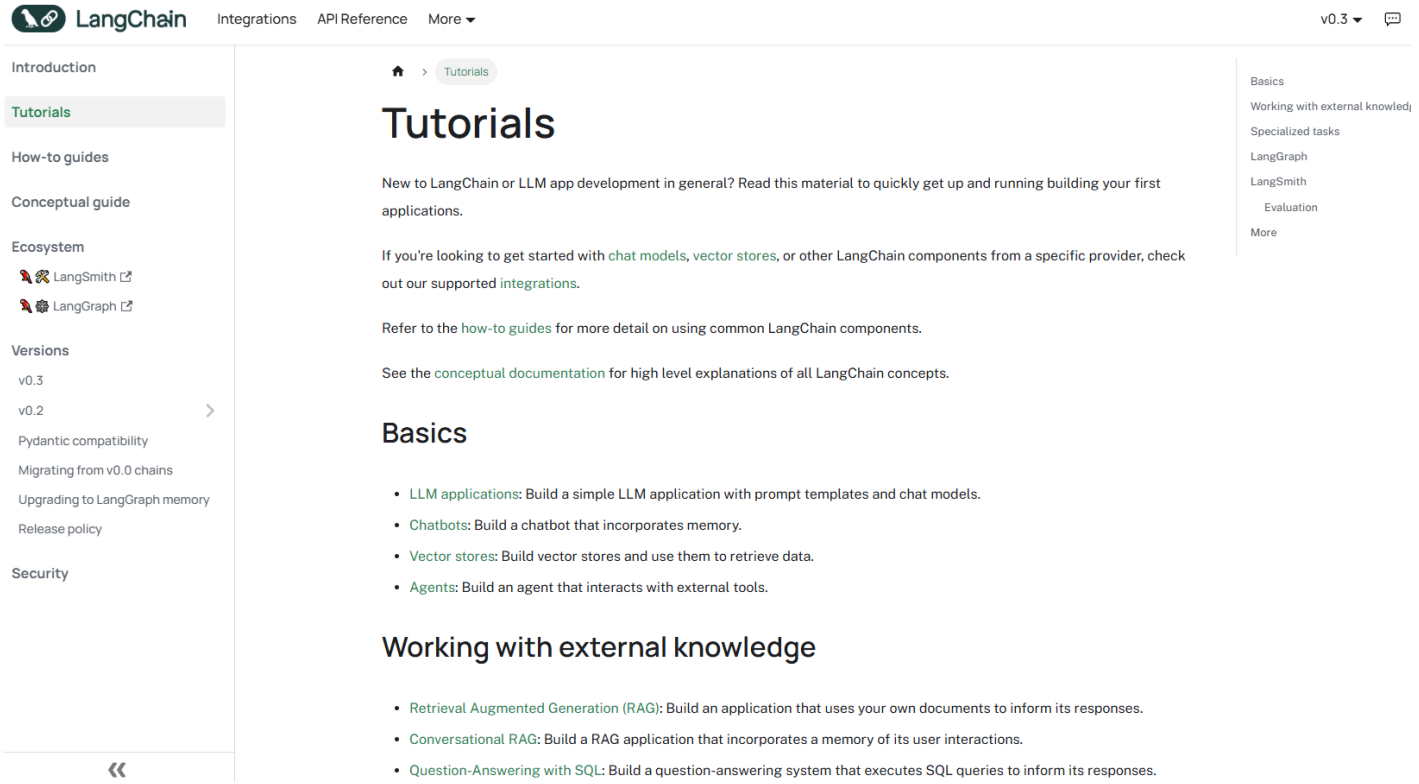
- LangChain is a framework for developing applications powered by large language models (LLMs).
  - Official Website: https://www.langchain.com/
  - Source code: https://github.com/langchain-ai/langchain

Introduction to LangChain

- For these applications, LangChain simplifies the entire application lifecycle:
  - Open-source libraries: core abstractions, third-party integrations, and application-building tools
  - Productionization: provide a developer platform that lets you debug, test, evaluate, and monitor chains built on any LLM.
  - Deployment: turn the applications into production-ready APIs and Assistants.

Introduction to LangChain

- You can build a personalized end-to-end RAG system based on the LangChain Official Tutorial.



Introduction to Ollama

- Ollama is a user-friendly interface for running large language models (LLMs) locally, specifically on MacOS, Linux, and Windows.
  - Official Website: https://ollama.com/
  - Source code: https://github.com/ollama/ollama

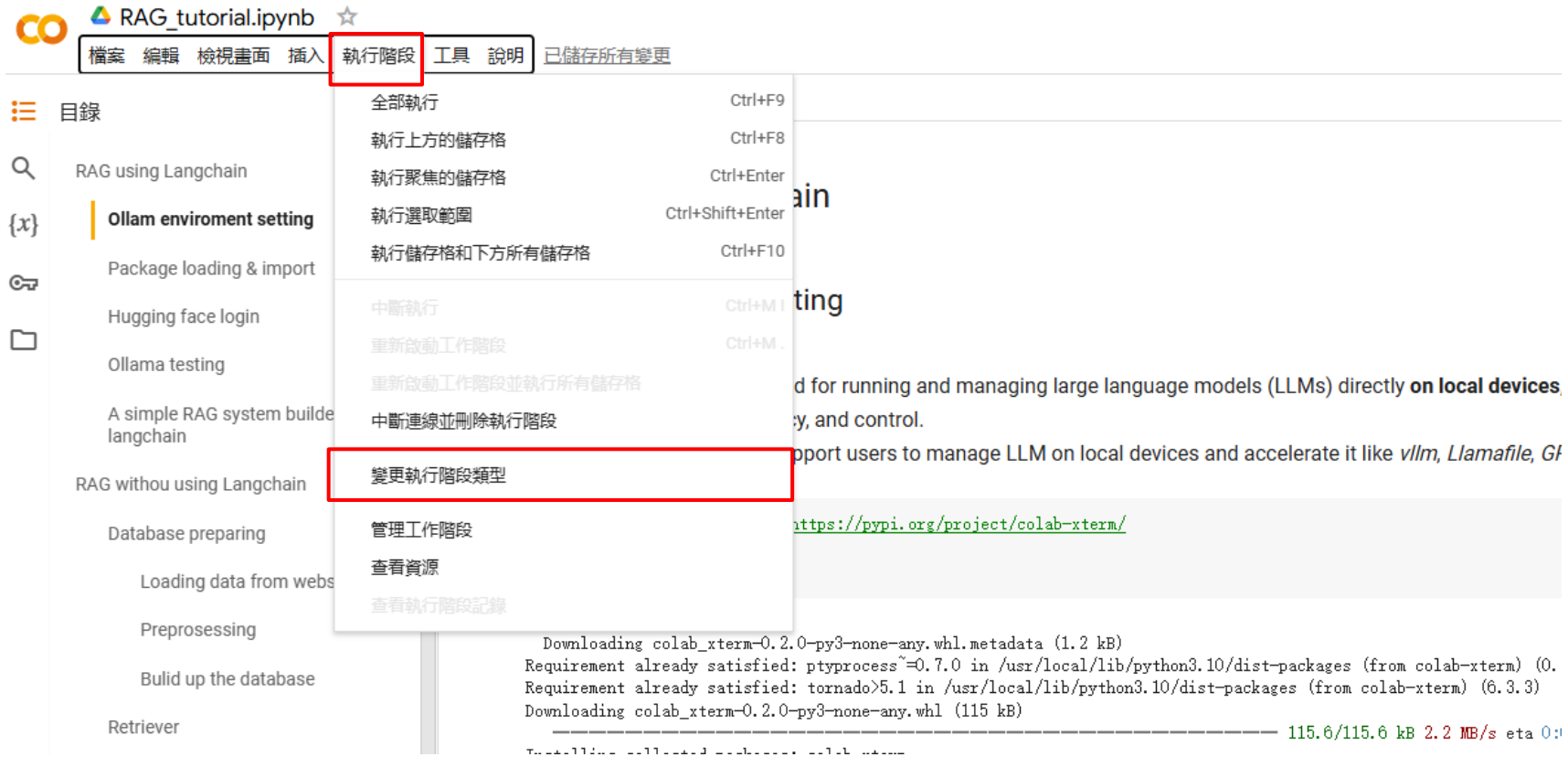
Introduction to Ollama

- **Ease of Use**: Ollama is easy to install and use.
- **Versatility and Model Installation**: Ollama supports a wide range of models, making it versatile for various applications.
- **GPU Acceleration**: Ollama leverages GPU acceleration. This feature is included out of the box and it requires zero intervention.
- **Integration Capabilities**: Ollama is compatible with several platforms like **Langchain**, llama-index, and more.
- **Privacy and Cost**: Running LLMs **locally** with Ollama ensures data privacy as your data is not sent to a third party.

Prerequisites

Setting Up Google Colab with T4 GPU and High RAM

- Runtime > Change runtime type



Prerequisites

Setting Up Google Colab with T4 GPU and High RAM

- Runtime type > Python3, Hardware accelerator > T4 GPU (GPU time for free is limited)

Packages loading & import

- Install and import all the necessary packages.

```
[9]  !pip install langchain
     !pip install langchain_community
     !pip install langchain_huggingface
     !pip install langchain_text_splitters
     !pip install langchain_chroma
     # !pip install pyserini
     !pip install rank-bm25
     !pip install huggingface_hub
```

40 秒

```
nltk.download('punkt')
nltk.download('punkt_tab')
```

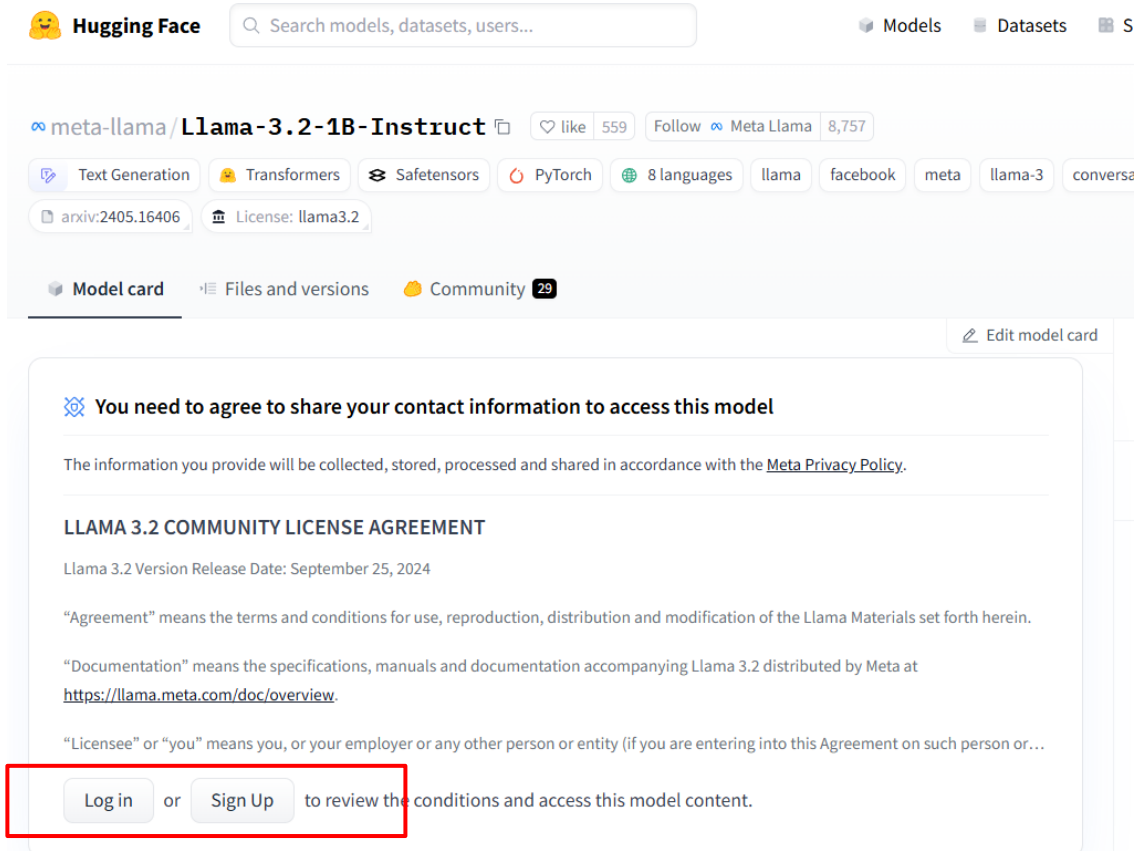
0 秒

```
import os
import json
import bs4
import nltk
import torch
import pickle
import numpy as np

# from pyserini.index import IndexWriter
# from pyserini.search import SimpleSearcher
from numpy.linalg import norm
from rank_bm25 import BM25Okapi
from nltk.tokenize import word_tokenize

from langchain_community.llms import Ollama
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain.chains import create_retrieval_chain
from langchain.vectorstores import Chroma
from sentence_transformers import SentenceTransformer
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.embeddings import JinaEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.docstore.document import Document
from langchain_core.prompts import ChatPromptTemplate
from langchain_community.document_loaders import WebBaseLoader
from transformers import AutoModel, AutoModelForCausalLM, AutoTokenizer

from tqdm import tqdm
```

34 秒

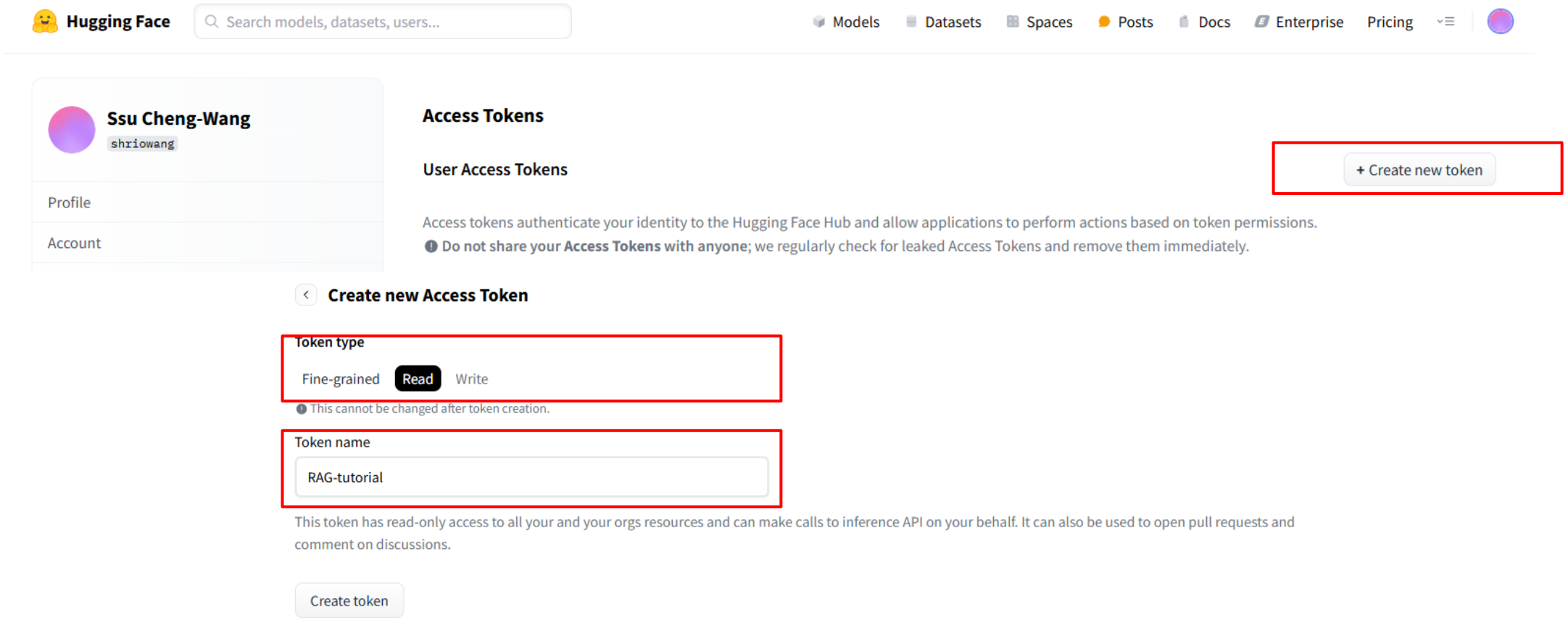
Hugging face login

- In this tutorial, we will need to apply the access to the model llama-3.2-1B-Instruct.
- After you get the access, you must log-in through hf-token.



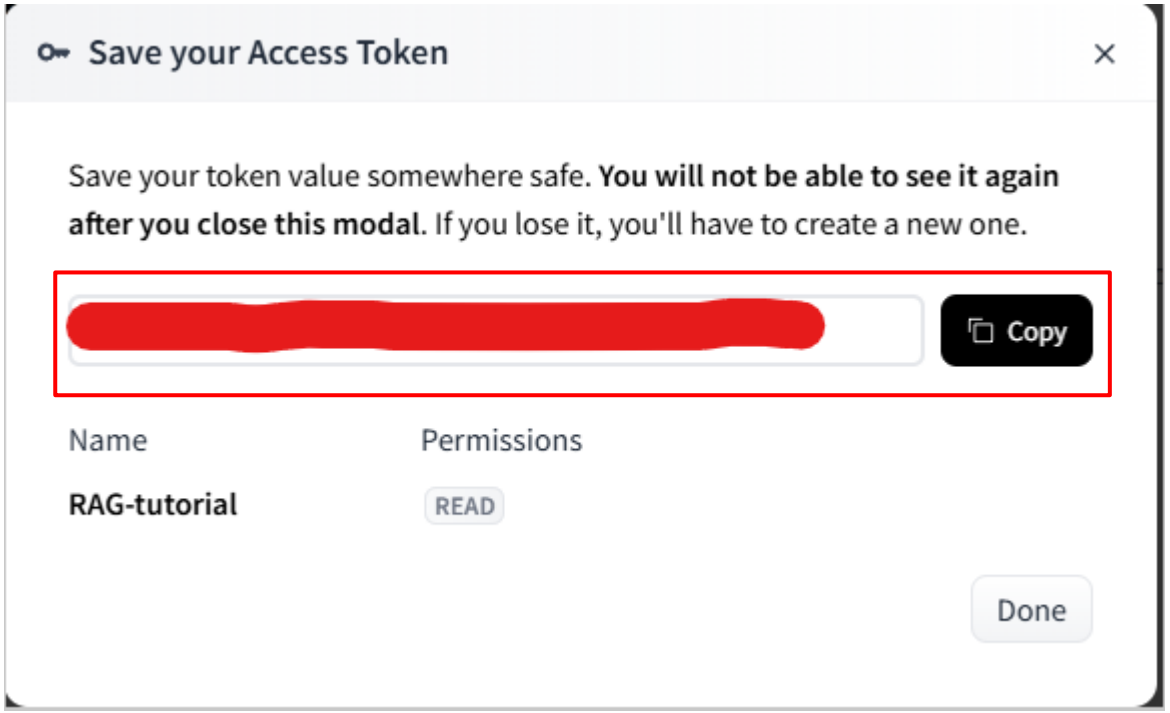
Hugging face login

- If you haven't created any tokens, create a new one.
- Select the token type and give a name.



Hugging face login

- Then you will get your own access token, save it because you will not be able to see it again.
- Back to colab, replace the hf_token with yours.




```
from huggingface_hub import login

hf_token = "hf_***" # @param {type:"string"}
login(token=hf_token, add_to_git_credential=True)
```

```
[18] !huggingface-cli whoami
```

```
shriowang
orgs: IKMMOE
```

Install Ollama on Colab

- First, install the colab-xterm package and load the extension.
- Colab-xterm allows you to open a terminal in a cell.

```
!pip install colab-xterm #https://pypi.org/project/colab-xterm/
%load_ext colabxterm
```

- Thus, you can open a terminal window within Colab.

```
%xterm
```

```
Launching Xterm...
```

Install Ollama on Colab

- Download Ollama package:

```
/content# curl -fsSL https://ollama.com/install.sh | sh
>>> Installing ollama to /usr/local
>>> Downloading Linux amd64 bundle
######################################################################################################## 100.0%
>>> Creating ollama user...
>>> Adding ollama user to video group...
>>> Adding current user to ollama group...
>>> Creating ollama systemd service...
WARNING: systemd is not running
WARNING: Unable to detect NVIDIA/AMD GPU. Install lspci or lshw to automatically detect and install GPU dependencies.
>>> The Ollama API is now available at 127.0.0.1:11434.
>>> Install complete. Run "ollama" from the command line.
/content#
```

- Activate Ollama serve. (If you idle for a long time, the connection would be closed forcedly. If so, run "ollama serve" again.):

```
/content# ollama serve
Couldn't find '/root/.ollama/id_ed25519'. Generating new private key.
Your new public key is:

ssh-ed25519 AAAAC3NzaC1lZDI1NTE5AAAAID1CarJRHzt3rUEznD+PjhcdWpLH/c1vPmz+faMhGNjw

2024/11/21 09:16:29 routes.go:1189: INFO server config env="map[CUDA_VISIBLE_DEVICES: GPU_DEVICE_ORDINAL: HIP_VISIBLE_DEVI
CES: HSA_OVERRIDE_GFX_VERSION: HTTPS_PROXY: HTTP_PROXY: NO_PROXY: OLLAMA_DEBUG:false OLLAMA_FLASH_ATTENTION:false OLLAMA_G
PU_OVERHEAD:0 OLLAMA_HOST:http://127.0.0.1:11434 OLLAMA_INTEL_GPU:false OLLAMA_KEEP_ALIVE:5m0s OLLAMA_LLM_LIBRARY: OLLAMA_
LOAD_TIMEOUT:5m0s OLLAMA_MAX_LOADED_MODELS:0 OLLAMA_MAX_QUEUE:512 OLLAMA_MODELS:/root/.ollama/models OLLAMA_MULTIUSER_CACH
E:false OLLAMA_NOHISTORY:false OLLAMA_NOPRUNE:false OLLAMA_NUM_PARALLEL:0 OLLAMA_ORIGINS:[http://localhost https://localho
```

Install Ollama on Colab

- Then execute this block again to download the LLM:

```
/content# ollama pull llama3.2:1b
pulling manifest
pulling 74701a8c35f6... 100%                                                  1.3 GB
pulling 966de95ca8a6... 100%                                                  1.4 KB
pulling fcc5a6bec9da... 100%                                                  7.7 KB
pulling a70ff7e570d9... 100%                                                  6.0 KB
pulling 4f659a1e86d7... 100%                                                   485 B
verifying sha256 digest
writing manifest
success
```

- You can find other available LLM on this website: https://ollama.com/library

Ollama Testing

- After all, you can try that the LLM to response your question or chat with it.

```
[7]  # Setting up the model that this tutorial will use
     MODEL = "llama3.2:1b" # https://ollama.com/library/llama3.2:3b
     EMBED_MODEL = "jinaai/jina-embeddings-v2-base-en"
```

```
[22] # Initialize an instance of the Ollama model
     llm = Ollama(model=MODEL)
     # Invoke the model to generate responses
     response = llm.invoke("What is the capital of Taiwan?")
     print(response)
```

```
The capital of Taiwan is Taipei.
```

Build a simple RAG system by using LangChain

- Initialize the LLM_model and Embedding_model.
  - For the details of model's setting (model_kwargs, encode_kwargs), you can check through this website.

```
# Initialize the Llama 3 model
llm_model = Ollama(model=MODEL)

# Create an embedding model
model_kwargs = {'trust_remote_code': True}
encode_kwargs = {'normalize_embeddings': False}
embeddings_model = HuggingFaceEmbeddings(
        model_name=EMBED_MODEL,
        model_kwargs=model_kwargs,
        encode_kwargs=encode_kwargs
)
```

Cosine similarity inherently normalizes vectors during computation, making pre-normalization unnecessary and potentially redundant.

Build a simple RAG system by using LangChain

- ChatPromptTemplate is a library to create a prompt template for chat models.
- In ChatModel, the prompt architecture will be like:
  - system: {guide, character setting}
  - human: {input}
  - assistant: {model output}
  - Human: {input} …

```
# Prompt setting
system_prompt = (
        "Use the given context to answer the question. "
        "If you don't know the answer, say you don't know. "
        "Use three sentence maximum and keep the answer concise. "
        "Context: {context}"
)
prompt = ChatPromptTemplate.from_messages(
        [
                ("system", system_prompt),
                ("human", "{input}"),
        ]
)
print(prompt)
```

Vector stores

- Vector store algorithms power RAG by enabling fast, scalable similarity searches in high-dimensional spaces.

1. Chroma
   - **Vector Database**: Focused on easy integration for machine learning workflows.
   - Supports both vector search and metadata filtering (e.g., based on tags).
   - Designed to work seamlessly with LLMs and other AI systems.
2. FAISS
   - **Library for Similarity Search**: Built by Meta for efficient similarity search in large-scale datasets.
   - Uses **inverted file index (IVF)** or **HNSW (graph-based)** algorithms for fast **approximate nearest neighbor (ANN)** search.
   - Optimized for scalability with large datasets and GPU support.

Vector stores

- Create a Chroma vector store, then use .as_retriever() to transform the vector store into a retriever.
  - fetch_k: Amount of documents to pass to MMR (Maximal Marginal Relevance) algorithm.
  - k: Amount of documents to return.

```
# Prepare documents
documents = [
        Document(page_content="The capital of Florida is Tallahassee.", metadata={"id": 0}),
        Document(page_content="Florida is known for its beautiful beaches and warm climate.", metadata={"id": 1}),
        Document(page_content="The largest city in Florida by population is Jacksonville.", metadata={"id": 2}),
        Document(page_content="The President of Miami Dade College is President Madeline Pumariega.", metadata={"id": 3}),
        Document(page_content="The Provost of Miami Dade College is Dr. Malou C. Harrison.", metadata={"id": 4}),
        Document(page_content="Dr. Ernesto Lee is an AI and Data Analytics Professor on the Kendall Campus at Miami Dade College.", met
]
```

```
# Create Chroma vector store
# search_type could be "similarity" (default), "mmr", or "similarity_score_threshold"
vector_store = Chroma.from_documents(documents, embedding=embeddings_model)
retriever = vector_store.as_retriever(search_type="mmr", search_kwargs={"k": 3, "fetch_k": 5})
```

MRR ANN

Load the QA chain

- Chains refer to sequences of calls - whether to an LLM, a tool, or a data preprocessing step.
- **LCEL** (LangChain Expression Language) was designed to support deploying prototypes into production without any code changes, from simple "prompt + LLM" chains to complex ones.

```
# Load the QA chain
question_answer_chain = create_stuff_documents_chain(llm_model, prompt) # Create a chain for passing a list of Documents to a model.
# print(question_answer_chain)

chain = create_retrieval_chain(retriever, question_answer_chain) # Create retrieval chain that retrieves documents and then passes them on.
```

chain

```
bound=RunnableAssign(mapper={
  context: RunnableBinding(bound=RunnableLambda(lambda x: x['input'])
           | VectorStoreRetriever(tags=['Chroma', 'HuggingFaceEmbeddings'], vectorstore=<langchain_community.vectorstores.chroma.Chroma object at 0x7c15d044f730>,
search_type='mmr', search_kwargs={'k': 3, 'fetch_k': 5}), kwargs={}, config={'run_name': 'retrieve_documents'}, config_factories=[])
})
| RunnableAssign(mapper={
    answer: RunnableBinding(bound=RunnableBinding(bound=RunnableAssign(mapper={
              context: RunnableLambda(format_docs)
            }), kwargs={}, config={'run_name': 'format_inputs'}, config_factories=[])
            | ChatPromptTemplate(input_variables=['context', 'input'], input_types={}, partial_variables={},
messages=[SystemMessagePromptTemplate(prompt=PromptTemplate(input_variables=['context'], input_types={}, partial_variables={}, template="Use the given context to
answer the question. If you don't know the answer, say you don't know. Use three sentence maximum and keep the answer concise. Context: {context}"), additional_kwargs={}),
HumanMessagePromptTemplate(prompt=PromptTemplate(input_variables=['input'], input_types={}, partial_variables={}, template='{input}'), additional_kwargs={})])
            | Ollama(model='llama3.2:1b')
            | StrOutputParser(), kwargs={}, config={'run_name': 'stuff_documents_chain'}, config_factories=[])
  }) kwargs={} config={'run_name': 'retrieval_chain'} config_factories=[]
```

QA with RAG system

- Test QA chain with some sample questions by calling chain.invoke().
  - We only need to give input to the chain.

```
# Use the QA chain to retrieve relevant documents and generate a response
queries = [
        "What is the capital of Florida?",
        "Who is the President of Miami Dade College?",
        "Who is the Provost of Miami Dade College?",
        "Who is Dr. Ernesto Lee?"
]

for query in queries:
        response = chain.invoke({"input": query})
        print(f"Query: {query}\nResponse: {response}\n")
```