```
%pip install --upgrade --quiet  langchain langchain-community
langchain-openai langchain-experimental neo4j tiktoken
yfiles_jupyter_graphs

from langchain_core.runnables import (
    RunnableBranch,
    RunnableLambda,
    RunnableParallel,
    RunnablePassthrough,
)

from langchain_core.prompts import ChatPromptTemplate
from langchain_core.prompts.prompt import PromptTemplate


!huggingface-cli login

from langchain_community.llms import HuggingFacePipeline
from transformers import AutoTokenizer, AutoModelForCausalLM, pipeline

# Model name
model_name = "meta-llama/Llama-3.2-1B"

# Load tokenizer and model
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name,
device_map="auto")

# Create text-generation pipeline
pipe = pipeline("text-generation", model=model, tokenizer=tokenizer,
max_new_tokens=512, temperature=0.7, **{'max_length': 1024}) #
Increased max_length
# Integrate with LangChain
llm = HuggingFacePipeline(pipeline=pipe)

import os

# Create the 'data' directory if it doesn't exist
data_dir = "data"
if not os.path.exists(data_dir):
    os.makedirs(data_dir)
    print(f"Directory '{data_dir}' created successfully.")
else:
    print(f"Directory '{data_dir}' already exists.")

!pip install langchain pypdf2

from PyPDF2 import PdfReader
from langchain.docstore.document import Document
from langchain.text_splitter import TokenTextSplitter
import os
```

```
# import os

# Path to the PDF directory in Colab
pdf_directory = "/content/data/"

# Get all PDF file paths
pdf_files = [os.path.join(pdf_directory, f) for f in
os.listdir(pdf_directory) if f.endswith(".pdf")]

if not pdf_files:
    raise FileNotFoundError("No PDF files found in the /content/data/
directory.")

# Now, pdf_files contains a list of all PDF file paths
print("Found PDF files:", pdf_files)


# Read PDF and extract text
def extract_text_from_pdf(pdf_path):
    # Iterate through each PDF file in the list
    all_text = ""
    for file_path in pdf_path:
        reader = PdfReader(file_path) # Pass the file path to
PdfReader
        text = "\n".join([page.extract_text() for page in reader.pages
if page.extract_text()])
        all_text += text  # Combine text from all PDFs

    return all_text

# Store the extracted text
raw_text = extract_text_from_pdf(pdf_files)

# Convert raw text into a list of Document objects
raw_documents = [Document(page_content=raw_text)]

# Initialize text splitter
text_splitter = TokenTextSplitter(chunk_size=200, chunk_overlap=24)

# Split the document into chunks
documents = text_splitter.split_documents(raw_documents)

# Print some chunked results
print(f"Total chunks created: {len(documents)}")

from typing import Tuple, List, Optional

from langchain_core.messages import AIMessage, HumanMessage
from langchain_core.output_parsers import StrOutputParser
```

```
from langchain_core.runnables import ConfigurableField

from yfiles_jupyter_graphs import GraphWidget
from neo4j import GraphDatabase

import os

try:
  import google.colab
  from google.colab import output
  output.enable_custom_widget_manager()
except:
  pass

from langchain_community.vectorstores import Neo4jVector

NEO4J_URI="neo4j+s://18163ba0.databases.neo4j.io"
NEO4J_USERNAME="neo4j"
NEO4J_PASSWORD="8PrzdwumxUpnxgfbfCxolKVt8Wj5ti5qqhaIX4VakWA"

# os.environ["OPENAI_API_KEY"] = OPENAI_API_KEY
os.environ["NEO4J_URI"] = NEO4J_URI
os.environ["NEO4J_USERNAME"] = NEO4J_USERNAME
os.environ["NEO4J_PASSWORD"] = NEO4J_PASSWORD

from langchain_community.graphs import Neo4jGraph

graph = Neo4jGraph()

!pip install json-repair

from langchain_experimental.graph_transformers import
LLMGraphTransformer
llm_transformer = LLMGraphTransformer(llm=llm)

from langchain_experimental.graph_transformers import
LLMGraphTransformer
llm_transformer = LLMGraphTransformer(llm=llm)

# Override the default config if needed
config = {"temperature": 0.7}  # Or any other positive float value

graph_documents =
llm_transformer.convert_to_graph_documents(documents[:20],
config=config)

graph_documents

graph.add_graph_documents(
    graph_documents,
    baseEntityLabel=True,
```

```
    include_source=True
)

# directly show the graph resulting from the given Cypher query
default_cypher = "MATCH (s)-[r:!MENTIONS]->(t) RETURN s,r,t LIMIT 50"

from yfiles_jupyter_graphs import GraphWidget
from neo4j import GraphDatabase

try:
  import google.colab
  from google.colab import output
  output.enable_custom_widget_manager()
except:
  pass

def showGraph(cypher: str = default_cypher):
    # create a neo4j session to run queries
    driver = GraphDatabase.driver(
        uri = os.environ["NEO4J_URI"],
        auth = (os.environ["NEO4J_USERNAME"],
                os.environ["NEO4J_PASSWORD"]))
    session = driver.session()
    widget = GraphWidget(graph = session.run(cypher).graph())
    widget.node_label_mapping = 'id'
    display(widget)
    return widget

showGraph()

from typing import Tuple, List, Optional

from langchain_community.vectorstores import Neo4jVector

from langchain_community.embeddings import HuggingFaceEmbeddings
from langchain_community.vectorstores.neo4j_vector import Neo4jVector

# Initialize Hugging Face embeddings (Llama model)
hf_embeddings = HuggingFaceEmbeddings(model_name="sentence-
transformers/all-MiniLM-L6-v2")

# Create a vector index from an existing Neo4j graph
vector_index = Neo4jVector.from_existing_graph(
    hf_embeddings,
    search_type="hybrid",
    node_label="Document",
    text_node_properties=["text"],
    embedding_node_property="embedding"
)
```

```
graph.query("CREATE FULLTEXT INDEX entity IF NOT EXISTS FOR
(e:__Entity__) ON EACH [e.id]")

from langchain_core.pydantic_v1 import BaseModel, Field
# Extract entities from text
class Entities(BaseModel):
    """Identifying information about entities."""

    names: List[str] = Field(
        ...,
        description="All the person, organization, or business
entities that "
        "appear in the text",
    )

from langchain_core.prompts import ChatPromptTemplate
from langchain_core.prompts.prompt import PromptTemplate

prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            "You are extracting emotional states, feelings of the
peoples and individuals and their reasons plus their solutions from
the text.",
        ),
        (
            "human",
            "Use the given format to extract information from the
following "
            "input: {question}",
        ),
    ]
)

from langchain.chains import LLMChain
from langchain.output_parsers import PydanticOutputParser

output_parser = PydanticOutputParser(pydantic_object=Entities)

# Create an LLMChain for entity extraction
entity_chain = LLMChain(
    llm=llm,
    prompt=prompt,
    output_parser=output_parser,
)

entity_chain = prompt | llm

response = entity_chain.invoke({"question": "how can one feel self
contained?"})
```

```
try:
    structured_response = response  # Process output based on the
Llama model's response format
    print(structured_response)
except Exception as e:
    print("Error parsing response:", e)
    print("Raw Output:", response)

from langchain_community.vectorstores.neo4j_vector import
remove_lucene_chars

def generate_full_text_query(input: str) -> str:
    full_text_query = ""
    words = [el for el in remove_lucene_chars(input).split() if el]
    for word in words[:-1]:
        full_text_query += f" {word}~2 AND"
    full_text_query += f" {words[-1]}~2"
    return full_text_query.strip()

# Fulltext index query
def structured_retriever(question: str) -> str:
    result = ""
    entities = entity_chain.invoke({"question": question})
    for entity in entities.names:
        response = graph.query(
            """CALL db.index.fulltext.queryNodes('entity', $query,
{limit:2})
            YIELD node,score
            CALL {
              WITH node
              MATCH (node)-[r:!MENTIONS]->(neighbor)
              RETURN node.id + ' - ' + type(r) + ' -> ' + neighbor.id
AS output
              UNION ALL
              WITH node
              MATCH (node)<-[r:!MENTIONS]-(neighbor)
              RETURN neighbor.id + ' - ' + type(r) + ' -> ' +  node.id
AS output
            }
            RETURN output LIMIT 50
            """,
            {"query": generate_full_text_query(entity)},
        )
        result += "\n".join([el['output'] for el in response])
    return result

def structured_retriever(question: str) -> str:
    result = ""
    entities = entity_chain.invoke({"question": question})
```

```
    # Debugging: Print the response to see what it returns
    print("Raw response from entity_chain:", entities)

    # Ensure entities is a list
    if isinstance(entities, str):  # If it's a string, wrap it in a
list
        entities = [entities]
    elif isinstance(entities, dict) and "names" in entities:  # If
it's a dict, extract 'names'
        entities = entities["names"]

    for entity in entities:
        response = graph.query(
            """CALL db.index.fulltext.queryNodes('entity', $query,
{limit:2})
            YIELD node,score
            CALL {
              WITH node
              MATCH (node)-[r:!MENTIONS]->(neighbor)
              RETURN node.id + ' - ' + type(r) + ' -> ' + neighbor.id
AS output
              UNION ALL
              WITH node
              MATCH (node)<-[r:!MENTIONS]-(neighbor)
              RETURN neighbor.id + ' - ' + type(r) + ' -> ' +  node.id
AS output
            }
            RETURN output LIMIT 50
            """,
            {"query": generate_full_text_query(entity)},
        )
        result += "\n".join([el['output'] for el in response])

    return result

print(structured_retriever("What is happiness?"))

def structured_retriever(question: str) -> str:
    result = ""
    entities = entity_chain.invoke({"question": question})

    # Debugging: Print the response to see what it returns
    print("Raw response from entity_chain:", entities)

    # Ensure entities is a list
    if isinstance(entities, str):  # If it's a string, wrap it in a
list
        entities = [entities]
    elif isinstance(entities, dict) and "names" in entities:  # If
```

```
it's a dict, extract 'names'
        entities = entities["names"]

    for entity in entities:
        response = graph.query(
            """CALL db.index.fulltext.queryNodes('entity', $query,
{limit:2})
            YIELD node,score
            CALL {
              WITH node
              MATCH (node)-[r:!MENTIONS]->(neighbor)
              RETURN node.id + ' - ' + type(r) + ' -> ' + neighbor.id
AS output
              UNION ALL
              WITH node
              MATCH (node)<-[r:!MENTIONS]-(neighbor)
              RETURN neighbor.id + ' - ' + type(r) + ' -> ' +  node.id
AS output
            }
            RETURN output LIMIT 50
            """,
            {"query": generate_full_text_query(entity)},
        )
        result += "\n".join([el['output'] for el in response])

    return result

print(structured_retriever("Who is happiness?"))

def retriever(question: str):
    print(f"Search query: {question}")
    structured_data = structured_retriever(question)
    unstructured_data = [el.page_content for el in
vector_index.similarity_search(question)]
    final_data = f"""Structured data:
{structured_data}
Unstructured data:
{"#Document ". join(unstructured_data)}
    """
    return final_data

_template = """Given the following conversation and a follow up
question, rephrase the follow up question to be a standalone question,
in its original language.
Chat History:
{chat_history}
Follow Up Input: {question}
Standalone question:"""

CONDENSE_QUESTION_PROMPT = PromptTemplate.from_template(_template)
```

```
def _format_chat_history(chat_history: List[Tuple[str, str]]) -> List:
    buffer = []
    for human, ai in chat_history:
        buffer.append(HumanMessage(content=human))
        buffer.append(AIMessage(content=ai))
    return buffer

_search_query = RunnableBranch(
    # If input includes chat_history, we condense it with the follow-
up question
    (
        RunnableLambda(lambda x:
bool(x.get("chat_history"))).with_config(
            run_name="HasChatHistoryCheck"
        ),  # Condense follow-up question and chat into a
standalone_question
        RunnablePassthrough.assign(
            chat_history=lambda x:
_format_chat_history(x["chat_history"])
        )
        | CONDENSE_QUESTION_PROMPT
        | llm.bind(temperature=0.7) # Use bind to set the temperature
parameter
        | StrOutputParser(),
    ),
    # Else, we have no chat history, so just pass through the question
    RunnableLambda(lambda x : x["question"]),
)

template = """Answer the question based only on the following context,
and try to be as empathatic as possible and if you don't know any
context just calmly say this is not my expertise but i will gain once
i have enough computing resources to be trained on:
{context}

Question: {question}
Use natural language and be concise.
Answer:"""

prompt = ChatPromptTemplate.from_template(template)

chain = (
    RunnableParallel(
        {
            "context": _search_query | retriever,
            "question": RunnablePassthrough(),
        }
    )
    | prompt
    | llm
```

```
    | StrOutputParser()
)

chain.invoke({"question":"why is sadness happen?"})

chain.invoke(
    {
        "question": "is it a bad thing?",
        "chat_history": [("why one is sad","Sadness helps us ‘to
adjust") ],
    }
)

!pip install gradio

import gradio as gr

def chatbot_response(message, history):
    try:
        # Ensure history is in the correct format
        if not isinstance(history, list):
            history = []

        # Convert chat history to the format Gradio expects
        formatted_history = []
        for human, ai in history:
            formatted_history.append([human, ai])

        response = chain.invoke({
            "question": message,
            "chat_history": formatted_history
        })

        # Return both the response and the updated history
        return response, formatted_history + [[message, response]]

    except Exception as e:
        print(f"An error occurred: {e}")
        return "I'm sorry, I'm having trouble processing your
request.", history

def set_example_prompt(prompt):
    return prompt  # Auto-fills the text input when clicked

with gr.Blocks(css="""
    body {background-color: #f0f4f8; font-family: 'Arial', sans-
serif;}
    .gradio-container {max-width: 750px; margin: auto; text-align:
center;}
    #title {color: #2c3e50; font-size: 28px; font-weight: bold;}
    #subtitle {color: #34495e; font-size: 18px; margin-bottom: 15px;}
```

```
    .chatbot {background: #ffffff; border-radius: 12px; box-shadow:
0px 4px 10px rgba(0, 0, 0, 0.1);}
    .textbox {border-radius: 8px; border: 2px solid #3498db; padding:
10px; font-size: 16px;}
    .send-button {background: #3498db; color: white; font-weight:
bold; border-radius: 8px; padding: 10px 20px;}
    .send-button:hover {background: #2980b9;}
    .example-btn {background: #ecf0f1; color: #2c3e50; border-radius:
8px; padding: 8px 12px; font-size: 14px; margin: 4px;}
    .example-btn:hover {background: #bdc3c7;}
""") as demo:
    gr.Markdown(
        """
        # **Emotional Support AI Assistant**
        ## □ Your AI Companion for Comfort & Motivation
        *Share your thoughts, and I'll be here to support you.*
        """,
        elem_id="title"
    )

    chatbot = gr.Chatbot(height=400, bubble_full_width=False,
elem_classes="chatbot")
    user_input = gr.Textbox(placeholder="Type your thoughts here...",
show_label=False, elem_classes="textbox")
    submit_button = gr.Button("□ Send", variant="primary",
elem_classes="send-button")

    gr.Markdown("### □ Need Help? Try These:", elem_id="subtitle")

    with gr.Row():
        example1 = gr.Button("I'm feeling overwhelmed, can you help?",
elem_classes="example-btn")
        example2 = gr.Button("How do I manage stress effectively?",
elem_classes="example-btn")
        example3 = gr.Button("I need some motivation to get through
the day.", elem_classes="example-btn")
        example4 = gr.Button("Can you suggest ways to improve mental
well-being?", elem_classes="example-btn")
        example5 = gr.Button("Tell me something positive.",
elem_classes="example-btn")

    # Event handlers for example buttons
    for example in [example1, example2, example3, example4, example5]:
        example.click(
            set_example_prompt,
            inputs=[],
            outputs=[user_input]
        )

    # Main chat submission
```

```
    submit_button.click(
    chatbot_response,
    inputs=[user_input, chatbot],
    outputs=[chatbot, chatbot]
)
# Launch the Gradio app
demo.launch(debug=False)
```

```
c:\Users\hp\AppData\Local\Programs\Python\Python311\Lib\site-packages\
tqdm\auto.py:21: TqdmWarning: IProgress not found. Please update
jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
c:\Users\hp\AppData\Local\Programs\Python\Python311\Lib\site-packages\
gradio\utils.py:924: UserWarning: Expected 1 arguments for function
<function set_example_prompt at 0x000001634E8AAFC0>, received 0.
  warnings.warn(
c:\Users\hp\AppData\Local\Programs\Python\Python311\Lib\site-packages\
gradio\utils.py:928: UserWarning: Expected at least 1 arguments for
function <function set_example_prompt at 0x000001634E8AAFC0>, received
0.
  warnings.warn(

Running on local URL:  http://127.0.0.1:7860

To create a public link, set `share=True` in `launch()`.

<IPython.core.display.HTML object>



IMPORTANT: You are using gradio version 4.26.0, however version 4.44.1
is available, please upgrade.
--------
```