```
!pip install llama-index llama-index-llms-groq groq llama-index-
embeddings-huggingface ipywidgets docx2txt torch transformers python-
pptx Pillow neo4j langchain-experimental -qq
```

```
  Preparing metadata (setup.py) ...
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 108.9/108.9 kB 3.9 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 472.8/472.8 kB 14.7 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 302.0/302.0 kB 17.7 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 209.0/209.0 kB 13.6 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 2.4/2.4 MB 50.2 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 58.4 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.2/1.2 MB 54.1 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 159.9/159.9 kB 10.4 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.6/1.6 MB 59.4 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 189.0/189.0 kB 12.4 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 298.0/298.0 kB 20.7 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.1/3.1 MB 82.2 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.2/1.2 MB 44.0 MB/s eta
0:00:00
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 49.5/49.5 kB 3.3 MB/s eta
0:00:00
```

```
from IPython.display import display
import ipywidgets as widgets
from llama_index.core import (
    VectorStoreIndex,
    SimpleDirectoryReader,
    StorageContext,
    load_index_from_storage
)
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core.node_parser import SentenceSplitter
from llama_index.llms.groq import Groq
from llama_index.core import Settings # import Settings from
llama_index.core
settings = Settings # import settings
import warnings
import os
```

```
from IPython.display import display
import ipywidgets as widgets
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader,
StorageContext, ServiceContext, load_index_from_storage
from llama_index.embeddings.huggingface import HuggingFaceEmbedding
from llama_index.core.node_parser import SentenceSplitter
from llama_index.llms.groq import Groq
import warnings
import os
from neo4j import GraphDatabase
import spacy

warnings.filterwarnings('ignore')
```

###Without Knowledge Graph

Create a folder named 'input' in '/content' and upload pdf in it

```
os.makedirs('input', exist_ok=True)

warnings.filterwarnings('ignore')

# Set the API key as an environment variable
os.environ["GROQ_API_KEY"] = ""

# Now you can access it in your code using os.getenv("GROQ_API_KEY")
GROQ_API_KEY = os.getenv("")

# Define your prompt template
prompt_template = """
Use the following pieces of information to answer the user's question.
If you don't know the answer, just say that you don't know, don't try
to make up an answer.

Context: {context}
Question: {question}

Answer the question and provide additional helpful information,
based on the pieces of information, if applicable. Be succinct.

Responses should be properly formatted to be easily read.
"""

# Define the context for your prompt
context = "This directory contains multiple documents providing
examples and solutions for various programming tasks."

# Data ingestion: load all files from a directory
directory_path = "/content/input"  # Update this with your directory
```

```
path
reader = SimpleDirectoryReader(input_dir=directory_path)
documents = reader.load_data()

# Split the documents into nodes
text_splitter = SentenceSplitter(chunk_size=1024, chunk_overlap=200)
nodes = text_splitter.get_nodes_from_documents(documents,
show_progress=True)

# Set up embedding model and LLM
embed_model =
HuggingFaceEmbedding(model_name="sentence-transformers/all-MiniLM-L6-
v2")
llm = Groq(model="llama3-70b-8192", api_key=GROQ_API_KEY)

# Configure settings for LlamaIndex
settings.llm = llm
settings.embed_model = embed_model
# Create and persist the vector store index
vector_index = VectorStoreIndex.from_documents(documents,
show_progress=True, node_parser=nodes)
vector_index.storage_context.persist(persist_dir="./storage_mini")

# Load the index from storage
storage_context =
StorageContext.from_defaults(persist_dir="./storage_mini")
index = load_index_from_storage(storage_context) # remove
service_context
```


```
{"model_id":"53bca90fb0e547ce9fab9666936e90a3","version_major":2,"vers
ion_minor":0}

{"model_id":"ae1ae4a23646477c8c15ea5b77836ae8","version_major":2,"vers
ion_minor":0}

{"model_id":"0de736ef81334ad08ebf5a476484ab2b","version_major":2,"vers
ion_minor":0}

{"model_id":"57271690ff26446a87b81b482534b56f","version_major":2,"vers
ion_minor":0}

{"model_id":"4573872a09cf4486b552d8b3fafb18e6","version_major":2,"vers
ion_minor":0}

{"model_id":"1ca53bf1c51b48d697e319f1d59f3ab0","version_major":2,"vers
ion_minor":0}

{"model_id":"3432215b6eaa4763a175618c22fbcfeb","version_major":2,"vers
ion_minor":0}
```

{"model_id":"1209f20e417347439c4b6336cb125116","version_major":2,"version_minor":0}

{"model_id":"a73c0ecce7c449b7880f3004a8731ca3","version_major":2,"version_minor":0}

{"model_id":"6696dc7cc54e424e87c92cab7f3e7884","version_major":2,"version_minor":0}

{"model_id":"ee084ed276d24f4681a04c44bc6fad23","version_major":2,"version_minor":0}

{"model_id":"4ac38bf6af894241bbe2d970610d0860","version_major":2,"version_minor":0}

{"model_id":"10677bac36eb411da8a6f202edac2005","version_major":2,"version_minor":0}

{"model_id":"aceb14573eda4c5dbf99906fdf32d325","version_major":2,"version_minor":0}

```
# Create the interactive widgets
input_box = widgets.Text(
    value='explain ?',
    placeholder='Type your question here',
    description='Question:',
    disabled=False
)

output_area = widgets.Output()

def on_button_click(b):
    with output_area:
        output_area.clear_output()
        question = input_box.value
        query_prompt = prompt_template.format(context=context,
question=question)
        resp = query_engine.query(query_prompt)
        print(resp.response)

button = widgets.Button(
    description='Ask',
    disabled=False,
    button_style='',
    tooltip='Ask the question',
    icon='check'
)

button.on_click(on_button_click)

display(input_box, button, output_area)
```

```
# Set up query engine
query_engine = index.as_query_engine() # remove service_context
```


```
{"model_id":"18e7c7f46c7a4cc7af18dffb90a32159","version_major":2,"vers
ion_minor":0}

{"model_id":"3e88ed4174294e669b07b5e780b8a145","version_major":2,"vers
ion_minor":0}

{"model_id":"872b02334b394e268b854699192c96ba","version_major":2,"vers
ion_minor":0}
```


With Knowledge Graph

change the neo4j aura instance if it gets deleted

replace the old url and pw with the new one

```
# ---- NEO4J SETUP ----
neo4j_uri = "neo4j+s://2e22c7c9.databases.neo4j.io"
neo4j_user = "neo4j"
neo4j_password = ""
driver = GraphDatabase.driver(neo4j_uri, auth=(neo4j_user,
neo4j_password))

# ---- ENVIRONMENT VARIABLES ----
os.environ["GROQ_API_KEY"] = ""
GROQ_API_KEY = os.getenv("")

# ---- PROMPT TEMPLATE ----
prompt_template = prompt_template = """
Use the following pieces of information to answer the user's question.
If you don't know the answer, just say that you don't know, don't try
to make up an answer.

Context: {context}
Graph Insights: {graph_insights}
Question: {question}

Answer the question and provide additional helpful information,
based on the pieces of information and graph insights, if applicable.
Be succinct.

Responses should be properly formatted to be easily read.
"""

#!pip install spacy[transformers] -q
#!python -m spacy download en_core_web_trf -q
```

```
# Define the context for your prompt
context = "This directory contains a variety of documents on multiple
topics, presented in different formats (e.g., text, PDF, HTML, JSON)."

# Data ingestion: load all files from a directory
directory_path = "/content/input"
reader = SimpleDirectoryReader(input_dir=directory_path)
documents = reader.load_data()

!python -m spacy download en_core_web_lg -q
```

```
━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 587.7/587.7 MB 876.4 kB/s eta
0:00:00
✔ Download and installation successful
You can now load the package via spacy.load('en_core_web_lg')
⚠ Restart to reload dependencies
If you are in a Jupyter or Colab notebook, you may need to restart
Python in
order to load all the package's dependencies. You can do this by
selecting the
'Restart kernel' or 'Restart runtime' option.
```

```
nlp = spacy.load("en_core_web_lg")
```

Function Overview: `populate_graph`

The `populate_graph` function processes a list of documents and populates a Neo4j graph database with concepts (entities) and their relationships. Here's how it works:

Steps

1. **Document Processing**:
   - For each document, the function extracts the text and uses an NLP model (likely spaCy or similar) to detect named entities, referred to as *concepts*.
2. **Node Creation**:
   - For each identified concept, a node labeled `Concept` is created in the Neo4j database if it doesn't already exist.
   - The `MERGE` operation in Neo4j ensures that duplicate nodes for the same concept are not created.
3. **Relationship Creation**:
   - The function creates a directed relationship (`RELATED_TO`) between consecutive concepts within the same document.
   - This links each concept to the next, capturing their sequential order within the document.

Summary

In essence, `populate_graph` extracts concepts from a series of documents and builds a graph in Neo4j, connecting these concepts through sequential relationships.

```
# Function to extract entities and relationships from documents
def populate_graph(documents, driver, nlp):
    doc_len = len(documents)
    index = 0
    with driver.session() as session:
        print(index,"/",doc_len)
        for doc in documents: # we are taking one doc at a time
            doc_text = doc.text
            nlp_doc = nlp(doc_text)
            #print(nlp_doc.ents)
            concepts = [ent.text for ent in nlp_doc.ents] # this gives
us a list of nodes(entities/concepts) for the current doc

            for concept in concepts: # adds each concept(node) to the
graph DB by ensuring no duplication
                session.run("MERGE (:Concept {name: $concept})",
concept=concept)

            # the below loop matches consecutive concepts with
'RELATED TO' relationship.
            # so if conceps=[a,b,c], then a->b, b->c
            #print(concepts)
            for i, concept in enumerate(concepts):
                if i + 1 < len(concepts):
                    next_concept = concepts[i + 1]
                    # print(concept," ",next_concept)
                    session.run(
                        """
                        MATCH (c1:Concept {name: $concept}),
(c2:Concept {name: $next_concept})
                        MERGE (c1)-[:RELATED_TO]->(c2)
                        """,
                        concept=concept, next_concept=next_concept
                    )

# Populate the Neo4j graph
populate_graph(documents, driver, nlp)
```

```
0 / 15
```

This code uses the `LlamaIndex` framework to create a vector store index from a collection of documents, enabling efficient retrieval of information.

1. **Document Splitting**:
   - The `SentenceSplitter` class is used to split the documents into nodes (text chunks) of 1024 characters with a 200-character overlap. This helps in structuring the documents for better indexing.
   - `get_nodes_from_documents()` is called to create nodes from the documents, displaying progress with `show_progress=True`.
2. **Setting Up Embedding and Language Models**:

- An embedding model (`HuggingFaceEmbedding`) and an LLM model (`Groq`) are initialized.
- `embed_model` uses `sentence-transformers/all-MiniLM-L6-v2` for embeddings.
- `llm` is set up with the `Groq` API for processing text.

3. **Service Context Configuration**:
   - `Settings` is used to configure the `service_context` with the `embed_model` and `llm`.
   - The `llm` and `embed_model` are assigned to `service_context`.
4. **Creating and Persisting the Vector Store Index**:
   - `VectorStoreIndex` is created from the documents, using the `service_context` and `nodes` for parsing.
   - The index is persisted to `./storage_mini` for later retrieval.
5. **Loading the Index from Storage**:
   - The index is reloaded using `StorageContext` with the default directory `./storage_mini`.
   - `load_index_from_storage` loads the persisted index, allowing for continued use without re-indexing the documents.

```
from llama_index.core import Settings

# Split the documents into nodes
text_splitter = SentenceSplitter(chunk_size=1024, chunk_overlap=200)
nodes = text_splitter.get_nodes_from_documents(documents,
show_progress=True)

# Set up embedding model and LLM
embed_model =
HuggingFaceEmbedding(model_name="sentence-transformers/all-MiniLM-L6-
v2")
llm = Groq(model="llama3-70b-8192", api_key=GROQ_API_KEY)

# Create service context
service_context = Settings  # (embed_model=embed_model, llm=llm)
service_context.llm = llm
service_context.embed_model = embed_model

# Create vector store index
vector_index = VectorStoreIndex.from_documents(documents,
show_progress=True, service_context=service_context,
node_parser=nodes)
vector_index.storage_context.persist(persist_dir="./storage_mini")

# Load the index from storage
storage_context =
StorageContext.from_defaults(persist_dir="./storage_mini")
index = load_index_from_storage(storage_context,
service_context=service_context)
```

{"model_id":"dce08698d726451eb0495fc7f1a503a6","version_major":2,"version_minor":0}

{"model_id":"7e1a4c71f68c4c1fa252eb545989986b","version_major":2,"version_minor":0}

{"model_id":"689bbd9c8838441c9db70bf290c171ad","version_major":2,"version_minor":0}

```
# Create the interactive widgets
input_box = widgets.Text(
    value='Explain Python?',
    placeholder='Type your question here',
    description='Question:',
    disabled=False
)

output_area = widgets.Output()
```

This function retrieves graph insights from a Neo4j graph database based on a user query. It searches for concepts related to the question and returns relevant insights.

1. **Neo4j Query Execution**:
   - A Neo4j session is started using `driver.session()`.
   - A Cypher query is executed to search for concepts (`Concept` nodes) whose names contain the question (case-insensitive search using `toLower()`).
   - The query also looks for related concepts using the `RELATED_TO` relationship. It optionally collects the related concepts connected to the original concept.
2. **Query Details**:
   - The query matches all `Concept` nodes and checks if their names contain the provided `question` string (ignoring case).
   - If the concept is found, it collects all related concepts via the `RELATED_TO` relationship.
   - The query returns the concept's name and its related concepts.
3. **Result Processing**:
   - The results from the query are looped through, and each record (concept and related concepts) is formatted into a string.
   - The function appends the formatted string to the `insights` list.
   - After processing all results, the function returns a joined string of insights or a message saying "No relevant graph insights found" if no results are found.

Key Points:

- **Purpose**: The function identifies concepts related to a user's question and provides a list of related concepts from the graph.
- **Output**: A formatted string of concepts and their related concepts, or a message indicating no relevant insights.

```
# Query Enhancement with Neo4j

def get_graph_insights(question):
    with driver.session() as session:
        result = session.run(
            """
            MATCH (c:Concept)
            WHERE toLower(c.name) CONTAINS toLower($question)
            OPTIONAL MATCH (c)-[r:RELATED_TO]->(other:Concept)
            RETURN c.name AS concept, collect(other.name) AS
related_concepts
            """,
            question=question
        )
        #print(result)
        insights = []
        for record in result:
            insights.append(f"Concept: {record['concept']}, Related
Concepts: {', '.join(record['related_concepts'])}")

        # Return after processing all results
        return "\n".join(insights)

prompt_template = """
Use the following pieces of information to answer the user's question.
If you don't know the answer, just say that you don't know, don't try
to make up an answer.

Context: {context}
Question: {question}

Answer the question and provide additional helpful information,
based on the pieces of information, if applicable. Be succinct.

Responses should be properly formatted to be easily read.
"""

context = "This directory contains multiple documents providing
examples and solutions for various programming tasks."

import ipywidgets as widgets
from IPython.display import display

output_area = widgets.Output()

question_input = widgets.Textarea(
    placeholder='Type your question here...',
    description='Question:',
    layout=widgets.Layout(width='400px', height='10')
)
```

```
ask_button = widgets.Button(
    description='Ask',
    disabled=False,
    button_style='',
    tooltip='Ask the question',
    icon='check'
)

def on_button_click(b):
    with output_area:
        output_area.clear_output()
        question = question_input.value

        graph_insights = get_graph_insights(question)
        query_prompt = prompt_template.format(context=context,
graph_insights=graph_insights, question=question)

        resp = query_engine.query(query_prompt)

        print(resp.response)

ask_button.on_click(on_button_click)

display(question_input, ask_button, output_area)

query_engine = index.as_query_engine(service_context=service_context)
```


```
{"model_id":"82883f40b1834b1e8c3fe9a533ced8cf","version_major":2,"vers
ion_minor":0}

{"model_id":"74ff9b0ce25242adba171a43267f9671","version_major":2,"vers
ion_minor":0}

{"model_id":"5e062cba975647588600fde1d88a75a2","version_major":2,"vers
ion_minor":0}
```