**Table of Contents**

1. [Overview](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#overview)
2. [Key Features](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#key-features)
3. [Technology Stack and Libraries](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#technology-stack-and-libraries)
4. [Before You Begin](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#before-you-begin)
5. [Code Explanation in Detail](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#code-explanation-in-detail)
   1. [Imports and Installations](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#imports-and-installations)
   2. [Setting Up Neo4j](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#setting-up-neo4j)
   3. [Environment Variables](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#environment-variables)
   4. [Prompt Template](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#prompt-template)
   5. [Loading Documents](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#loading-documents)
   6. [Populating the Graph with Spacy](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#populating-the-graph-with-spacy)
   7. [Splitting Documents into Nodes](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#splitting-documents-into-nodes)
   8. [Embedding and Language Model Setup](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#embedding-and-language-model-setup)
   9. [Vector Store Index Creation](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#vector-store-index-creation)
   10. [Index Loading from Storage](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#index-loading-from-storage)
   11. [Query Engine Creation](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#query-engine-creation)
   12. [Query Enhancement with Neo4j](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#query-enhancement-with-neo4j)
   13. [Feedback Mechanism Setup](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#feedback-mechanism-setup)
   14. [UI Widgets: Input and Output](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#ui-widgets-input-and-output)
   15. [Callbacks for Asking Questions and Submitting Feedback](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#callbacks-for-asking-questions-and-submitting-feedback)
   16. [Displaying the Interface](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#displaying-the-interface)
6. [How to Run the Code](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#how-to-run-the-code)
7. [How the Chatbot Works](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#how-the-chatbot-works)
8. [Extending and Customizing](https://chatgpt.com/c/677982bf-aa14-8009-963d-c513016aaba6?model=o1#extending-and-customizing)

**1. Overview**

This project demonstrates how to **integrate a Large Language Model (LLM) with a Neo4j knowledge graph** to create a chatbot that can:

* Store and retrieve insights (concepts, relationships) from a knowledge graph.
* Use a language model to generate answers in a human-readable form.
* Collect user feedback (ratings) to help improve or monitor performance over time.

The primary components are:

1. **Neo4j** for storing entities and relationships.
2. **Embeddings** to represent document text semantically.
3. **Groq** (or another LLM API) to generate coherent responses.
4. **ipywidgets** to build a simple user interface in Jupyter notebook.
5. **Feedback collection** stored in a CSV file.

**2. Key Features**

1. **Knowledge Graph Integration:** Extracts entities from text using Spacy and populates a Neo4j graph with nodes and relationships.
2. **Semantic Search with Vector Embeddings:** Converts text segments into numerical vectors for efficient retrieval.
3. **Language Model for Answer Generation:** Uses a large language model (Groq in this case) to provide context-aware responses.
4. **User-Friendly Interface:** Provides an interactive input box and an output area for real-time question answering.
5. **Feedback Collection:** After receiving an answer, the user can submit a rating (1–5). The question, answer, and rating are saved to a CSV file.

**3. Technology Stack and Libraries**

1. **Python 3**
2. **Jupyter Notebook / JupyterLab**
3. **Neo4j** (local or remote instance)
4. **ipywidgets** for the UI elements.
5. **Spacy** for NLP-based entity extraction.
6. **llama-index (LlamaIndex)** for building the vector store index and query engine.
7. **sentence-transformers** (HuggingFace) for creating text embeddings.
8. **Groq** (or a similar LLM) for generating responses.

**4. Before You Begin**

* **Install Requirements:** You need neo4j, langchain-experimental, spacy, llama-index, ipywidgets, sentence-transformers, and Groq (or another LLM API) installed in your environment.
* **Configure Neo4j:** Make sure Neo4j is up and running. Have your neo4j\_uri, username, and password ready to insert into the code.
* **Obtain API Key for the LLM:** If you’re using Groq, have your API key ready. If you use a different LLM, you’ll adapt that part of the code.

**5. Code Explanation in Detail**

Below is a structured walkthrough of the code cells in the Jupyter notebook. The code is presented in one single cell for convenience, but you can split it across multiple cells if desired.

**5.1 Imports and Installations**

!pip install neo4j langchain-experimental

!pip install spacy==3.5.2

!python -m spacy download en\_core\_web\_sm

* The **exclamation mark (!)** in Jupyter cells runs shell commands.
* We install **neo4j** for connecting to the Neo4j database, **langchain-experimental**, and **spacy**.
* We also download the **English Spacy model** (en\_core\_web\_sm) for entity extraction.

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

import csv

from neo4j import GraphDatabase

import spacy

* **ipywidgets**: Provides interactive elements in Jupyter (like text boxes, sliders, and buttons).
* **llama\_index**: Used for building vector indexes, retrieving documents, etc.
* **HuggingFaceEmbedding**: Generates embeddings for textual data using the HuggingFace model.
* **Groq**: A placeholder for a large language model.
* **warnings**, **os**, and **csv**: Standard Python libraries.
* **neo4j**: Official Neo4j driver.
* **spacy**: NLP library for entity extraction.

warnings.filterwarnings('ignore')

* Disables any non-critical warnings.

**5.2 Setting Up Neo4j**

neo4j\_uri = "ENTER URI"

neo4j\_user = "neo4j"

neo4j\_password = "ENTER PASSWORD"

driver = GraphDatabase.driver(neo4j\_uri, auth=(neo4j\_user, neo4j\_password))

* Replace ENTER URI and ENTER PASSWORD with the correct connection details for your Neo4j instance.
* The driver is responsible for session management to interact with Neo4j.

**5.3 Environment Variables**

os.environ["GROQ\_API\_KEY"] = "ENTER YOUR GROQ API KEY"

GROQ\_API\_KEY = os.getenv("GROQ\_API\_KEY")

* Sets and retrieves the **Groq API key** from environment variables. If you’re using a different LLM, this part would change accordingly.

**5.4 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.

"""

* This is the **prompt** that gets sent to the language model.
* Notice placeholders: {context}, {graph\_insights}, {question}.
* The model is instructed to remain concise and avoid making up answers if unsure.

**5.5 Loading Documents**

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

directory\_path = "ENTER YOUR DIRECTORY PATH"

reader = SimpleDirectoryReader(input\_dir=directory\_path)

documents = reader.load\_data()

* **context**: A short descriptive text about the documents.
* **directory\_path**: Points to the folder containing your text or PDF files.
* **SimpleDirectoryReader**: A utility from LlamaIndex that reads all documents from the specified directory.
* **documents**: A list of loaded data structures representing each file’s content.

**5.6 Populating the Graph with Spacy**

nlp = spacy.load("en\_core\_web\_sm")

def populate\_graph(documents, driver, nlp):

...

* Loads the **Spacy** English model.
* populate\_graph function:
  + For each document:
    - Processes the text with Spacy.
    - Extracts **entities** (e.g., ORG, PRODUCT) and treats them as **concepts** in the Neo4j graph.
    - Creates **:Concept** nodes in Neo4j for each entity.
    - Creates a **RELATED\_TO** relationship between consecutive concepts found in the text.

Finally, the function is called:

populate\_graph(documents, driver, nlp)

This step ensures that Neo4j has some data to reference for graph insights.

**5.7 Splitting Documents into Nodes**

text\_splitter = SentenceSplitter(chunk\_size=1024, chunk\_overlap=200)

nodes = text\_splitter.get\_nodes\_from\_documents(documents, show\_progress=True)

* **SentenceSplitter**: Takes the raw text and splits it into chunks (nodes) suitable for embedding.
  + chunk\_size=1024 means each chunk will have up to 1024 tokens (or characters, depending on implementation).
  + chunk\_overlap=200 means chunks may overlap to maintain context in consecutive chunks.

**5.8 Embedding and Language Model Setup**

embed\_model = HuggingFaceEmbedding(model\_name="sentence-transformers/all-MiniLM-L6-v2")

llm = Groq(model="llama3-70b-8192", api\_key=GROQ\_API\_KEY)

service\_context = ServiceContext.from\_defaults(embed\_model=embed\_model, llm=llm)

* **HuggingFaceEmbedding** uses the "sentence-transformers/all-MiniLM-L6-v2" model to generate embeddings (vector representations) of text.
* **Groq** is used as the large language model.
* **ServiceContext** is a utility from LlamaIndex that bundles the embedding model and the LLM, making it easier to reuse across different components.

**5.9 Vector Store Index Creation**

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")

1. **VectorStoreIndex.from\_documents**: Creates an index that allows you to query documents semantically.
2. **persist**: Saves the index to a local folder (./storage\_mini), so you don’t have to rebuild it every time you run the notebook.

**5.10 Index Loading from Storage**

storage\_context = StorageContext.from\_defaults(persist\_dir="./storage\_mini")

index = load\_index\_from\_storage(storage\_context, service\_context=service\_context)

* Loads the previously saved index from the ./storage\_mini directory.
* This is handy if you want to **resume** or **reuse** your index in future sessions without reprocessing all documents.

**5.11 Query Engine Creation**

query\_engine = index.as\_query\_engine(service\_context=service\_context)

* Transforms the index into a **query engine**, which can handle user queries.
* The service\_context ensures the same LLM and embedding model are used during query time.

**5.12 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

)

insights = []

...

return "\n".join(insights) if insights else "No relevant graph insights found."

* **get\_graph\_insights** function:
  + Uses the user’s question to find matching concepts in Neo4j whose names contain the question text (case-insensitive).
  + Retrieves related concepts (neighbors) via the RELATED\_TO relationship.
  + Creates a string summary of the findings (e.g., “Concept: XYZ, Related Concepts: ABC, DEF”).
* These results are inserted into the final prompt so that the LLM can use them when generating its answer.

**5.13 Feedback Mechanism Setup**

last\_question = None

last\_response = None

feedback\_csv\_path = "feedback\_log.csv"

if not os.path.isfile(feedback\_csv\_path):

with open(feedback\_csv\_path, "w", newline="") as f:

writer = csv.writer(f)

writer.writerow(["question", "response", "rating"])

* **Global Variables**:
  + last\_question and last\_response temporarily store the **current** Q&A pair.
  + feedback\_csv\_path points to the CSV file where feedback is recorded.
* **CSV Initialization**:
  + If feedback\_log.csv doesn’t exist, a new one is created with headers: [question, response, rating].

**5.14 UI Widgets: Input and Output**

input\_box = widgets.Text(

value='Explain Python?',

placeholder='Type your question here',

description='Question:',

disabled=False

)

output\_area = widgets.Output()

feedback\_output = widgets.Output()

* **Text Widget** (input\_box) for the user’s question.
* **Output Widget** (output\_area) where the answer is displayed.
* **feedback\_output** for printing feedback submission confirmations or errors.

rating\_slider = widgets.IntSlider(

value=3,

min=1,

max=5,

step=1,

description='Rating:',

style={'description\_width': 'initial'}

)

feedback\_button = widgets.Button(

description='Submit Feedback',

disabled=False,

button\_style='',

tooltip='Submit your feedback rating',

icon='thumbs-up'

)

* **Rating Slider** (rating\_slider) allows the user to choose a rating between 1 and 5.
* **Feedback Button** (feedback\_button) to submit the chosen rating.

**5.15 Callbacks for Asking Questions and Submitting Feedback**

def on\_button\_click(\_):

global last\_question, last\_response

with output\_area:

output\_area.clear\_output()

question = input\_box.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)

last\_question = question

last\_response = resp.response

* **on\_button\_click** function:
  + Clears the output area.
  + Retrieves the user’s question from input\_box.
  + Gets any relevant insights from Neo4j.
  + Formats the prompt using the template, context, graph insights, and question.
  + Calls the query\_engine.query() to get the LLM response.
  + Prints the response, and stores the question and response into last\_question and last\_response.

def on\_feedback\_click(\_):

global last\_question, last\_response

with feedback\_output:

feedback\_output.clear\_output()

if not last\_question or not last\_response:

print("No recent question/response to rate. Please ask a question first.")

return

rating\_value = rating\_slider.value

with open(feedback\_csv\_path, "a", newline="") as f:

writer = csv.writer(f)

writer.writerow([last\_question, last\_response, rating\_value])

print(f"Feedback recorded for question: '{last\_question}' with rating: {rating\_value}")

* **on\_feedback\_click** function:
  + Checks if there’s a recent Q&A pair. If none, it prompts the user to ask a question first.
  + Grabs the rating from the slider.
  + Appends a new row to feedback\_log.csv with [question, response, rating\_value].
  + Displays a confirmation message.

Finally, we attach these callbacks to the **Ask** button and **Submit Feedback** button:

ask\_button = widgets.Button(

description='Ask',

disabled=False,

button\_style='',

tooltip='Ask the question',

icon='check'

)

ask\_button.on\_click(on\_button\_click)

feedback\_button.on\_click(on\_feedback\_click)

**5.16 Displaying the Interface**

display(input\_box, ask\_button, output\_area)

display(rating\_slider, feedback\_button, feedback\_output)

* **display** is used in Jupyter notebooks to render widgets.
* Displays the text input, the **Ask** button, and the output area in one section.
* Displays the rating slider, **Submit Feedback** button, and the feedback output area in another section.

**6. How to Run the Code**

1. **Open Jupyter Notebook** (or JupyterLab) in your terminal by typing:
2. jupyter notebook

or

jupyter lab

1. **Create a New Notebook**: In the Jupyter interface, select “New → Python 3 (ipykernel)”.
2. **Paste the Code**: Copy and paste the entire code snippet into a single cell (or split it if you prefer).
3. **Replace Placeholder Values**:
   * neo4j\_uri: e.g., bolt://localhost:7687
   * neo4j\_password: the actual password.
   * directory\_path: the path to your documents folder.
   * os.environ["GROQ\_API\_KEY"]: your LLM API key.
4. **Run Each Cell**: Press Shift+Enter to execute. If everything is set up correctly, you’ll see the UI widgets appear.
5. **Ask a Question**: Enter a question in the text box and click **“Ask.”**
6. **View the Answer**: The response will appear in the output area beneath the buttons.
7. **Provide Feedback**: Adjust the rating slider (1–5), click **“Submit Feedback,”** and see a confirmation message. Feedback is saved to **feedback\_log.csv**.

**7. How the Chatbot Works**

1. **Data Ingestion**
   * Reads documents from a directory using SimpleDirectoryReader.
   * Splits the text into manageable chunks using SentenceSplitter.
2. **Knowledge Graph Population**
   * Spacy identifies entities (like ORG, PRODUCT) in the text.
   * Each entity becomes a **:Concept** node in Neo4j.
   * Consecutive entities in the text gain a **RELATED\_TO** relationship.
3. **Vector Embedding**
   * Each chunk of text is transformed into a vector using **HuggingFace embeddings**.
   * These embeddings enable **semantic retrieval**—the chatbot can find relevant chunks even if the user’s words differ from the document’s words.
4. **Query**
   * When a user asks a question, the chatbot:
     1. **Searches** the knowledge graph for relevant insights.
     2. **Builds** a prompt that includes the user’s question, the context, and any graph insights.
     3. **Calls** the language model to generate a context-aware answer.
5. **Feedback Collection**
   * After receiving the answer, the user can rate it on a scale of 1–5.
   * The code logs the question, the answer, and the rating in **feedback\_log.csv**.

**8. Extending and Customizing**

1. **Change Entity Types**
   * In the Spacy extraction, you might want to capture different entity labels (e.g., PERSON, GPE, DATE). Adjust the filtering logic accordingly.
2. **Refine Graph Relationship Logic**
   * Instead of linking only consecutive entities, you could devise more advanced rules to relate entities based on their context in the text.
3. **Use a Different LLM**
   * If you prefer OpenAI GPT, Azure OpenAI, or any other service, replace the Groq integration with the respective API calls.
4. **Front-End Options**
   * If you want a more user-friendly or public-facing front end, integrate the core logic into a web framework like **Flask**, **FastAPI**, or **Streamlit**.
5. **Feedback Analysis**
   * Currently, feedback is stored in a CSV. You can build a system that **retrains** or **fine-tunes** your model based on user ratings, or you can generate usage analytics.
6. **Security and Access Control**
   * In production, ensure you have proper authentication for both the chatbot UI and the Neo4j database.
   * Mask or encrypt sensitive credentials and API keys.