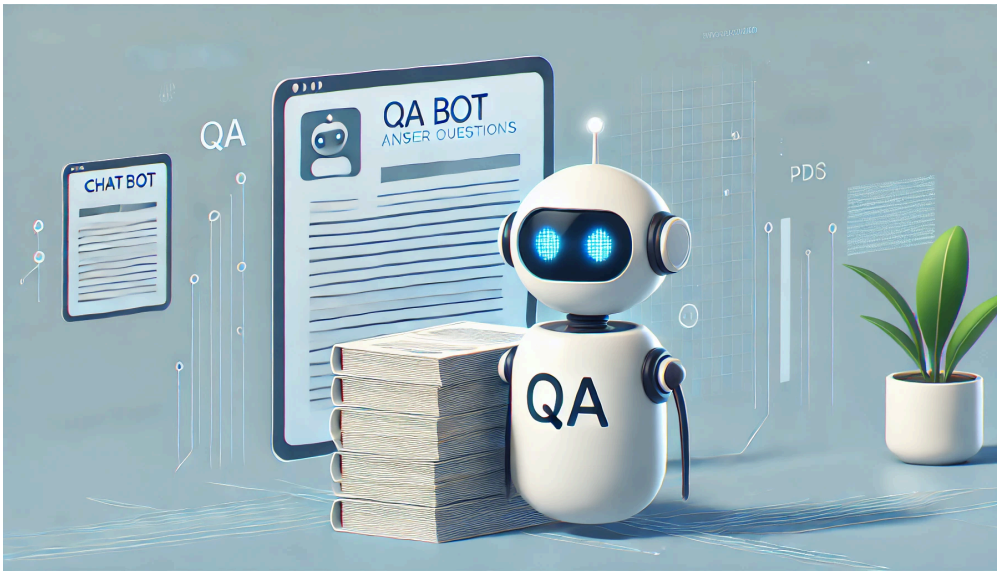
Construct a QA Bot that Leverages LangChain and LLMs to Answer Questions from Loaded Documents

Estimated time needed: 60 minutes

In this project, you will construct a question-answering (QA) bot. This bot will leverage LangChain and a large language model (LLM) to answer questions based on content from loaded PDF documents. To build a fully functional QA system, you'll combine various components, including document loaders, text splitters, embedding models, vector databases, retrievers, and Gradio as the front-end interface.

Imagine you're tasked with creating an intelligent assistant that can quickly and accurately respond to queries based on a company's extensive library of PDF documents. This could be anything from legal documents to technical manuals. Manually searching through these documents would be time-consuming and inefficient.



Source: DALL-E

In this project, you will construct a QA bot that automates this process. By leveraging LangChain and an LLM, the bot will read and understand the content of loaded PDF documents, enabling it to provide precise answers to user queries. You will integrate the tools and techniques, from document loading, text splitting, embedding, vector storage, and retrieval, to create a seamless and user-friendly experience via a Gradio interface.

Learning objectives

By the end of this project, you will be able to:

- Combine multiple components, such as document loaders, text splitters, embedding models, and vector databases, to construct a fully functional QA bot
- Leverage LangChain and LLMs to solve the problem of retrieving and answering questions based on content from large PDF documents

Set up

Setting up a virtual environment

Let's create a virtual environment. Using a virtual environment allows you to manage dependencies for different projects separately, avoiding conflicts between package versions.

In the terminal of your Cloud IDE, ensure that you are in the path `/home/project`, then run the following commands to create a Python virtual environment.

```
pip install virtualenv
virtualenv my_env # create a virtual environment named my_env
source my_env/bin/activate # activate my_env
```

Installing necessary libraries

To ensure seamless execution of your scripts, and considering that certain functions within these scripts rely on external libraries, it's essential to install some prerequisite libraries before you begin. For this project, the key libraries you'll need are `Gradio` for creating user-friendly web interfaces and `IBM-watsonx-AI` for leveraging advanced LLM models from the IBM watsonx API.

- **`gradio`** allows you to build interactive web applications quickly, making your AI models accessible to users with ease.
- **`ibm-watsonx-ai`** for using LLMs from IBM watsonx.ai.
- **`langchain`, `langchain-ibm`, `langchain-community`** for using relevant features from Langchain.
- **`chromadb`** for using the chroma database as a vector database.
- **`pypdf`** is required for loading PDF documents.

Here's how to install these packages (from your terminal):

```
# installing necessary packages in my_env
python3.11 -m pip install \
gradio==4.44.0 \
ibm-watsonx-ai==1.1.2  \
langchain==0.2.11 \
langchain-community==0.2.10 \
langchain-ibm==0.1.11 \
chromadb==0.4.24 \
pypdf==4.3.1 \
pydantic==2.9.1 \
huggingface_hub==0.23.0
```

Now, the environment is ready to create the application.

Construct the QA bot

It's time to construct the QA bot!

In this lab, you'll fill in the **missing code** to create a QA Bot.

Let's start by creating a new Python file to store your bot. Click the button below to create a new Python file, and call it `qabot.py`. If, for whatever reason, the button does not work, make the new file by going to `File --> New Text File`. Be sure to save the file as `qabot.py`.

Open **qabot.py** in IDE

You will populate `qabot.py` in the following sections with your bot.

Import the necessary libraries

Inside `qabot.py`, import the following from `gradio`, `ibm_watsonx.ai`, `langchain_ibm`, `langchain`, and `langchain_community`. The imported classes are necessary for initializing models with the correct credentials, splitting text, initializing a vector store, loading PDFs, generating a question-answer retriever, and using Gradio.

```
from ibm_watsonx_ai.foundation_models import ModelInference
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames as GenParams
from ibm_watsonx_ai.metanames import EmbedTextParamsMetaNames
from ibm_watsonx_ai import Credentials
from langchain_ibm import WatsonxLLM, WatsonxEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma
from langchain_community.document_loaders import PyPDFLoader
from langchain.chains import RetrievalQA
from huggingface_hub import HfFolder
import gradio as gr
# You can use this section to suppress warnings generated by your code:
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
warnings.filterwarnings('ignore')
```

Initialize the LLM

Please refer to the lab before proceeding with this section. Click Here

In this section, you'll fill in the **missing code** to create a initialize LLM.

You will now initialize the LLM by creating an instance of `WatsonxLLM`, a class in `langchain_ibm`. `WatsonxLLM` can use several underlying foundational models. In this particular example, you will use Mixtral 8x7B, although you could have used other models, such as Llama 3.3 70B. For a list of foundational models available in

watsonx.ai, refer to the documentation.

To initialize the LLM, use the following code into `qabot.py`. Note that you are required to initialize the model with a temperature of 0.5 and set the maximum token generation limit to 256.

```
## LLM
def get_llm():
    model_id = 'ibm/granite-3-2-8b-instruct'
    parameters = {
        .......,
        .......,
    }
    project_id = "skills-network"
    watsonx_llm = WatsonxLLM(
        model_id=......,
        url="https://us-south.ml.cloud.ibm.com",
        project_id=.......,
        params=.....,
    )
    return watsonx_llm
```

Task 1: Load document using LangChain for different sources

Define the PDF document loader

In this section, your task is to complete the provided code to set up the PDF document loader.

In this lab, you'll fill in the **missing code** to create a QA Bot.

To load PDF documents, you will use the `PyPDFLoader` refer to the lab class from the `langchain_community` library.

The syntax is quite straightforward:

- First, you create the PDF loader as an instance of `PyPDFLoader`.
- Then, you load the document and return the loaded document.

To incorporate the PDF loader in your bot, add the following to `qabot.py`. Here, you'll fill in the **missing code** to create a Document loader:

```
## Document loader
def document_loader(file):
    loader = .......(file.name)
    loaded_document = .......load()
    return loaded_document
```

Note: Capture a screenshot (save as `pdf_loader`) that displays the code used.

Task 2: Apply text splitting techniques

Define the text splitter

In this section, you will be given a set of code with blanks that you are requested to fill in to complete the tasks.

- The PDF document loader loads the content, but its `.load()` method does not split it into chunks. Therefore, you must define a separate document splitter in qabot.py to perform this text chunking.
- For this example, you'll define a `RecursiveCharacterTextSplitter` with a chunk size of 1000, though other splitters or parameter values are also viable.

Please refer to the lab under the **Split by Character** section before proceeding.

To incorporate the Text splitter in your bot, add the following to `qabot.py`. Here, you'll fill in the **missing code** to create a Text splitter:

```
## Text splitter
def text_splitter(data):
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=.......,
        chunk_overlap=....,
        length_function=.....,
    )
    chunks = text_splitter.split_documents(data)
    return chunks
```

Capture a screenshot (save as `code_splitter.png`) that displays the code used to split.

Task 3: Embed documents

Define the embedding model

Before continuing with this section, please review the lab under 'Build Model' for Embedding Text Parameters. Click Here

This embedding model is needed to convert chunks of text into vector representations.

- The following code defines a `watsonx_embedding()` function that returns an instance of `WatsonxEmbeddings`, a class from `langchain_ibm` that generates embeddings.
- In this particular case, the embeddings are generated using IBM's Slate 125M English embeddings model.

Complete the code using the respective components provided below, and then paste the finalized code into the `qabot.py` file.

```
## Embedding model
def watsonx_embedding():
    embed_params = {
        ..........
        ..........
    },
    }
    watsonx_embedding = WatsonxEmbeddings(
        model_id=............,
        url=............,
        project_id=.....,
        params=.........,
    )
    return watsonx_embedding
```

Capture a screenshot (save as `embedding.png`) that displays the code used to embed the following sentence and its corresponding results, which display the first five embedding numbers.

Task 4: Create and configure vector databases to store embeddings

Define the vector store

You'll need to fill in the blanks within the provided code for this task.

Add the following code to `qabot.py` to define a function that embeds the chunks using a embedding model and stores the embeddings in a ChromaDB vector store:

```
## Vector db
def vector_database(chunks):
    embedding_model = watsonx_embedding()
    vectordb = Chroma.from_documents(...... , ........)
    return vectordb
```

Capture a screenshot (save as `vectordb.png`) that displays the code used to create a Chroma vector database that stores the embeddings of the document.

Task 5: Develop a retriever to fetch document segments based on queries

Define the retriever

The retriever function offers multiple options. Please refer to the lab to explore them and gain a better understanding of this section.

Now that your vector store is defined, you must define a retriever that retrieves chunks of the document from it.
In this particular case,

- You will define a vector store-based retriever that retrieves information using a simple similarity search.

You'll need to fill in the blanks within the provided code for this task. Once completed, copy these lines and paste them into `qabot.py` to finalize the relevant section.

```
## Retriever
def retriever(file):
    splits = document_loader(....)
    chunks = text_splitter(....)
    vectordb = vector_database(.....)
    retriever = vectordb.as_retriever()
    return retriever
```

Capture a screenshot (save as `retriever.png`) that displays the code.

Task 6: Construct a QA Bot that leverages the LangChain and LLM to answer questions

Define a question-answering chain

Refer to the lab under the `RetrievalQA` section, which clearly explains how to build a QA bot that can answer questions from a document. You may also be interested in exploring additional exciting applications.

Finally, it is time to define a question-answering chain! In this particular example, you will use `RetrievalQA` from `langchain`, a chain that performs natural-language question-answering over a data source using retrieval-augmented generation (RAG).

To define your question-answering chain, you'll need to fill in the blanks in the code provided. Once you've completed it, copy and paste those lines directly into your `qabot.py` file to finalize this section.

```
## QA Chain
def retriever_qa(file, query):
    llm = get_llm()
    retriever_obj = retriever(file)
    qa = RetrievalQA.from_chain_type(llm=llm,
                                     chain_type=......,
                                     retriever=.......,
                                     return_source_documents=.......)
    response = qa.invoke(......)
    return response['result']
```

Let's recap how all the elements in our bot are linked.

- Note that `RetrievalQA` accepts an LLM (`get_llm()`) and a retriever object (an instance generated by `retriever()`) as arguments.
- However, the retriever is based on the vector store (`vector_database()`), which in turn needed an embeddings model (`watsonx_embedding()`) and chunks generated using a text splitter (`text_splitter()`).
- The text splitter, in turn, needed raw text, and this text was loaded from a PDF using `PyPDFLoader`.
- This effectively defines the core functionality of your QA bot!

Set up the Gradio interface

Given that you have created the core functionality of the bot, the final item to define is the Gradio interface. Click Here to refer the lab. Your Gradio interface should include:

- A file upload functionality (provided by the `File` class in Gradio)
- An input textbox where the question can be asked (provided by the `Textbox` class in Gradio)
- An output textbox where the question can be answered (provided by the `Textbox` class in Gradio)

Add the following code to `qabot.py` to add the Gradio interface:

```
# Create Gradio interface
rag_application = gr.Interface(
    fn=......,
    allow_flagging=.....,
    inputs=[
        gr.File(label="Upload PDF File", file_count="single", file_types=['.pdf'], type="filepath"),  # Drag and drop file upload
        gr.Textbox(label="Input Query", lines=2, placeholder="Type your question here...")
    ],
    outputs=gr.Textbox(label=.....),
    title=.......,
```

```
    description="Upload a PDF document and ask any question. The chatbot will try to answer using the provided document."
)
```

Add code to launch the application

Finally, you need to add one more line to `qabot.py` to launch your application using port 7860:

```
# Launch the app
rag_application.launch(server_name=......., server_port= .......)
```

After adding the above line, save `qabot.py`.

Verify `qabot.py`

Your `qabot.py` should now look like the following:

```
from ibm_watsonx_ai.foundation_models import ModelInference
from ibm_watsonx_ai.metanames import GenTextParamsMetaNames as GenParams
from ibm_watsonx_ai.metanames import EmbedTextParamsMetaNames
from ibm_watsonx_ai import Credentials
from langchain_ibm import WatsonxLLM, WatsonxEmbeddings
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma
from langchain_community.document_loaders import PyPDFLoader
from langchain.chains import RetrievalQA
from huggingface_hub import HfFolder
import gradio as gr
# You can use this section to suppress warnings generated by your code:
def warn(*args, **kwargs):
    pass
import warnings
warnings.warn = warn
warnings.filterwarnings('ignore')
## LLM
def get_llm():
    model_id = 'mistralai/mistral-small-3-1-24b-instruct-2503'
    parameters = {
        .......,
        .......,
    }
    project_id = "skills-network"
    watsonx_llm = WatsonxLLM(
        model_id=......,
        url="https://us-south.ml.cloud.ibm.com",
        project_id=.......,
        params=.....,
    )
    return watsonx_llm
## Document loader
def document_loader(file):
    loader = .......(file.name)
    loaded_document = .......load()
    return loaded_document
## Text splitter
def text_splitter(data):
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=.......,
        chunk_overlap=....,
        length_function=.....,
    )
    chunks = text_splitter.split_documents(data)
    return chunks
## Vector db
def vector_database(chunks):
    embedding_model = watsonx_embedding()
    vectordb = Chroma.from_documents(chunks, embedding_model)
    return vectordb
## Embedding model
def watsonx_embedding():
    embed_params = {
        ..........
        ..........
    },
    }
    watsonx_embedding = WatsonxEmbeddings(
        model_id=...........,
        url=...........,
        project_id=.....,
        params=.........,
```

```
        )
        return watsonx_embedding
    ## Retriever
    def retriever(file):
        splits = document_loader(file)
        chunks = text_splitter(splits)
        vectordb = vector_database(chunks)
        retriever = vectordb.as_retriever()
        return retriever
    ## QA Chain
    def retriever_qa(file, query):
        llm = get_llm()
        retriever_obj = retriever(file)
        qa = RetrievalQA.from_chain_type(llm=llm,
                                         chain_type=......,
                                         retriever=.......,
                                         return_source_documents=.......)
        response = qa.invoke(......)
        return response['result']
    # Create Gradio interface
    rag_application = gr.Interface(
        fn=......,
        allow_flagging=.....,
        inputs=[
            gr.File(label="Upload PDF File", file_count="single", file_types=['.pdf'], type="filepath"),  # Drag and drop file upload
            gr.Textbox(label="Input Query", lines=2, placeholder="Type your question here...")
        ],
        outputs=gr.Textbox(label=.....),
        title=.......,
        description="Upload a PDF document and ask any question. The chatbot will try to answer using the provided document."
    )
    # Launch the app
    rag_application.launch(server_name=......., server_port= .......)
```

Serve the application

To serve the application, paste the following into your Python terminal:

```
python3.11 qabot.py
```

If you cannot find an open Python terminal or the buttons on the above cell do not work, you can launch a terminal by going to `Terminal --> New Terminal`. However, if you launch a new terminal, do not forget to source the virtual environment you created at the beginning of this lab before running this line:

```
source my_env/bin/activate # activate my_env
```

Launch the application

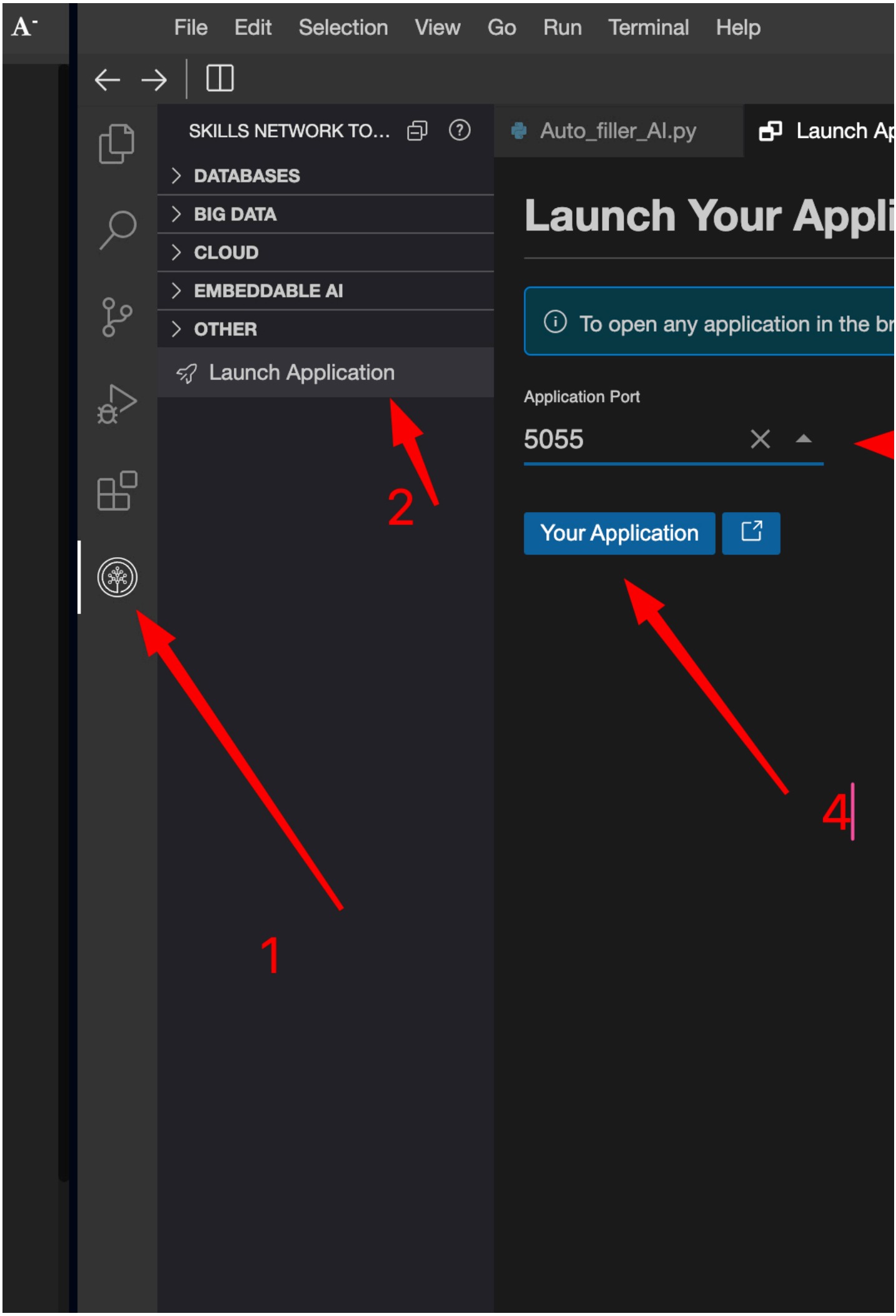
You are now ready to launch the served application! To do so, click on the following button:

Launch Application

If the above button does not work, use the following instructions:

1. Select the Skills Network extension.
2. Click **Launch Application**
3. Insert the port number (in this case, 7860, which is the server port we put in `qabot.py`)
4. Click **Your application** to launch the application.

Note: If the application does not work using **Your Application**, use the icon **Open in new browser tab**.


File Edit Selection View Go Run Terminal Help
SKILLS NETWORK TO...
DATABASES
BIG DATA
CLOUD
EMBEDDABLE AI
OTHER
Launch Application
Auto_filler_AI.py
Launch Ap
Launch Your Appli
To open any application in the br
Application Port
5055
Your Application
1
2
4

You can now interact with the application by uploading a readable PDF document and asking a question about its contents!

For best results, ensure that the PDF document is not too large. Large documents will fail with the current setup.

> *Capture a screenshot (save as `QA_bot.png`) that displays the QA bot interface you created.*

If you finish experimenting with the app and want to exit, press ctrl+c in the terminal and close the application tab.

Author

Author(s)

Kang Wang

Wojciech "Victor" Fulmyk

Kunal Makwana

Ricky Shi

Contributor(s)

Hailey Quach