



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.



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.

- [streamlit](#) 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
python2.11 -m pip install \
  gradle==4.4.0 \
  ibm-watson==1.1.2 \
  langchain==0.2.11 \
  langchain-community==0.2.10 \
  langchain-ibm==0.1.11 \
  chromadb==0.4.24 \
  pypdf==4.2.1 \
  pydantic==2.9.1 \
  huggingface_hub==0.22.0
```

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 `qbot.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 `qbot.py`.

Open `qabot.py` in IDE

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

Import the necessary libraries

Inside qbot.py, import the following from gradis, 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.

[illegible]

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 `Mistral 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 = {
        "temperature": 0.7,
        "max_tokens": 1000,
    }
    project_id = "skills-network"
    watsonx_llm = WatsonxLLM(
        model_id=model_id,
        url="https://us-south.ml.cloud.ibm.com",
        project_id=project_id,
        params=parameters
    )
    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 Q&A 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 `gabet.py`. Here, you'll fill in the **missing code** to create a Document loader:

```
## Document loader
```

```

from lib_watsumai_01_foundation_model import GetFoundationModel as GetFoundation
from lib_watsumai_01_metasomes import GetEmbedParamsEmbedToken as GetEmbedToken
from lib_watsumai_01 import Credentials
from langchain_community.vectorstores import Chroma
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain_community.vectorstores import Chroma
from langchain_community.document_loaders import PyPDFLoader
from langchain_community.document_loaders import PyPDFLoader
from huggingface_hub import login
import sys
import os
import warnings
def main():
    # Suppress warnings generated by your code
    warnings.filterwarnings("ignore")
    # Suppress warnings
    warnings.filterwarnings("ignore")
    # Suppress warnings
    warnings.filterwarnings("ignore")

```

[illegible]

Serve the application

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

```
python2.11 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 `qubeo.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**

The screenshot shows the VS Code interface with the 'Launch Application' configuration file open. The configuration is for a Flask application running on port 5055. The terminal output shows the command to run the application and the resulting output, including a warning about using a development server and the application's logs.

1. Arrow pointing to the 'Launch Application' button in the left sidebar.

2. Arrow pointing to the 'Launch Application' button in the top bar.

3. Arrow pointing to the 'Application Port' field (5055) in the configuration file.

4. Arrow pointing to the 'Your Application' button in the configuration file.

Terminal Output:

```

theia@theia-sinanz: /home/project/AI-Form-Filler x
theia@theia-sinanz: /home/project/AI-Form-Filler$ python3.11 Auto_filler_AI.py
* Serving Flask app 'Auto_filler_AI'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
* Running on http://127.0.0.1:5055
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 102-801-527
127.0.0.1 - - [03/Jan/2024 16:34:04] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [03/Jan/2024 16:34:04] "GET /static/autofill_tax.js HTTP/1.1" 200 -

```

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_hot.png) that displays the QA hot 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

Whichever "Victor" Fulms

[Reset Machine](#)

[Reset IDE](#)

Contributor(s)

[Reset Course](#)



Skills Network

