**Internal**

**Report**

**Build your own documentation Q&A chatbot**

**AI/ML**

|  |  |
| --- | --- |
| **Disclaimer** | Some parts of a report of this nature are inevitably subjective and/or based on information obtained in good faith from third party sources. Where opinions are expressed, they are the opinions of the individual author and/or the relevant third party source and not those of Sagentia Innovation or its group. Furthermore, if new facts become available and/or the commercial or technological environment evolves, the relevance and applicability of opinions and conclusions in this report may be affected. Accordingly, while this report has been compiled in good faith, no representation or warranty, express or implied, is made by Sagentia Innovation as to its completeness, accuracy or fairness. Except where limited by law, neither Sagentia Innovation nor its group shall be responsible for any actions taken or not taken as a result of any opinions and conclusions provided in this report and you agree to indemnify Sagentia Innovation, its group and/or personnel against any liability resulting from the same. |

Executive summary

This document provides instructions on how to setup a documentation question and answer chatbot on any pfd, excel, word ant text files you place in a folder (without data leakage of intellectual property).

Two implementations are provided:

* **Microsoft Azure Open AI based**. This will require a Microsoft Azure account and to fill in a form to access Microsoft Open AI services (if not already done so). Confidentiality is covered by the company Enterprise agreement with Microsoft. This will use GPT3.5-turbo. This is the ChatGPT LLM widely available to the public.
* **GPT4ALL**. This is a standalone implementation which does not require internet access once setup. You will need a powerful PC for this one as the inference will be performed on the PC.

The implementation uses Semantic Search as opposed to LLM fine tuning on the documents.

The creation of the Doc QA chatbot should take no longer than 30 minutes! Such is the power of the democratisation of AI/ML tools.

Overall, the Microsoft Azure Open AI based performs very well out of the box on any document content whereas for the GPT4ALL you may need to choose the LLM and embeddings to suit your requirements.

Contents

[**1 Introduction**](#_Toc148384215)

[**2 Environment Setup**](#_Toc148384216)

[**3 Azure Open AI**](#_Toc148384217)

[*3.1 Create Azure Open AI Deployment*](#_Toc148384218)

[*3.2 Configure Environment*](#_Toc148384219)

[*3.3 Create Indexer*](#_Toc148384220)

[*3.4 Create the chat bot application*](#_Toc148384221)

[*3.5 Run the DocQAChatBot*](#_Toc148384222)

[**4 GPT4ALL**](#_Toc148384223)

[*4.1 Create Indexer*](#_Toc148384224)

[*4.2 Download LLM model from GPT4ALL*](#_Toc148384225)

[*4.3 Create the chat bot application*](#_Toc148384226)

[*4.4 Run the chat bot application*](#_Toc148384227)

[**5 Conclusion**](#_Toc148384228)

[**Appendix A Requirements.txt**](#_Toc148384229)

# Introduction

Many AI products are coming out these days that allow you to interact with your own private PDFs and documents. But how do they work? And how do you build one? Behind the scenes, it’s easy.

At a basic level, how does a document chatbot work? At its core, it’s just the same as ChatGPT. On ChatGPT, you can copy a bunch of text into the prompt, and then ask ChatGPT to summarise the text for you or generate some answers based on the text.

**Interacting with multiple documents**

Where it gets a little more interesting is when the document is very large, or there are many documents we want to interact with. Passing in all the information from these documents into a request to an LLM (Large Language Model) is impossible since these requests usually have size (token) limits.

This is where Semantic Search with embeddings comes to the rescue without the need to fine tuning the LLM.

**Embeddings and Vector Stores**

We want a way to send only relevant bits of information from our documents to the LLM prompt. Embeddings and vector stores can help us with this.

An embedding allows us to organise and categorise a text based on its semantic meaning. So, we split our documents into lots of little text chunks and use embeddings to characterise each bit of text by its semantic meaning. An embedding transformer is used to convert a bit of text into an embedding.

An embedding categorises a piece of text by giving it a vector (coordinate) representation. That means that vectors (coordinates) that are close to each other represent pieces of information that have a similar meaning to each other. The embedding vectors are stored inside a vector store, along with the chunks of text corresponding to each embedding.

Once we have a prompt, we can use the embeddings transformer to match it with the bits of text that are most semantically relevance to it, so we know how a way to match our prompt with other related bits of text from the vector store.

Now that we have a smaller subset of the information, which is relevant to our prompt, we can query the LLM with our initial prompt, while passing in only the relevant information as the context to our prompt.

**Chatbox principle of operation**

Let’s explain with a simple example of a PDF file. In fact, it can be any file (i.e., docx, xlsx, pptx, txt and so on) if we can convert it into text automatically.

The steps we will need to follow are (see Figure 1):

* Split all the documents into small chunks of text.
* Pass each chunk of text into an embedding transformer to turn it into an embedding.
* Store the embeddings and related pieces of text in a vector store.

A diagram of a diagram of a diagram

Description automatically generated with medium confidence

Figure 1 Embeddings workflow

Once we have loaded our content as embeddings into the vector store, we are back to a similar situation as to when we only had one PDF to interact with. As in, we are now ready to pass information into the LLM prompt. However, instead of passing in all the documents as a source for our context to the chain, as we did initially, we will pass in our vector store as a source, which the chain will use to retrieve only the relevant text based on our question and send that information only inside the LLM prompt.

A diagram of a diagram of a diagram

Description automatically generated

Figure 2 Chatbot workflow

# Environment Setup

We will use Python virtual environments for our implementation. If not installed download the latest Python version from:

[Download Python | Python.org](https://www.python.org/downloads/)

Launch the Windows Command prompt.

We will create a virtual environment called DocQAChatBot. You can choose any virtual environment name you wish.

python -m venv DocQAChatBot

Activate virtual environment.

.\DocQAChatBot\Scripts\activate

If the virtual environment is activated you will see (DocQAChatBot) before the prompt.

A black screen with white text

Description automatically generated

Install the necessary packages by creating a requirements.txt file from Appendix A and running the command:

pip install -r requirements.txt

Go into the DocQAChatBot folder and change directory

cd DocQAChatBot

Create “Docs” folder and place the documents to be indexed for Q&A

mkdir Docs

# Azure Open AI

## Create Azure Open AI Deployment

You will need the following:

* Microsoft Azure Account – You should be able to get £45 credit every month if you have a Visual Studio subscription.
* Azure Open AI Studio Access– You will need to ask Microsoft access by filling in a form. No sweat I got it enabled in 2 days! You only need enable it once.

Create the following deployment as below.

A screenshot of a computer

Description automatically generated

Figure 3 Microsoft Azure Open AI deployments

## Configure Environment

Create an .env file with the contents below.

You need to get the values of **OPENAI\_DEPLOYMENT\_ENDPOINT** and **OPENAI\_API\_KEY** from your Microsoft Azure Deployment.

OPENAI\_DEPLOYMENT\_ENDPOINT =<YOUR END POINT HERE>

OPENAI\_API\_KEY = <YOUR KEY HERE>

OPENAI\_DEPLOYMENT\_NAME = "gpt-35-turbo"

OPENAI\_DEPLOYMENT\_VERSION = "2023-07-01-preview"

OPENAI\_MODEL\_NAME="gpt-35-turbo"

OPENAI\_ADA\_EMBEDDING\_DEPLOYMENT\_NAME = "text-embedding-ada-002"

OPENAI\_ADA\_EMBEDDING\_MODEL\_NAME = "text-embedding-ada-002"

## Create Indexer

Create file app\_indexer\_open\_ai.py to create a vector store of embeddings (i.e., indexer)

Simply copy and paste the code below into VS Code or any editor of your choice and save it.

from langchain.document\_loaders import PyPDFLoader

from langchain.document\_loaders import Docx2txtLoader

from langchain.document\_loaders import TextLoader

from langchain.document\_loaders import UnstructuredPowerPointLoader

from langchain.document\_loaders.merge import MergedDataLoader

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.vectorstores import FAISS

from langchain.document\_loaders import DirectoryLoader

from dotenv import load\_dotenv

import openai

import os

#load environment variables

load\_dotenv()

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

OPENAI\_DEPLOYMENT\_ENDPOINT = os.getenv("OPENAI\_DEPLOYMENT\_ENDPOINT")

OPENAI\_DEPLOYMENT\_NAME = os.getenv("OPENAI\_DEPLOYMENT\_NAME")

OPENAI\_MODEL\_NAME = os.getenv("OPENAI\_MODEL\_NAME")

OPENAI\_DEPLOYMENT\_VERSION = os.getenv("OPENAI\_DEPLOYMENT\_VERSION")

OPENAI\_ADA\_EMBEDDING\_DEPLOYMENT\_NAME = os.getenv("OPENAI\_ADA\_EMBEDDING\_DEPLOYMENT\_NAME")

OPENAI\_ADA\_EMBEDDING\_MODEL\_NAME = os.getenv("OPENAI\_ADA\_EMBEDDING\_MODEL\_NAME")

#init Azure OpenAI

openai.api\_type = "azure"

openai.api\_version = OPENAI\_DEPLOYMENT\_VERSION

openai.api\_base = OPENAI\_DEPLOYMENT\_ENDPOINT

openai.api\_key = OPENAI\_API\_KEY

if \_\_name\_\_ == "\_\_main\_\_":

    embeddings=OpenAIEmbeddings(deployment=OPENAI\_ADA\_EMBEDDING\_DEPLOYMENT\_NAME,

                                model=OPENAI\_ADA\_EMBEDDING\_MODEL\_NAME,

                                openai\_api\_base=OPENAI\_DEPLOYMENT\_ENDPOINT,

                                openai\_api\_type="azure",

                                chunk\_size=1)

    pdf\_loader  = DirectoryLoader('./Docs/', glob="\*\*/\*.pdf")

    txt\_loader  = DirectoryLoader('./Docs/', glob="\*\*/\*.txt")

    word\_loader = DirectoryLoader('./Docs/', glob="\*\*/\*.docx")

    ppt\_loader  = DirectoryLoader('./Docs/', glob="\*\*/\*.pptx")

    loaders = [pdf\_loader, txt\_loader, word\_loader, ppt\_loader]

    loader\_all = MergedDataLoader(loaders)

    pages = loader\_all.load\_and\_split()

    print(f"Total number of document chunks: {len(pages)}")

    #Use Langchain to create the embeddings using text-embedding-ada-002

    db = FAISS.from\_documents(documents=pages, embedding=embeddings)

    #save the embeddings into FAISS vector store

    db.save\_local("./dbs\_open\_ai/documentation/faiss\_index")

## Create the chat bot application

Create file app\_chainlit\_open\_ai.py with the contents below.

Simply copy and paste the code below into VS Code or any editor of your choice and save it.

import os

from dotenv import load\_dotenv

import openai

from typing import List

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.chains import ConversationalRetrievalChain

from langchain.docstore.document import Document

from langchain.memory import ChatMessageHistory, ConversationBufferMemory

#from langchain.chains import RetrievalQA

from langchain.vectorstores import FAISS

from langchain.chains.question\_answering import load\_qa\_chain

from langchain.chat\_models import AzureChatOpenAI

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.vectorstores import FAISS

from langchain.chains import ConversationalRetrievalChain

from langchain.prompts import PromptTemplate

import chainlit as cl

# Load environment variables

load\_dotenv()

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

OPENAI\_DEPLOYMENT\_ENDPOINT = os.getenv("OPENAI\_DEPLOYMENT\_ENDPOINT")

OPENAI\_DEPLOYMENT\_NAME = os.getenv("OPENAI\_DEPLOYMENT\_NAME")

OPENAI\_MODEL\_NAME = os.getenv("OPENAI\_MODEL\_NAME")

OPENAI\_DEPLOYMENT\_VERSION = os.getenv("OPENAI\_DEPLOYMENT\_VERSION")

OPENAI\_ADA\_EMBEDDING\_DEPLOYMENT\_NAME = os.getenv("OPENAI\_ADA\_EMBEDDING\_DEPLOYMENT\_NAME")

OPENAI\_ADA\_EMBEDDING\_MODEL\_NAME = os.getenv("OPENAI\_ADA\_EMBEDDING\_MODEL\_NAME")

QUESTION\_PROMPT = PromptTemplate.from\_template("""Given the following conversation and a follow up question, rephrase the follow up question to be a standalone question.

    Chat History:

    {chat\_history}

    Follow Up Input: {question}

    Standalone question:""")

@cl.on\_chat\_start

async def on\_chat\_start():

    # Configure OpenAI API

    openai.api\_type = "azure"

    openai.api\_base = os.getenv('OPENAI\_API\_BASE')

    openai.api\_key = os.getenv("OPENAI\_API\_KEY")

    openai.api\_version = os.getenv('OPENAI\_API\_VERSION')

    llm = AzureChatOpenAI(deployment\_name=OPENAI\_DEPLOYMENT\_NAME,

                      model\_name=OPENAI\_MODEL\_NAME,

                      openai\_api\_base=OPENAI\_DEPLOYMENT\_ENDPOINT,

                      openai\_api\_version=OPENAI\_DEPLOYMENT\_VERSION,

                      openai\_api\_key=OPENAI\_API\_KEY,

                      openai\_api\_type="azure")

    embeddings=OpenAIEmbeddings(deployment=OPENAI\_ADA\_EMBEDDING\_DEPLOYMENT\_NAME,

                                model=OPENAI\_ADA\_EMBEDDING\_MODEL\_NAME,

                                openai\_api\_base=OPENAI\_DEPLOYMENT\_ENDPOINT,

                                openai\_api\_type="azure",

                                chunk\_size=1)

    # Initialize gpt-35-turbo and our embedding model

    # load the faiss vector store we saved into memory

    vectorStore = FAISS.load\_local("./dbs\_open\_ai/documentation/faiss\_index", embeddings)

    # use the faiss vector store we saved to search the local document

    # increase the number of k for more accurate answers but higher cost due to hifgher nubmer of tokens submitted

    retriever = vectorStore.as\_retriever(search\_type="similarity", search\_kwargs={"k":2})

    message\_history = ChatMessageHistory()

    memory = ConversationBufferMemory(

        memory\_key="chat\_history",

        output\_key="answer",

        chat\_memory=message\_history,

        return\_messages=True,

    )

    qa = ConversationalRetrievalChain.from\_llm(llm=llm,

                                            retriever=retriever,

                                            condense\_question\_prompt=QUESTION\_PROMPT,

                                            return\_source\_documents=True,

                                            memory=memory,

                                            verbose=False)

    cl.user\_session.set("chain", qa)

@cl.on\_message

async def main(message):

    qa = cl.user\_session.get("chain")  # type: ConversationalRetrievalChain

    cb = cl.AsyncLangchainCallbackHandler()

    res = await qa.acall(message, callbacks=[cb])

    answer = res["answer"]

    source\_documents = res["source\_documents"]  # type: List[Document]

    text\_elements = []  # type: List[cl.Text]

    if source\_documents:

        for source\_idx, source\_doc in enumerate(source\_documents):

            source\_name = f"source\_{source\_idx}"

            # Create the text element referenced in the message

            text\_elements.append(

                cl.Text(content=source\_doc.page\_content, name=source\_name)

            )

        source\_names = [text\_el.name for text\_el in text\_elements]

        if source\_names:

            answer += f"\nSources: {', '.join(source\_names)}"

        else:

            answer += "\nNo sources found"

    await cl.Message(content=answer, elements=text\_elements).send()

## Run the DocQAChatBot

Make sure virtual environment is active.

Run the command to index your docs if not already done so. You need to do it every time you change the Docs folder contents.

Python app\_indexer\_open\_ai.py

To launch the QA chat bot simply run:

chainlit run app\_chainlit\_open\_ai.py

Your default web browser will launch automatically.

Simply type a question relevant to your documents. I asked about skin electronics from a an article related to skin electronics.

A screenshot of a computer

Description automatically generated

# GPT4ALL

## Create Indexer

Create the file app\_indexer\_hugging\_face.py to create a vector store of embeddings (i.e., indexer)

Simply copy and paste the code below into VS Code or any editor of your choice and save it.

I choose the all-mpnet-base-v2 embeddings from hugging face. You may need to choose a different embeddings model if it doesn’t performed well on your data .

from langchain.document\_loaders import PyPDFLoader

from langchain.document\_loaders import Docx2txtLoader

from langchain.document\_loaders import TextLoader

from langchain.document\_loaders import UnstructuredPowerPointLoader

from langchain.document\_loaders.merge import MergedDataLoader

from langchain.embeddings.openai import OpenAIEmbeddings

from langchain.vectorstores import FAISS

import os

from langchain.document\_loaders import DirectoryLoader

from langchain.embeddings import HuggingFaceEmbeddings

from langchain.text\_splitter import RecursiveCharacterTextSplitter

if \_\_name\_\_ == "\_\_main\_\_":

    embeddings = HuggingFaceEmbeddings(model\_name="sentence-transformers/all-mpnet-base-v2")

    pdf\_loader  = DirectoryLoader('./Docs/', glob="\*\*/\*.pdf")

    txt\_loader  = DirectoryLoader('./Docs/', glob="\*\*/\*.txt")

    word\_loader = DirectoryLoader('./Docs/', glob="\*\*/\*.docx")

    ppt\_loader  = DirectoryLoader('./Docs/', glob="\*\*/\*.pptx")

    loaders = [pdf\_loader, txt\_loader, word\_loader, ppt\_loader]

    loader\_all = MergedDataLoader(loaders)

    pages = loader\_all.load\_and\_split()

    text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1536, chunk\_overlap=64)

    texts = text\_splitter.split\_documents(pages)

    print(texts)

    print(f"Total number of documents: {len(texts)}")

    #Use Langchain to create the embeddings using text-embedding-ada-002

    db = FAISS.from\_documents(documents=texts, embedding=embeddings)

    #save the embeddings into FAISS vector store

    db.save\_local("./dbs\_gpt4all/documentation/faiss\_index")

## Download LLM model from GPT4ALL

Download your LLM from the web site below.[GPT4All](https://gpt4all.io/index.html)

I choose the llama-2-7b which is free for commercial use. Note that not all are free for commercial use. A screenshot of a computer

Description automatically generated

## Create the chat bot application

Create the file app\_chainlit\_gpt4all.py with the contents below.

Simply copy and paste the code below into VS Code or any editor of your choice and save it.

import os

from langchain.llms import GPT4All

from langchain.embeddings import HuggingFaceEmbeddings

from typing import List

from langchain.docstore.document import Document

from langchain.memory import ChatMessageHistory, ConversationBufferMemory

from langchain.vectorstores import FAISS

from langchain.chains import ConversationalRetrievalChain

from langchain.prompts import PromptTemplate

from langchain.callbacks.streaming\_stdout import StreamingStdOutCallbackHandler

import chainlit as cl

QUESTION\_PROMPT = PromptTemplate.from\_template("""Given the following conversation and a follow up question, rephrase the follow up question to be a standalone question.

    Chat History:

    {chat\_history}

    Follow Up Input: {question}

    Standalone question:""")

@cl.on\_chat\_start

async def on\_chat\_start():

    local\_path = (

        "./llama-2-7b-chat.ggmlv3.q4\_0.bin"  # replace with your desired local file path

    )

    # Callbacks support token-wise streaming

    callbacks = [StreamingStdOutCallbackHandler()]

    # Verbose is required to pass to the callback manager

    llm = GPT4All(model=local\_path, callbacks=callbacks, verbose=True)

    embeddings = HuggingFaceEmbeddings(model\_name="sentence-transformers/all-mpnet-base-v2")

    # Initialize our embedding model

    #load the faiss vector store we saved into memory

    vectorStore = FAISS.load\_local("./dbs\_gpt4all/documentation/faiss\_index", embeddings)

    # use the faiss vector store we saved to search the local document

    # change k to a higher value for more accurate answers

    retriever = vectorStore.as\_retriever(search\_type="similarity", search\_kwargs={"k":6})

    message\_history = ChatMessageHistory()

    memory = ConversationBufferMemory(

        memory\_key="chat\_history",

        output\_key="answer",

        chat\_memory=message\_history,

        return\_messages=True,

    )

    qa = ConversationalRetrievalChain.from\_llm(llm=llm,

                                            retriever=retriever,

                                            condense\_question\_prompt=QUESTION\_PROMPT,

                                            return\_source\_documents=True,

                                            memory=memory,

                                            verbose=False)

    cl.user\_session.set("chain", qa)

@cl.on\_message

async def main(message):

    qa = cl.user\_session.get("chain")  # type: ConversationalRetrievalChain

    cb = cl.AsyncLangchainCallbackHandler()

    res = await qa.acall(message, callbacks=[cb])

    answer = res["answer"]

    source\_documents = res["source\_documents"]  # type: List[Document]

    text\_elements = []  # type: List[cl.Text]

    if source\_documents:

        for source\_idx, source\_doc in enumerate(source\_documents):

            source\_name = f"source\_{source\_idx}"

            # Create the text element referenced in the message

            text\_elements.append(

                cl.Text(content=source\_doc.page\_content, name=source\_name)

            )

        source\_names = [text\_el.name for text\_el in text\_elements]

        if source\_names:

            answer += f"\nSources: {', '.join(source\_names)}"

        else:

            answer += "\nNo sources found"

    await cl.Message(content=answer, elements=text\_elements).send()

## Run the DocQAChatBot

Make sure virtual environment is active.

Run the command to index your docs if not already done so. You need to do it every time you change the Docs folder contents.

Python app\_indexer\_hugging\_face.py

To launch the QA chat bot simply run:

chainlit run app\_chainlit\_gpt4all.py

Your default web browser will launch automatically.

Simply type a question relevant to your documents.

# Conclusion

The objective of this document was to provide a basic Document Q&A tool which of course needs to be further refined.

If you use this document, then I would appreciate you update it and let me know of your improvements.

Have fun!

1. Requirements.txt

Requirements.txt

aiofiles==23.2.1

aiohttp==3.8.6

aiosignal==1.3.1

altair==5.1.2

annotated-types==0.6.0

antlr4-python3-runtime==4.9.3

anyio==3.7.1

asttokens==2.4.0

async-timeout==4.0.3

asyncer==0.0.2

attrs==23.1.0

backcall==0.2.0

backoff==2.2.1

beautifulsoup4==4.12.2

bidict==0.22.1

blinker==1.6.3

cachetools==5.3.1

certifi==2023.7.22

cffi==1.16.0

chainlit==0.7.2

chardet==5.2.0

charset-normalizer==3.3.0

click==8.1.7

colorama==0.4.6

coloredlogs==15.0.1

comm==0.1.4

contourpy==1.1.1

cryptography==41.0.4

cycler==0.12.1

dataclasses-json==0.5.14

debugpy==1.8.0

decorator==5.1.1

Deprecated==1.2.14

docx2txt==0.8

EbookLib==0.18

effdet==0.4.1

emoji==2.8.0

et-xmlfile==1.1.0

executing==2.0.0

faiss-cpu==1.7.4

fastapi==0.100.1

fastapi-socketio==0.0.10

filelock==3.12.4

filetype==1.2.0

flatbuffers==23.5.26

fonttools==4.43.1

frozenlist==1.4.0

fsspec==2023.9.2

gitdb==4.0.10

GitPython==3.1.37

googleapis-common-protos==1.61.0

gpt4all==1.0.12

greenlet==3.0.0

grpcio==1.59.0

h11==0.14.0

httpcore==0.18.0

httpx==0.25.0

huggingface-hub==0.17.3

humanfriendly==10.0

idna==3.4

importlib-metadata==6.8.0

iopath==0.1.10

ipykernel==6.25.2

ipython==8.16.1

jedi==0.19.1

Jinja2==3.1.2

joblib==1.3.2

jsonpatch==1.33

jsonpointer==2.4

jsonschema==4.19.1

jsonschema-specifications==2023.7.1

jupyter\_client==8.4.0

jupyter\_core==5.4.0

kiwisolver==1.4.5

langchain==0.0.312

langdetect==1.0.9

langsmith==0.0.43

layoutparser==0.3.4

Lazify==0.4.0

lxml==4.9.3

Markdown==3.5

markdown-it-py==3.0.0

MarkupSafe==2.1.3

marshmallow==3.20.1

matplotlib==3.8.0

matplotlib-inline==0.1.6

mdurl==0.1.2

mpmath==1.3.0

msg-parser==1.2.0

multidict==6.0.4

mypy-extensions==1.0.0

nest-asyncio==1.5.8

networkx==3.1

nltk==3.8.1

nodeenv==1.8.0

numpy==1.26.0

olefile==0.46

omegaconf==2.3.0

onnx==1.14.1

onnxruntime==1.15.1

openai==0.28.1

opencv-python==4.8.1.78

openpyxl==3.1.2

opentelemetry-api==1.20.0

opentelemetry-exporter-otlp==1.20.0

opentelemetry-exporter-otlp-proto-common==1.20.0

opentelemetry-exporter-otlp-proto-grpc==1.20.0

opentelemetry-exporter-otlp-proto-http==1.20.0

opentelemetry-instrumentation==0.41b0

opentelemetry-proto==1.20.0

opentelemetry-sdk==1.20.0

opentelemetry-semantic-conventions==0.41b0

packaging==23.2

pandas==2.1.1

parso==0.8.3

pdf2image==1.16.3

pdfminer.six==20221105

pdfplumber==0.10.2

pickleshare==0.7.5

Pillow==10.0.1

platformdirs==3.11.0

portalocker==2.8.2

prisma==0.10.0

prompt-toolkit==3.0.39

protobuf==4.24.4

psutil==5.9.5

pure-eval==0.2.2

pyarrow==13.0.0

pyclipper==1.3.0.post5

pycocotools==2.0.7

pycparser==2.21

pydantic==2.4.2

pydantic\_core==2.10.1

pydeck==0.8.1b0

Pygments==2.16.1

PyJWT==2.8.0

pypandoc==1.11

pyparsing==3.1.1

pypdf==3.16.4

PyPDF2==3.0.1

pypdfium2==4.21.0

pyreadline3==3.4.1

pytesseract==0.3.10

python-dateutil==2.8.2

python-docx==1.0.1

python-dotenv==1.0.0

python-engineio==4.7.1

python-graphql-client==0.4.3

python-iso639==2023.6.15

python-magic==0.4.27

python-multipart==0.0.6

python-pptx==0.6.21

python-socketio==5.9.0

pytz==2023.3.post1

pywin32==306

PyYAML==6.0.1

pyzmq==25.1.1

rapidfuzz==3.4.0

rapidocr-onnxruntime==1.3.7

referencing==0.30.2

regex==2023.10.3

requests==2.31.0

rich==13.6.0

rpds-py==0.10.6

safetensors==0.4.0

scikit-learn==1.3.1

scipy==1.11.3

sentence-transformers==2.2.2

sentencepiece==0.1.99

shapely==2.0.1

simple-websocket==1.0.0

six==1.16.0

smmap==5.0.1

sniffio==1.3.0

soupsieve==2.5

SQLAlchemy==2.0.21

stack-data==0.6.3

starlette==0.27.0

streamlit==1.27.2

sympy==1.12

syncer==2.0.3

tabulate==0.9.0

tenacity==8.2.3

threadpoolctl==3.2.0

tiktoken==0.5.1

timm==0.9.7

tokenizers==0.14.1

toml==0.10.2

tomli==2.0.1

tomlkit==0.12.1

toolz==0.12.0

torch==2.1.0

torchvision==0.16.0

tornado==6.3.3

tqdm==4.66.1

traitlets==5.11.2

transformers==4.34.0

typing-inspect==0.9.0

typing\_extensions==4.8.0

tzdata==2023.3

tzlocal==5.1

unstructured==0.10.21

unstructured-inference==0.7.2

unstructured.pytesseract==0.3.12

uptrace==1.20.2

urllib3==2.0.6

uvicorn==0.23.2

validators==0.22.0

watchdog==3.0.0

watchfiles==0.20.0

wcwidth==0.2.8

websockets==11.0.3

wrapt==1.15.0

wsproto==1.2.0

xlrd==2.0.1

XlsxWriter==3.1.7

yarl==1.9.2

zipp==3.17.0

About Sagentia Innovation

Sagentia Innovation is a global science, product and technology development company. Our mission is to help companies maximise the value of their investments in R&D. We partner with clients in the consumer, industrial, medical and oil & gas sectors to help them understand the technology and market landscape, decide their future strategy, solve the complex science and technology challenges and deliver commercially successful products.

Sagentia Innovation is a Science Group company

About Science Group plc

Science Group plc (AIM:SAG) is an international consulting services group supporting the product innovation lifecycle to enable our clients to deliver on their investments in R&D. Our services include: Applied Science, Product Development, Technology Advisory and Regulatory. These services are combined with vertical market expertise in medical, food & beverage, industrial, chemical, energy and consumer sectors. With offices throughout Europe and North America and over 30 languages written and spoken, Science Group supports a global client base in over 100 countries.

[info@sciencegroup.com](mailto:info@sciencegroup.com)  
[www.sciencegroup.com](http://www.sciencegroup.com/)