APPLIED COMPUTER SCIENCES

FH JOANNEUM University of Applied Sciences

Project Documentation

Winter semester 2024/25

Susanne Jandl

Offline Chatbot

Version 02-gh | 2025-01-10

Table of contents

1 Offline Chatbot Project

1.1 General Information

- Repository: https://github.com/SusanneJandl/offline-chatbot-project-FHJ

This documentation is tightly connected to the Repository. This Documentation includes absolute links to the repository. To benefit from the included links it is recommended to download and open it in a PDF reader. From inside a local copy of the repository Documentation_gh_local_v02_en.pdf including relative links should be used.

The repsitory includes different approaches to create an offline chatbot or parts of it. The final solution can be found in the Final_Project folder. To start the project follow the instruction of the README_FP in Final_Project folder.

1.2 Problem

Nowadays many websites provide a chatbot function for customer support. Many out-of-the-box solutions are available for implementation. If none of this solutions fits well it is easy to create a custom chatbot using online APIs like OpenAI API or Huggingface inference API, which are paid services at low cost.

But what if there is a local application without internet access and customers should be provided with the benefits of a chatbot? This is where this project starts:
To implement an offline chatbot it is necessary to download and store AI models locally. Depending on available hardware resources it can be challenging to find fitting models. Usually large language models (LLMs) of smaller size can not offer results of the same quality as larger models do.

The specific information about the topic the chatbot shall help with must be available for the LLM. To achieve this, there are the possibilities to either create a vector store with the necessary information or fine tune a pretrained large language model. For creating a vector store a second model has to be stored locally for creating and comparing vectors.

Besides the main problem of finding a solution to find and store appropriate models and provide them with information, there are additional goals for this project:
The chatbot itself shall be a C# implementation. It has to support at least English and German. The language should be recognized automatically.

The chatbot shall be started by button click in a different application. The button click shall send an information about the pagename in addition. This will help to predefine the context to be used for answering questions. Like that the chatbot can be used for different applications in future and will be able to answer questions more accurately by working with minimal resources.

All included packages and AI models have to be free for commercial use. Licenses have to be checked and notes added to the project if applicable.

1.3 Motivation

At work I got the task to develop an offline chatbot. The chatbot will support users to use the locally installed software, without internet access.

For this project I will include the main points of the tasks, but generalize it a bit. I will not focus on a specific software, but use a sample app that demonstrates providing different information depending on which page the chatbot is started from. This will result in good progress in finding a technical solution. Another advantage will be that the result of the project will be more generalized and easy to adapt to future needs.

1.4 Research and different approaches

For creating a chatbot, also a local one, there are many frameworks and thousands of AI models available. The first step was trying to get an overview of possibilities. Research included finding ready-to-use solutions on the market, investigate available large language models, and frameworks as well as checking tutorials to get an idea how to get started with creating an offline chatbot.

1.4.1 Market research

An important first step before starting a software project is to find out if there is a solution available onthe market that fits the needs. There are two out-of-the-box solutions that were worth to take a closer look.

	GPT4All	JAN
works offline	✓	✓
supports using information from documents	✓	(✓)
free of charge		✓
use for own UI (run as API)		✓
customizable UI		

Table 1: GPT4All and JAN overview

GPT4All and JAN can be downloaded and installed locally. Both work completely offline as soon as models to use are downloaded. The UI of GPT4All and JAN does not fit for using it as a chatbot. Regarding the new experimental version of JAN that includes retrieving information from provided documents and the option to use it as a local API might be a good and easy approach for creating a local chatbot with an own UI in future. An important point to be aware of is the fact that the software contains a lot of functions that are not necessary for the specific usecase. These functions need resources for storage and possibly running in the background without being needed.

1.4.2 Fine tuning a model

The first idea was to fine tune a model with specific information the chatbot needs. For that purpose sample training data was created. It consisted of only five question and answer pairs for the first try. A pretrained model was selected and a python script was written to fine tune the model. A minimal implementation on training a model is included in the 01_train_model folder in the Repository.

It worked well as long as the question did not vary too much from the training data and no other information was added (like chat history). It would need multiple question and answer pairs for one fact to be trained For example a question and answer pair in the training data is: ”What is AI?”. Different ways to ask for it, have to be added, like: ” What does AI stand for?”, ”What does AI mean?” or ”Tell me the meaning of the abbreviation ”AI”.”. In addition this has to be done for all languages used. Depending on the pretraining of the model the effort can vary.

After fine tuning a model, the new information is only a part of the whole knowledge the model has. It has no higher impact on the answer than all the other information the model had before.

1.4.3 Use Langflow

Link: https://www.langflow.org

Langflow is a low-code tool to create AI workflows. Workflows can be created in an intuitive UI including a playground to use the created workflow. Langflow can be executed in the background as an API as well. Workflows can be down- and uploaded in the form of json files.

A chat widget that can be implemented very easily is available. An example for using the widget is represented in the 03_chat_with_widget folder in the Repository.

For offline use it is possible to use Ollama for providing large language models, which was chosen here. Ollama provides a local API that manages AI models and their usage for chat and embeddings. A closer description is in section *1.5.6.1 Ollama* on page 10.

Two workflows were created for using Retrieval Augmented Generation (RAG). The first workflow stores a vector store that is created from a folder containing documents including relevant information. To do so the text of the documents is extracted and then an embedding model (provided by Ollama) converts the text into vectors.

The second workflow starts with the user query. The embedding model converts the user query into a vector and compares it to the vectors stored in the vector store. So the relevant context based on the user query is found and then provided as part of the prompt (instruction for the chatbot) alongside the query itself. This prompt is passed to the large language model, that finally creates the answer to the user query.

This approach worked well, but was too slow. It took up to 5 minutes until an answer was created. A downside when using langflow is the installation that includes a lot of packages that are not needed. This was a large improvement to the first tries, where it took up to 20 minutes to retrieve an answer. A new finding was, that context creation worked fine and fast enough with langflow. So a new workflow, with the context as output was creted a python script to retrieve it was written. This was part of the next approach with using Ollama directly in C#.

1.4.4 OllamaSharp and Python.Included in C#

This approach includes a C# console application, which can be accessed in the 02_chat_with_langflow_vs folder in the Repository.

Obviously a console application will not be the final version, but it is the simplest way to try out functions and debugging is easy as well. Part of this application was to use the created python package for retrieving context using langflow. For that purpose Python.Included Nuget package was used. This way python packages can be used in C# code. An important part that is still the same in the final version is the usage of OllamaSharp that enables streaming. As LLMs produce their answers token by token (a token is a word or part of a word) the stream starts when the first token is received. That means that there could be a response time of three minutes, but the stream starts after 10 seconds and outputs token by token until the answer is completed after three minutes. That is a huge advantage for user experience.

Unfortunately the packages for using python in C# raised some security risks by needing BinaryFormatterDeserialization. The installation of python packages did not work as expected and packages had to be added manually. Another issue was that it took longer to execute python packages from C# code than executing them directly.

So the final approach was developed where a flask API is used for starting the chatbot and also to retrieve context from the vector store. Vector store creation will work with own python scripts that avoid the use and installation of langflow in production environment.

1.5 Final implementation

1.5.1 Overview

Finally, a solution for the offline chatbot was found. For demonstration purpose a sample app was created. The sample app has a dog and a bird page, each containing a ”Start chatbot” button. Accordingly a dogs and a birds vector store were created. On button click the startbot endpoint of the Flask API is called and the page/vector store name are passed to the API. The sent name has to be the same name of the vector store.

The Flask API triggers the chatbot app and passes the vector store name to the WPF window. It also includes a query endpoint to retrieve relevant context from the vector store.

The WPF application with the chatbot window uses OllamaSharp and Ollama. It sends the query to the Flask API and retrieves the context.

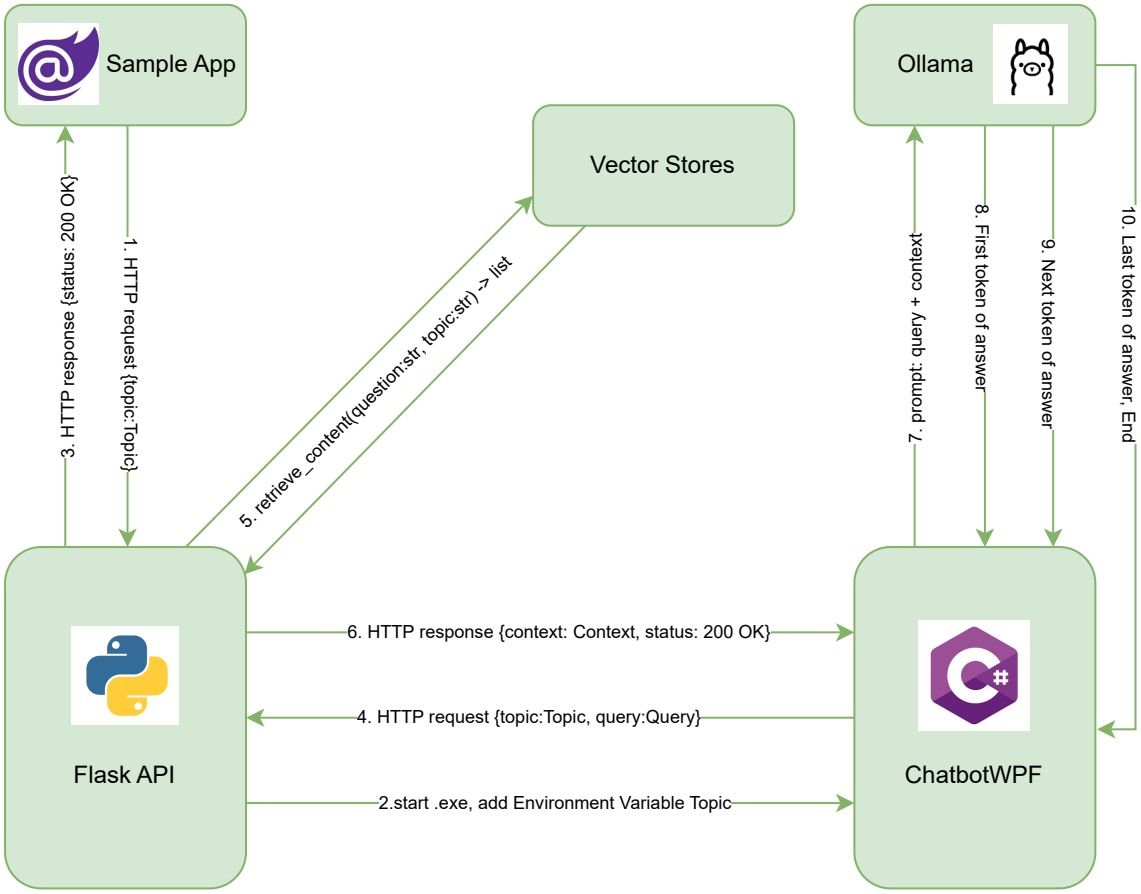

Figure 1: Overview

The code parts for the Final implementation are stored in the Final_Project folder in the repository.

1.5.2 Vector store creation

For this project the sample vector stores are provided. Creating the vector stores is not part of the production environment. Still it is an important step that will be documented here.

After different approaches, the decision was made to write own python scripts for vector store creation. The packages used in the final version are hnswlib and sentence transformers. For extracting text from different file types bs4 and pdfplumber are used. For transforming text to vectors and vice versa the embedding model ”all-mpnet-base-v2” was chosen. It is multilingual, supporting English and German, and not too large. The choice was made by comparing output, size and speed of different models.

In a first step the extracted text is cut into chunks that are then transformed into vectors. The chunk size and other parameters in indexing.py help to manipulate the output. This example refers to the use of hnswlib package:

- chunk size:
  Number of characters in a chunk that will then become a vector
- overlap:
  Number of characters that are in the previous chunk as well.
  Example: chunk size = 100, overlap = 20
  First chunk includes character 0-99, second chunk includes character 80-179
- M:
  can be a value from 12-48 and represents the maximum number of edges per node.
  higher M means more accuracy but needs more hardware resources
- ef:
  Controls the size of the dynamic list of candidates during the search process.
  higher ef means more accuracy but nees more hardware resources

To create a vector store according to your needs follow the steps in the README_createVS.

1.5.3 Sample app

The sample app is a minimalistic Blazor app that includes a ”Dog” and a ”Bird” page. Each page has a start chatbot button that calls a Flask API and passes a value. The passed value has to be the name of the vector store to be used. In this example the ”Bird” page passes ”Birds” and the ”Dog” page passes ”Dogs”.

This sample app can be replaced by any other app from which it is possible to send a http post request.

1.5.4 Flask API

Caused by the fact that the use of Python.Included in C# did not work so well, the choice was to set up a flask API with functions to trigger the chatbot and retrieve the relevant context from the vector store based on the query.

The flask API has two endpoints:

- /startbot
  - is called by the sample app that sends the topic in a post request
  - starts the chatbot.exe and sets the topic as an environment variable in the chatbot app.
- /query
  - is called by the chatbot app that sends the user input as query and the value of the environment variable as topic in a post request.
  - calls the retrieve_context function that returns relevant context from the vector store based on the query.
  - passes the context as a string to the chatbot app.

1.5.4.1 Extract context form vector store

With help of a python script the relevant context is extracted from the vector store. The query is converted to a vector using the same model that was used for the creation of the vector store. The model will compare vectors inside the vector store to the vector created from the query. Vectors that match best are identified. The found vectors are transformed into text.

The ”k” value in context_provider.py defines the number of results to retrieve.

1.5.5 Chatbot app

The chatbot is implemented as a C# WPF application. The decision was made by the concept that specifies to implement a C# application and the fact that the position and size of the WPF application can be set easily.

The chatbot app uses the OllamaSharp nuget package, which provides predefined functions to interact with Ollama. Ollama simplifies the use of large language models and helps to increase performance on machines with limited hardware resources. A more detailed description can be found in section *1.5.6.1 Ollama* on page 10. For this project OllamaSharp is used to set the large language model (LLM) and to execute the predefinded chat function. The streaming function, that enables retrieving the answer token by token is very important. As the performance of LLMs can be very slow for local use (depending on the hardware) streaming helps to avoid frustration due to long waiting time on user side.

The context is created by a http post request to the flask API passing the query and the topic. This context is then part of the prompt that is passed to the LLM, that answers the query based on the retrieved context. The prompt will also contain the chat history, but not the context of the previous queries to avoid an unneccessary overhead of text that the LLM has to process.

The user interface’s design is very simple. It has an input field, a list box, and a send button. The user writes the question into the input field and clicks the send button. The listbox shows the chat history including the user input.

Language detection is covered by an instruction in the prompt that tells the LLM to use the language of the latest query. With the use of a multilingual LLM this is all there is to do to retrieve the answer in the prefered laguage.

1.5.5.1 Choice of Large Language Model

It can be very challenging to find a large language model that fulfills the specific needs. In this project it was important to find a model that can be used with limited hardware resources and supports English and German. It has to be pretrained so that it offers good language skills. There was a lot of research and trying out different models. For testing purpose the use of langflow was very helpful. Without coding and changing scripts all the time the different models could be tested.

In the end it was decided to use llama3.2:3b that works on most machines and supports multiple languages. It has the capability to build reasonable sentences in English and German.

If the model is still too large, llama3.2:1b can be used as an alternative. The language skills are not as good as in the 3b model, but the quality is acceptable. Sometimes there is a mixture of English and German in the output and some grammar issues.

1.5.6 Tools and models

1.5.6.1 Ollama

Ollama helps to manage large language models and provides a models, a chat and an embedding endpoint in an API that runs on localhost. It is available for Windows, Linux and MacOS. Ollama improves the usage of hardware by detecting gpu if available and carrying out tasks on available resources. It supports the upload and download of LLMs and includes a simple console chat function.

The usage of Ollama is quite simple. It has to be downloaded and installed. To activate it is necessary to open a terminal and enter the command ”ollama serve”.

License Information

MIT LICENSE

- free for commercial use
- add whole license text to LICENSE.txt in project root ✓
- add notice to NOTICE.txt in project root ✓

1.5.6.2 OllamaSharp

OllamaSharp is a NuGet package that offers an easy way to interact with the Ollama API. It supports streaming responses, which reduces waiting time for responses and so improves user experience.

License Information

MIT LICENSE

- free for commercial use
- add whole license text to LICENSE.txt in project root ✓
- add notice to NOTICE.txt in project root ✓

1.5.6.3 hnswlib

Hnswlib stands for Hierarchical Navigable Small World library. It is designed for approximate nearest neighbour (ANN) search in the vector store.

License Information

APACHE LICENSE VERSION 2.0

- free for commercial use
- add whole license text to LICENSE.txt in project root ✓
- add notice to NOTICE.txt in project root ✓
- add to Acknowldegements in README ✓
- add comment in scripts that use the product ✓

1.5.6.4 Flask

Flask is a lightweight Web Server Gateway Interface (WSGI) framework. Using flask in python it is easy to set up an API.

License Information

BSD 3-CLAUSE LICENSE

- free for commercial use
- add whole license text to LICENSE.txt in project root ✓
- add notice to NOTICE.txt in project root ✓

1.5.6.5 pdfplumber

pdfplumber is a python package that helps extract text from pdf files.

License Information

MIT LICENSE

- free for commercial use
- add whole license text to LICENSE.txt in project root ✓
- add notice to NOTICE.txt in project root ✓

1.5.6.6 beautifulsoup4

Beautifulsoup4 is a python package that helps extract text from html documents.

License Information

MIT LICENSE

- free for commercial use
- add whole license text to LICENSE.txt in project root ✓
- add notice to NOTICE.txt in project root ✓

1.5.6.7 sentence transformers

Sentence transformers provides image and text embedding models as well as embedding functions.

License Information

APACHE LICENSE VERSION 2.0

- free for commercial use
- add whole license text to LICENSE.txt in project root ✓
- add notice to NOTICE.txt in project root ✓
- add to Acknowldegements in README ✓
- add comment in scripts that use the product ✓

1.5.6.8 llama3.2:3b

Llama3.2 is a multilingual large language model that supports English as well as German. It trade-off between response quality and required hardware resources is good compared to other models.

License Information

LLAMA 3.2 COMMUNITY LICENSE AGREEMENT

- free for commercial use up to 700 million active users per month
- add whole license text to LICENSE.txt in project root ✓
- add notice with copyright to NOTICE.txt in project root ✓
- add to Acknowldegements in README ✓
- ”Built with Llama” has to be added visibly for end users ✓

1.5.6.9 all-mpnet-base-v2

all-mpnet-base-v2 is a mutltilingual embedding model that supports English and German. It converts text into vectors and vice versa and compares vectors.

License Information

APACHE LICENSE VERSION 2.0

- free for commercial use
- add whole license text to LICENSE.txt in project root ✓
- add notice to NOTICE.txt in project root ✓
- add to Acknowldegements in README ✓
- add comment in scripts that use the model ✓

1.6 Known Problems

1.6.1 Starting APIs

To use the project Ollama and Flask API have to be running. In the existing code so far there is no check for running APIs and no function to start them if not.

1.6.2 Multiple open chatbots

Multiple chatbots can be opened. When they are triggerd from different pages this can lead to confusion.

Chatbots do not automatically close when a page in the sample application is closed.

1.6.3 Not ready for production

A lot of installation steps and some adaption are necessary to use the chatbot

- install Ollama
- install Python
- store vector store
- create at least one python environment
- adapt paths in script to create vector store
- store chatbot.exe, adapt path in Flask API /startbot
- adapt app from which chatbot is started (add http request, set topic according to vector store name)

1.7 Preview of Bachelor Thesis

1.7.1 Title

Optimizing Performance of Local AI Applications - Enhancing Offline Functionality Based on Hardware Constraints

1.7.2 Problem

The use of artificial intelligence solutions is increasing and users expect to be provided with applications like chatbots as state-of-the-art tools. This also applies for use cases where no internet connection is available. It is not a problem to download large language models and use them offline in general. The challenge is to find a match that provides the required functionality, and can be processed with the existing hardware. There are thousands of LLMs available. The number is increasing day by day.

In most cases it is the goal to get some specific information from an LLM, for example when used in a chatbot. There are LLMs with a huge knowledge base that might be able to do that without any additional implementation. The problem is that those models are too large for most machines. If they can manage to start at all, it would take hours to get a response.

1.7.3 Plan

A good solution for offline AI applications is to use Retrieval Augmented Generation (RAG). With RAG text is converted into vectors, which in fact are numbers. Computers capabilities are based on numbers. A computer does not understand words or speech. It can make it look like that by transforming words into numbers, numbers into a row of the two conditions it really knows. These can be interpreted as 0 and 1, true or false, on or off. So for the computer it is much easier to work with these vectors instead of speech.

The advantage compared to fine tuning a model is that the vector store can be replaced to add or remove knowledge. In addition the LLM will look for information in the vector store before using knowledge from training data. New added training data from a fine tuning process is just part of the whole knowledge base ususally including billions of parameters.

Pretrained AI models come with varying knowledge and skills. Based on the training method the output of models with same size can vary in quality. So the impact of the used training mehtod on the model's performance is something to take a closer look at.

The use of cpu and gpu will be part of the evaluation. Gpu is faster and more efficient than cpu, but not every gpu can be used for performing AI tasks.

1.7.4 Result

As a result there will be a summary of criteria to take into account when it comes to implementing an offline AI application. It will be categorized in minimal hardware requirements, best trade-of and best performance that is realistic on a common pc.

The result of this thesis can be used as a guidline to develop offline AI applications that fit for the hardware used.

1.8 Links

1.8.1 Tools

- **GPT4All** by nomic.ai: https://www.nomic.ai/gpt4all
- **JAN** by jan.ai: https://jan.ai
- **Langflow**: https://www.langflow.org
- **Ollama**: https://ollama.com
- **OllamaSharp**: https://www.nuget.org/packages/OllamaSharp/2.0.11
- **hnswlib**: https://github.com/nmslib/hnswlib
- **Flask**: https://flask.palletsprojects.com/en/stable
- **sentence transformers**: https://www.sbert.net

1.8.2 AI Models

- **llama3.2** with Ollama: https://ollama.com/library/llama3.2
- **sentence-transformers/all-mpnet-base-v2** from huggingface: https://huggingface.co/sentence-transformers/all-mpnet-base-v2