

Omdena-Lagos-Ng Chatbot Development

Project Overview

To make navigation on Omdena's website seamless and access to updated information easier, this project was conceived.

The goals for this project are to:

- develop and implement a chatbot feature on the Omdena website to provide efficient and interactive support.
- train the chatbot to effectively respond to user queries and provide accurate and relevant information.
- improve user engagement and satisfaction by enhancing the website's information dissemination process through the chatbot.
- enhance the overall user experience on Omdena's website by providing a seamless and efficient information retrieval system through the chatbot.
- establish the chatbot as a valuable tool for both new and existing users, contributing to the growth and success of Omdena's initiatives.

Technologies Used

- LucidChart - a free collaborative tool used to create wireframes. We used it to create mockups of the chatbot's dialogue flow/ interface.
- Google Sheets - to create stories/ intents for each category of the dialogue flow. Afterwards, these files were converted into .yaml files for use in RASA.
- Python (RASA 3.x) - tool used to create our chatbot model. After comparison with other tools, we chose RASA for its customizability.
- Streamlit - for deploying the model's front-end.
- Hugging Face - for deploying the model's back-end.
- Github.
- Open AI LLM
- FAISS (Facebook AI Similarity Search)

Architecture Design (Dialogue Flows and stories developed)

Architecture design in Rasa involves defining intents, entities, and designing coherent dialogue flows and stories. These stories serve as templates for conversation paths, guiding how the bot responds to user inputs. You can employ rule-based or machine learning-based dialogue management and incorporate error handling.

The stories and rules that we have created are hosted here along with the trained Rasa model:

[omdena-lc/omdena-ng-lagos-chatbot-model](https://github.com/omdena-lc/omdena-ng-lagos-chatbot-model)

These are the sections for which we've crafted engaging narratives, enabling the chatbot to effectively address inquiries pertaining to:

- About Omdena
- Collaborator Career Path
- Local Chapters
- AI Innovation Challenge
- Top Talent Program
- Omdena School
- Startup
- Universities
- General FAQs

We've meticulously structured these sections within RASA dialogue flows and stories. In terms of our fallback mechanism, it seamlessly incorporates OpenAI integration to provide responses when needed.

We've chosen to depart from using flow charts in the stories.yml file because the bot's behavior hasn't met our expectations. Consequently, we've decided to streamline our approach by incorporating only a single question per story for improved clarity and control.

Actions and Helper Functions Overview

Rasa actions are Python classes in the Rasa conversational AI framework responsible for generating dynamic responses and performing custom logic in chatbots

The actions and scripts mentioned in the following sections are hosted here:-

[omdena-lc/omdena-ng-lagos-chatbot-actions-server](https://github.com/omdena-lc/omdena-ng-lagos-chatbot-actions-server)

Here we have used 3 types of actions with distinct functionalities. Group 1 actions enable follow-up questions, Group 2 actions retrieve context and interact with OpenAI, and Group 3 actions reset slots in the conversation.

Group 1: Actions to Enable Follow-up Questions

- **ActionJoinClassify:** Classifies user intent and responds accordingly. Enables follow-up questions related to joining a local chapter.
- **ActionEligibilityClassify:** Classifies user intent and responds accordingly. Enables follow-up questions related to eligibility for local chapters.
- **ActionCostClassify:** Classifies user intent and responds accordingly. Enables follow-up questions related to the cost of joining local chapters.
- **GeneralHelp:** Provides help based on the user's role. Offers follow-up questions based on the user's response.

Group 2: Actions to Retrieve Context and Call OpenAI

- **GetOpenAIResponse:** Uses the OpenAI GPT-3.5 Turbo model to generate context-aware responses based on user queries and conversation history.
 - **Extract Conversation Data:** The action first extracts the conversation history, which includes both user queries and bot responses. It separates user queries from the conversation history and stores them in the `user_queries` list.
 - **Search Existing FAISS Vector Library:** The user query is then used to search an existing FAISS vector library for matches, returning the top 5 answers for the user query.
 - **Context Building:** The answers are joined into a single string, and the first 2014 characters are stored in the variable "context."
 - **Generate Bot Response:** The latest 3 user queries and the context are sent to OpenAI to generate a bot response.

Group 3: Actions to Reset Slots

- **ResetSlotsAction:** Resets all slots to None.

Helper Functions for actions.py

- **create_faiss_index.py:** This script is designed to create and save a Faiss index for a dataset, allowing for rapid similarity-based searches by encoding dataset entries and normalizing vector representations using a Sentence Transformer model.
- **search_content.py:** The purpose of this script is to facilitate semantic content search by encoding user queries using a pre-trained Sentence Transformer model and utilizing a Faiss index to retrieve relevant dataset entries efficiently.

1. create_faiss_index.py

- Data Loading and Preprocessing:
 - Loads and preprocesses a dataset from a CSV file (omdena_faq_training_data.csv).
 - Creates input examples by combining questions and answers into a single "QNA" column, which serves as input for the Sentence Transformer model.
- Sentence Transformer Model:
 - Loads a pre-trained Sentence Transformer model (all-distilroberta-v1) using Hugging Face Transformers.
 - Saves the loaded model to a file (all-distilroberta-v1-model.pkl) for later use.
- Faiss Index Creation and Saving:
 - Uses the loaded model to encode the "QNA" column from the dataset into vector representations.
 - Normalizes the vectors for consistent similarity calculations.
 - Creates a Faiss index with the normalized embeddings, enabling fast similarity-based searches.
 - Saves the created Faiss index to a file (index.faiss) for future retrieval and use.

2. search_content.py

- Dataset Preprocessing:
 - The script loads questions and answers dataset from a CSV file (omdena_faq_training_data.csv) and preprocesses it. It combines question and answer pairs into a single "QNA" column, which is used for search retrieval.
- Search Functionality:

- The search_content function takes a user query as input and performs the following steps:
 - Encodes the query using the Sentence Transformer model.
 - Normalizes the query vector.
 - Loads Faiss index from storage, Searches the Faiss index using the query vector, retrieving the top-k results based on similarity.
 - Extracts the IDs and similarities of the top-k results.
 - Retrieves the corresponding results from the dataset using the IDs, including questions and answers and similarities.

User Interface (UI) (Include mockups or wireframes of the chatbot's user interface and a description of the user interaction flow)

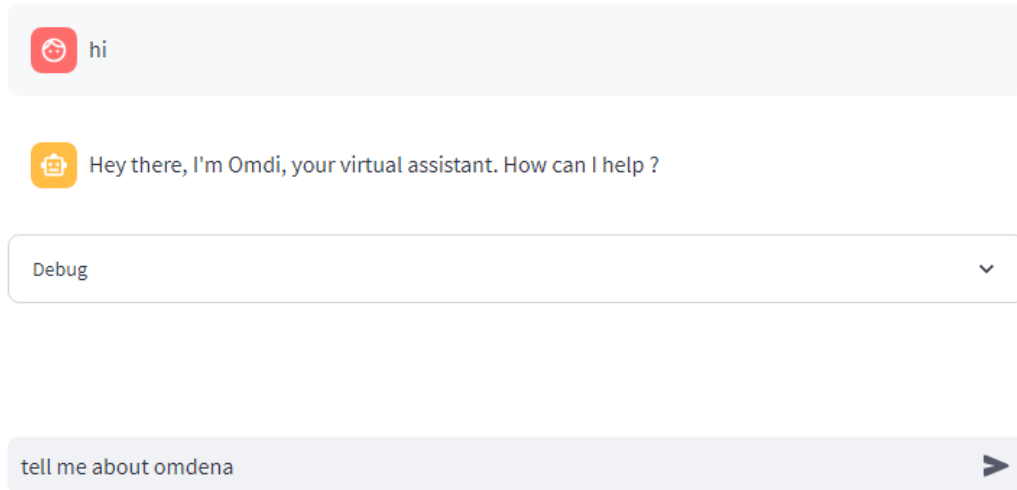
The Omdena Chatbot Interface is a web-based Streamlit application hosted on HuggingFace, designed to facilitate communication between users and the Rasa AI model. This interface allows users to interact with the chatbot, receive responses, and store conversation data in a Google Sheets document. Below is a description of the various components of the user interface (UI)

To interact with the chatbot click [here](#)

You can explore the codebase at : [omdena-lc/omdena-ng-lagos-chatbot-interface](#)

- General Interaction: The interface includes a title, chat history display and a User Input field for user messages. Users can engage in conversations with the chatbot and the chat history is displayed to keep track of the conversation. The chatbot responses are displayed with a real-time typing effect.
- API Integration with RASA server: User messages are sent to the Rasa server via a webhook and responses are received and displayed in real-time.
- Google Sheets Integration: Conversation data (user input and chatbot responses) are sent to Google Sheets using a post request to Google Apps Script. Data is recorded for improving the bot responses.
- Debug Section: An expandable section provides debugging information. Details about the Rasa AI model and its version are accessible through this section.

Omdena Chatbot Interface



The screenshot displays the Omdena Chatbot Interface. At the top, a light gray message bubble contains a red circular icon with a white robot head and the text "hi". Below this, a yellow circular icon with a white robot head is followed by the text "Hey there, I'm Omdi, your virtual assistant. How can I help ?". Underneath is a white input field with the placeholder text "Debug" and a small downward arrow on the right. At the bottom, a light gray message bubble contains the text "tell me about omdena" and a right-pointing arrow.

Data Sources (links to databases)

We collected data from different sections of the Omdena Website. This data included information from Local Chapters, AI Innovation projects, Omdena School courses, and the General FAQs section. This diverse set of sources allowed us to capture a wide range of topics and user query types.

During the data collection process, we formulated potential user queries in a Question and Answer format. For instance, we identified common questions users might have about Omdena's local chapters, AI projects, courses, or general inquiries. Each question was paired with an appropriate answer, creating a dataset of Q&A pairs.

To enhance the reliability and robustness of our chatbot, we leveraged ChatGPT. ChatGPT is a powerful language model capable of generating human-like text. We used ChatGPT to identify diverse ways users might ask questions. This involved generating variations of the same question to cover different phrasings, synonyms, and expressions that users might use.

[Omdena FAQ Data - Source 1](#)

[Omdena FAQ Data - Source 2](#)

Natural Language Processing

Techniques

Since the rise of big-data (defined by 5-Vs: Volume, Velocity, Value, Variety, Veracity), the manual-searching has become unfeasible regardless of any improvement in UI/UX. This led to the birth of Search Engines, a program that allows users to input keywords or phrases (queries) to sift through vast amounts of data and provide relevant results.

However, the search engine requires users to formulate queries in a specific format or perform actions other than querying and logging, hence the chatbots were developed. Chatbots are a dialogue-based approach to search-engine where users can query using natural language, to fetch a predefined set of responses or automate the performance of any action, using the first five reranking strategies. To generate responses they use mathematical and basic linguistic approaches to extract semantics from the input dialogue and then display the relevant results in the form of a template that is hardcoded natural language sentence where only the value of entities is variable.

The advancement in chatbots came in the form of Large Language Models (LLMs). Unlike chatbots, these dialogue-systems, are programmed using large amount of artificial neural networks (allowing them to mimic the natural language understanding and generation like that done by human brain with little or no functional competency, that is, the domain knowledge and ability to perform non-linguistic tasks they are not trained to), in order to fine-tune or augment them to retrieve domain knowledge or perform actions they could not do before.

Techniques Used

Firstly, we considered that the Omdena platform with multiple pages and role-based accesses was difficult to navigate and understand the functionality so optimizing the manual-searching UI/UX was not enough.

Next, we pondered on if a search-engine that takes role-based access as context and tabs as filters would be enough; however we realized that despite optimizing the navigation, it would still not describe the functionality of the module the user is directed to.

Then, we considered chatbots that would not only provide navigation like search-engine but also explain the modules. Since we are in the era of LLMs, we considered if and how we should use a conversational-agent to provide the formal-linguistic competence that chatbots fail at thus decreasing UX. However, we realized that the cloud-based ones are not customizable enough to

be tailored to our complex to understand modules whereas fine-tunable LLMs would be resource-wasteful and difficult to govern the responses when used alone.

Finally, we came up with an out-of-the-box approach of making the chatbot fallback on LLM, and use the contextual data stored in a vector database, to generate a creative yet relevant response.

Tools

To develop a chatbot we require a dialog-interaction framework, like BotKit, for defining interpretation and response of incoming messages, and NLP libraries or rule-defined frameworks (almost all are python-based), like RASA, Chatterbot or MindMeld, for Natural Language Processing (Understanding and Generating).

The Chatterbot works by passing the input through a preprocessing pipeline, then through a pipeline of logic-adapters, like The MathematicalEvaluation adapter that solves math problems that use basic operations, and then a pipeline that applies space-occupation reducing filters before storing them in a local-storage or a database through storage-adapters. Chatterbot is initialized with a main driver file, a logger and ability to handle concurrent and comparison-based interactions. It can be easily deployed on Django.

RASA works in a different way that separates the Natural Language Understanding engine (RASA NLU) from the Dialogue Management engine (RASA Core) in a modular fashion, allowing developers to integrate its own NLU engine with any other Dialogue Management engine, like BotKit, and vice-versa, that is, RASA Core with any other NLP program deployed on an API.

Mindmeld is structured similar to RASA but has a different workflow sequence, that is:

- 1 Select the right use case of the five use-cases
- 2 Script own ideal dialogue interactions
- 3 Define the domain, intent, entity, and role hierarchy
- 4 Define the dialogue state handlers
- 5 Create the question answerer knowledge base
- 6 Generate representative training data
- 7 Train the natural language processing classifiers
- 8 Implement the language parser
- 9 Optimize question answering performance
- 10 Deploy trained models to production

However, unlike RASA, Mindmeld does not allow developers modular access to its NLU and Dialogue Management Engine nor has the interactive training capability (for training the bot by creating two-way communications by developer taking the role of both user and the bot).

Tools Used

Firstly, based on the data collected and the conversation flow designed, it became evident that we would need a framework that provides predefined entity and intent classification techniques since the entities provide context to the same intents that lead to a different, hence Chatterbot would be resource (time and compute) intensive to develop.

Next, we analyzed the depth of customization our chatbot would need, and realized that in order to achieve our goals of integrating and triggering external services based on the intent extracted, Mindmeld became an unfeasible option due to its lack of customization capability.

In addition, we wanted to solve the issue of lack of creative understanding and response generation that chatbots suffer from; however it would be uneconomic to use a Large Language Model alone, even if open-source.

Finally, we decided to use RASA to create a chatbot that falls-back on a GPT-3 API integrated with it when the chatbot is unable to understand or generate a response, with the Dockerized model deployed as server and the Q/A interface built in Streamlit as separate Hugging-Face repositories.

Sustainability and Scalability

Sustainability:

Content Maintenance: One of the key sustainability challenges for a chatbot project like this is keeping the information up-to-date. Omdena should establish a process for regularly updating the chatbot's knowledge base to ensure that users receive accurate and relevant information. This may require dedicated personnel or automated content curation systems.

User Feedback Loop: Implement a feedback mechanism that allows users to report inaccuracies or suggest improvements. This feedback loop can be invaluable for maintaining the quality of responses and addressing user needs over time.

AI Model Updates: Stay updated with advancements in AI and NLP technologies. Periodically assess the chatbot's AI model (e.g., RASA) and consider upgrading to newer versions or integrating more advanced models to improve response quality.

Monitoring and Analytics: Continuously monitor the chatbot's performance using analytics tools. Track user interactions, satisfaction, and conversion rates to identify areas for improvement and gauge the chatbot's impact on Omdena's goals.

Scalability of Hosting: Ensure that the infrastructure supporting the chatbot can handle increased usage as Omdena grows. Scalability is crucial to accommodate a larger user base without compromising performance.

Scalability:

Multi-Lingual Support: As Omdena expands its global reach, consider adding support for multiple languages to make the chatbot accessible to a broader audience. This might involve training the chatbot in additional languages and adapting the interface.

Additional Use Cases: Explore opportunities to extend the chatbot's functionality beyond answering FAQs. For example, it could assist with onboarding new collaborators, provide updates on ongoing projects, or help with event registration. Each additional use case should be carefully designed and integrated into the chatbot's architecture.

Integration with External Systems: Consider integrating the chatbot with other Omdena systems and databases to provide more personalized and context-aware responses. For instance, it could pull real-time project updates or collaborate with other AI-driven tools.

AI Model Enhancements: Invest in improving the AI model's capabilities. This might involve training the model on a larger dataset, fine-tuning for specific domains, or experimenting with advanced AI models that can handle complex user queries better.

Deployment on Multiple Platforms: Make the chatbot available on various platforms, such as social media, messaging apps, and voice assistants, to meet users where they are. This expands the chatbot's reach and accessibility.

User Onboarding: Develop strategies for efficiently onboarding new users to the chatbot. As Omdena attracts more users, ensuring that they can easily engage with the chatbot and understand its capabilities becomes essential.

Community Engagement: Encourage Omdena's community to contribute to the chatbot's development by providing training data, suggesting improvements, or building new features. Crowdsourced efforts can accelerate scalability.

Security and Privacy: As the chatbot scales, maintain a strong focus on security and user privacy. Implement robust security measures to protect user data and maintain compliance with relevant data protection regulations.

Appendix - Glossary