**Nishauri-AI-Chatbot Documentation**

**Introduction and Overview**

The aim of this project was to develop an AI powered chatbot that integrates with WhatsApp and other MHealth solutions (e.g., the Nishauri app) to provide patients with accurate and reliable clinical information, answer questions, and provide guidance on when to seek medical attention.

**Technologies used and Data source**  
The core technologies utilized in the development of our chatbot include:

1. Rasa Open-Source Framework: We leveraged the power of the Rasa Open-Source framework to build a robust conversational AI system. Rasa provided us with a comprehensive set of tools and libraries for natural language processing, dialogue management, and intent recognition, enabling our chatbot to understand user queries accurately and give relevant responses.
2. Twilio: We used Twilio's messaging API to seamlessly integrate our chatbot with WhatsApp.
3. Python: Python served as the primary programming language for developing the chatbot's underlying models and overall functionality.
4. HTML, CSS, and JavaScript: The development of the chatbot's web user interface was facilitated by the use of HTML, CSS, and JavaScript. These web technologies allowed us to create an intuitive and visually appealing interface for users to interact with the chatbot, enhancing the overall user experience.

The data used for the chatbot developed was source from Nishauri’s app Frequently Asked Questions

**Data preparation**

To train our chatbot effectively, we carefully prepared the data by preparing three essential files required by the Rasa models. These files are critical in allowing our chatbot to understand user messages, determine their intents, and respond accurately. Let's look at each file and its function:

Intent – The idea or the goal a message is expressing

1.Nlu.yml – This file focuses on capturing the intents, which represent the ideas or goals expressed in a user's message. By providing a range of example messages and mapping them to their corresponding intents, we train our chatbot to recognize user intentions regardless of how they phrase their messages.

example:  
intent: Greetings  
examples:  
 hello

Hey

Wasssup

2.Stories.yml – This file encompasses example conversations that guide the chatbot on how to respond appropriately based on the predicted intent of the previous user message in the conversation.

example:  
intent: greeting

Action: utter\_greet

3.Rules.yml – This file outlines specific segments of conversations that should consistently follow a predetermined path, regardless of prior user messages in the conversation. Rules help enforce specific behaviors or actions based on recognized intents.

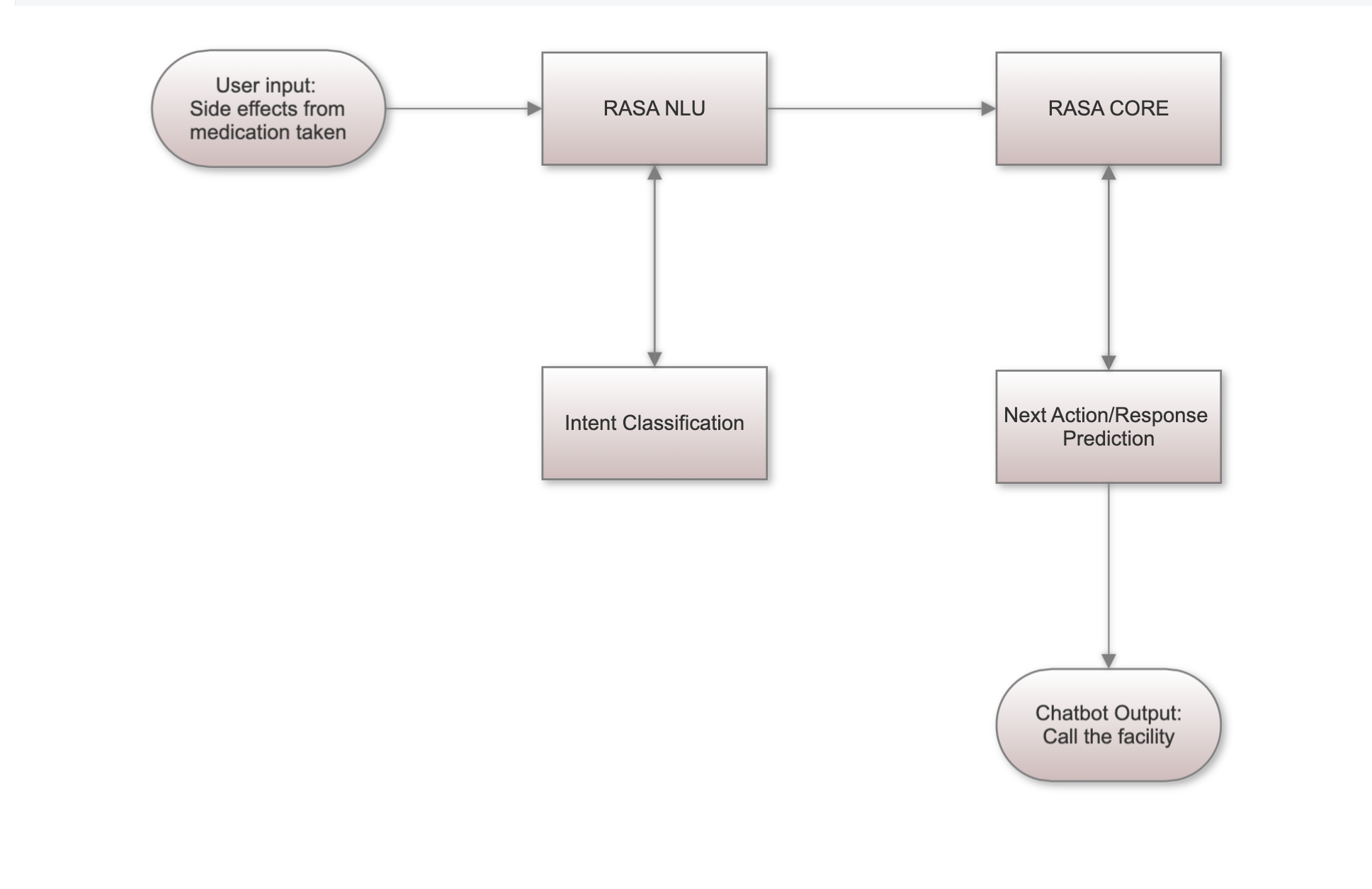
Example:

Rules:

Intent: nlu\_fallback

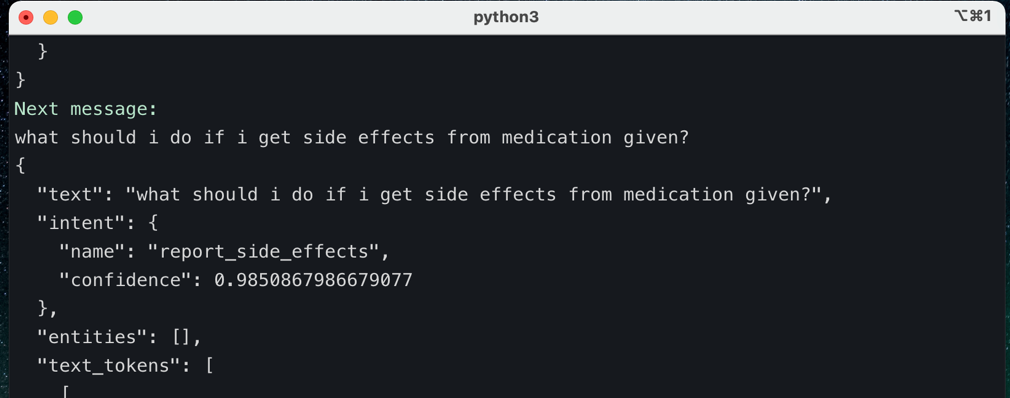
Action: utter\_please\_rephrase

**Architecture and Design**

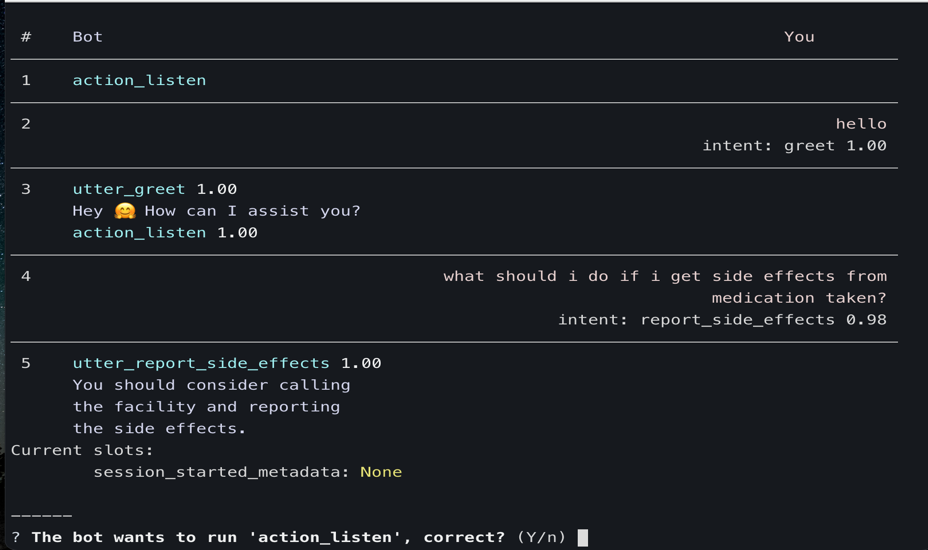


Steps:

1. The user enters a text: The interaction begins with the user entering a text message or query into the chatbot interface. This text serves as the input for the machine learning models powering the chatbot.
2. Rasa NLU model: The user's text input is then passed through the first model, Rasa NLU (Natural Language Understanding). Rasa NLU is responsible for analyzing and classifying the text into its corresponding intent. i.e., the intent with the highest probability is selected



1. Intent identification: Once the correct intent has been determined by Rasa NLU, the next step is to predict the appropriate action or response based on this identified intent. This prediction is performed by the second model, Rasa Core. Rasa Core focuses on dialog management and ensures the chatbot's conversation flows smoothly and intelligently.
2. Action or response prediction: Leveraging the insights gathered from the identified intent, Rasa Core predicts the most suitable action or response to be delivered by the chatbot.

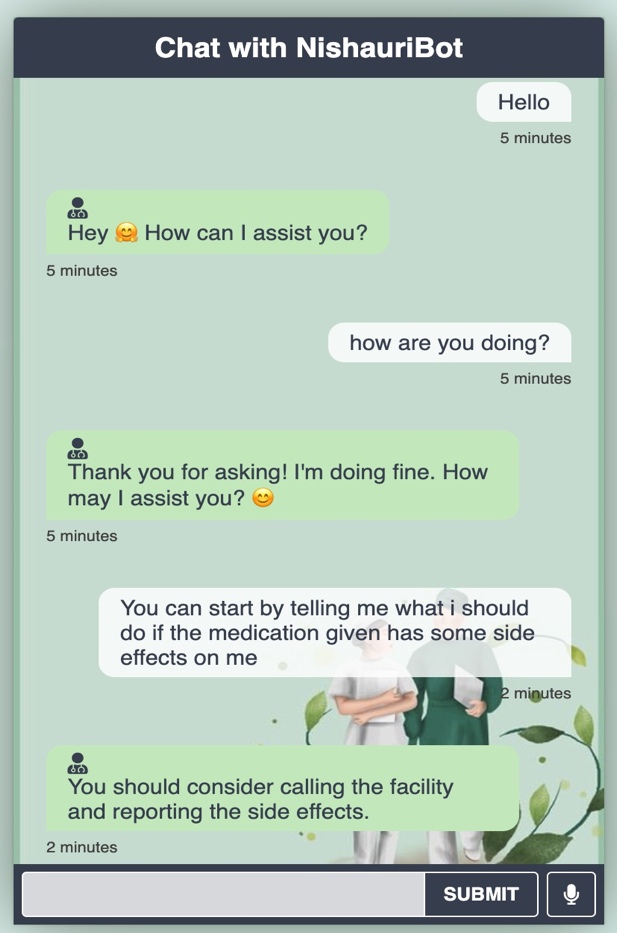
* 

1. Output sent to connected channels: Once the action or response has been determined, the output is then sent back to the connected channels. These channels can include messaging platforms like WhatsApp, Facebook Messenger, Telegram or any other designated channel where users interact with the chatbot e.g., web interfaces.

**Reasons for this Architecture**

1. Complete control of user data and privacy: By adopting this architecture, we maintain full control over user data. This means we don't have to share sensitive user information with any third-party services or platforms.
2. Flexibility with machine learning algorithms: This architecture provides us with the freedom to experiment with different versions of machine learning algorithms that drive our chatbot. We can explore various models, train them on our specific data, and optimize them to suit our unique requirements. This flexibility allows us to tailor the chatbot's performance and behavior to best meet the needs and preferences of our users.
3. Rasa as an open-source tool: Rasa, being an open-source framework, offers a vibrant community, and continuous advancements in natural language understanding and dialog management.

User Interface and Interaction



**Conclusions**

1. Enhanced accessibility: The chatbot successfully expanded its reach by connecting to multiple channels simultaneously. This multi-channel integration enables a wider user base to access the chatbot's services.
2. The Improved user experience and engagement: The implementation of a text-driven assistant has significantly enhanced the user experience.
3. Data privacy: The architecture of the solution guarantees the privacy of user data. With complete control over the data and the absence of involvement with third-party services, users can trust that their sensitive information remains secure and confidential.
4. Accurate information provision: The chatbot successfully addresses users' queries by leveraging the trained data.

* **Recommendations**

1. The Acquire more data: To improve the chatbot's accuracy and responsiveness, it is recommended to acquire more diverse and relevant data. Expanding the training dataset will enable the chatbot to handle a wider range of user queries and scenarios effectively.
2. Enhance handling of new instances and failure: The chatbot can benefit from further refinement in how it handles new instances and failure cases. By continuously updating and retraining the machine learning models, the chatbot can adapt to emerging intents and improve its ability to handle previously unseen user queries. Additionally, implementing robust error handling mechanisms and fallback strategies will ensure graceful handling of unexpected or ambiguous user inputs, enhancing the chatbot's overall performance and user experience.