# AI HELPDESK SYSTEM USING RASA

## ABSTRACT

The project aims to develop an AI helpdesk system (chatbot) which can answer to the user's questions or queries with the available knowledge. The chatbot will be designed to provide quick and accurate responses to common questions and issues faced by users, such as technical support, customer support, HR support, and more. The system will be developed using natural language understanding (NLU) framework called RASA which can understand and respond to user queries in a conversational manner using natural language processing (NLP) and advance machine learning techniques. The chatbot can be integrated into an existing platform or website and will be able to access a knowledge base containing information relevant to the user's queries. The project will involve the design and development of the chatbot's user interface, the development of knowledge for the chatbot and integration of all components using RASA. The end result will be an AI helpdesk system or chatbot that provides users with efficient and effective support, improving user satisfaction and reducing the workload on support staff.

## KEY OBJECTIVES

1. Design a user interface that is user-friendly, allows users to easily interact with the chatbot.
2. Develop a backend system using RASA that can respond to that query accordingly by using the available knowledge.
3. Improve the chatbot's ability to accurately interpret user queries and answer with most relevant information by configuring the NLU pipeline of RASA according to our use case.
4. Create a knowledge base that contains all the information needed by the chatbot to provide helpful and informative answers.
5. Integrate the frontend of the chatbot with the backend developed to answer the query through API.
6. Continuously update the knowledge base to ensure that the chatbot remains relevant and up-to-date.

## PROPOSED METHODOLOGY

### What is RASA ?

Rasa is an open-source framework for building conversational AI chatbots, assistants, and contextual assistants. It provides a complete development environment to build, train, and deploy chatbots that can understand natural language and carry out complex tasks.

**The Rasa framework consists of two main components:**

1. **Rasa NLU (Natural Language Understanding):** The Rasa NLU component uses machine learning models to recognize intents and extract entities from user inputs. These models can be trained on annotated training data to improve their accuracy. Rasa NLU also allows developers to define custom actions, which can be used to perform tasks based on the user's input.

2. **Rasa Core:** It is responsible for managing the conversation flow and deciding how the chatbot should respond to user inputs. It uses a machine learning-based approach to learn from previous conversations and predict the most appropriate action to take based on the user's intent and the current context.

Rasa also provides tools for testing and evaluating chatbots, as well as deploying them on various platforms such as Facebook Messenger, Slack, and Telegram. It also has a growing community of developers and users who contribute to the development of the framework and provide support to fellow users.

### Different files in Rasa:

**nlu.yml**: This file contains the NLU (natural language understanding) data for your chatbot. It includes the training examples for each intent, as well as any entity labels and synonyms.

```
## intent:greet
- hey
- hello
- hi
- good morning
- good evening
- hey there

## intent:bot_challenge
- are you a bot?
- are you a human?
- am I talking to a bot?
- am I talking to a human?

## intent:corona_intro
- What is corona virus
- what is covid
- what is a novel corona virus
- what is covid-19
- tell me about corona
- can you tell me about covid
```

**stories.yml**: This file contains the stories for your chatbot, which are the examples of conversations between the user and the chatbot. Each story is a sequence of actions that the chatbot takes in response to user input.

```
## say greet
* greet
  - utter_greet

## say goodbye
* goodbye
  - utter_goodbye

## bot challenge
* bot_challenge
  - utter_iamabot

## what is corona
* corona_intro
  - utter_corona_intro
```

**domain.yml**: This file contains the domain of your chatbot, including the intents, entities, actions, slots, and responses. It defines what your chatbot can do and how it should respond to user input.

```yaml
responses:
  utter_greet:
  - text: Hey! How are you?
  utter_did_that_help:
  - text: Did that help you?
  utter_goodbye:
  - text: Bye
  utter_iamabot:
  - text: I am a bot, powered by Rasa.
  utter_corona_intro:
  - text: Coronaviruses are a group of related viruses that cause diseases in mammals
      and birds. In humans, coronaviruses cause respiratory tract infections that
      can be mild, such as some cases of the common cold (among other possible causes,
      predominantly rhinoviruses), and others that can be lethal, such as SARS, MERS,
      and COVID-19
```

**config.yml**: This file contains the configuration for your chatbot, including the language, pipeline, policies for processing user input and the hyperparameters for training the model.

```yaml
language: en

pipeline:
# # No configuration for the NLU pipeline was provided. The following default pipeline was used to train your model.
# # If you'd like to customize it, uncomment and adjust the pipeline.
# # See https://rasa.com/docs/rasa/tuning-your-model for more information.
#   - name: WhitespaceTokenizer
#   - name: RegexFeaturizer
#   - name: LexicalSyntacticFeaturizer
#   - name: CountVectorsFeaturizer
#   - name: CountVectorsFeaturizer
#     analyzer: char_wb
#     min_ngram: 1
#     max_ngram: 4
#   - name: DIETClassifier
#     epochs: 100
#     constrain_similarities: true
#   - name: EntitySynonymMapper
#   - name: ResponseSelector
#     epochs: 100
#     constrain_similarities: true
#   - name: FallbackClassifier
#     threshold: 0.3
#     ambiguity_threshold: 0.1
```

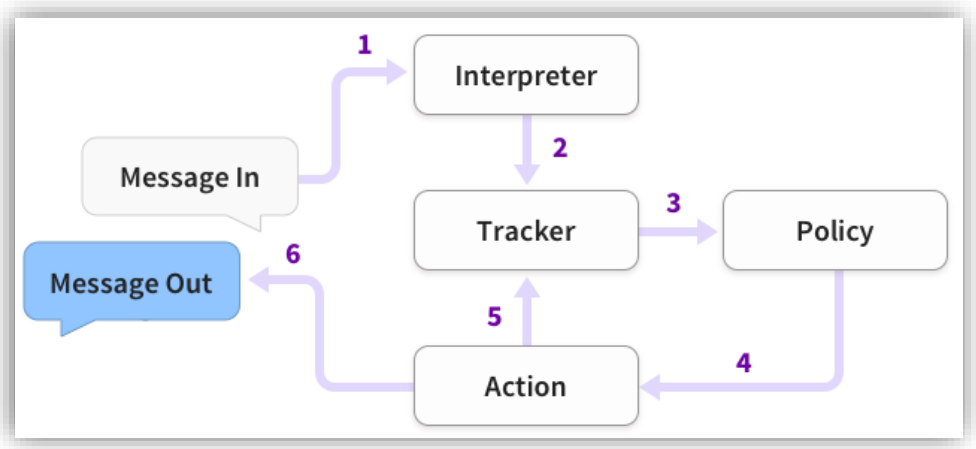
```yaml
policies:
# # No configuration for policies was provided. The following default policies were used to train your model.
# # If you'd like to customize them, uncomment and adjust the policies.
# # See https://rasa.com/docs/rasa/policies for more information.
#   - name: MemoizationPolicy
#   - name: RulePolicy
#   - name: UnexpecTEDIntentPolicy
#     max_history: 5
#     epochs: 100
#   - name: TEDPolicy
#     max_history: 5
#     epochs: 100
#     constrain_similarities: true
```

**endpoints.yml**: This file contains the endpoints for your chatbot, including the URL of the NLU server and the URL of the action server.

**actions.py**: This file contains the code for the custom actions that your chatbot can take in response to user input. It defines the logic for processing user input and generating responses.

**models:** This directory contains the trained models for your chatbot. Each model is a snapshot of your chatbot's state at a particular point in time, and can be used to run your chatbot in production.

### Rasa Architecture




The steps are:

1. The message is received and passed to an Interpreter, which converts it into a dictionary including the original text, the intent, and any entities that were found. This part is handled by NLU.
2. The message is passed from the Interpreter to the Tracker. The Tracker is the object which keeps track of conversation state and receives the structured output from interpreter.
3. The current state of the tracker is sent to each policy.
4. The policy decides which appropriate next action is to be performed.
5. The log of selected action is maintained by tracker.
6. The appropriate response is provided to the user, using intents defined in nlu.md, like utter_response.

### WHY RASA OVER OTHER FRAMEWORKS

Rasa framework is best suitable for retrieval-based chatbots for several reasons:

1. **Open source:** Rasa is an open-source framework, which means that developers can use and modify it according to their specific needs without incurring licensing fees or other costs.
2. **Customization:** Rasa provides a high degree of customization, which allows developers to create chatbots that are tailored to their specific business needs. This is particularly important for businesses that have unique requirements for their chatbot functionality.
3. **Scalability:** Rasa is highly scalable and can be used to build chatbots that can handle large volumes of traffic. This makes it a great option for businesses that need to build chatbots that can handle a high volume of customer inquiries.
4. **Natural language processing:** Rasa provides a powerful natural language processing (NLP) engine that can understand user input and respond appropriately. This allows businesses to build chatbots that can engage in human-like conversations with their customers.

5. **Integration:** Rasa integrates easily with other systems and platforms, such as messaging apps, CRMs, and customer support systems. This makes it a flexible solution that can be integrated into existing workflows and systems.

While other frameworks like opendialog, chatterbot, and Dialogflow may also have some of these features, Rasa is specifically designed for building conversational AI applications, which makes it the best choice for building sophisticated, high-performance chatbots.

## FRONTEND OF THE CHATBOT

The frontend of the chatbot is developed as dynamic web page using HTML, CSS and JavaScript. HTML will enable you to structure the content of the web page and create the different components of the chatbot interface, such as the chat window, input field, and buttons. CSS will allow you to style these components and customize the look and feel of the chatbot to match the branding of the website or platform.

JavaScript will be used to add interactivity to the chatbot, such as sending the user query to the backend through API, responding to user inputs, retrieving information from the backend, and displaying responses to the user.

## BACKEND OF THE CHATBOT

The backend of the chatbot includes implementation of RASA framework with our customized dataset.

Here is a step-by-step procedure for finding the response to the user query using RASA:

1. **User Query**: The process starts with the user entering a query or message to the chatbot.
2. **NLU Pipeline**: The message is then sent through the NLU pipeline, which decides which approaches or methods are to be used for components such as:
    - Tokenization
    - Vectorization
    - entity extraction.
    - Number of epochs
    - Fallback threshold etc

    The NLU pipeline can be customized based on use cases and better performance. The methods or approaches used in NLU pipeline are saved in **config.yml**.
3. **Intent Classification**: In this step, the intent of the user message is determined. RASA uses machine learning algorithms such as SVM (Support Vector Machines).

    The intent classification algorithm tries to match the user's message to one of the predefined intents with the help of training examples in **nlu.yml** file. The intent with the highest score is selected as the predicted intent.

    **Intent classification using support vector machine (SVM):**

    **Data preparation**: The first step is to prepare the training data and the user query for the intent classification. The dataset should have enough examples for each intent to enable the model to learn the patterns in the data.
    Tokenize the user query and training data, which means breaking it down into individual words and creating a vocabulary of all the unique words in the dataset.

Also removing of stopwords, punctuation and part-of-speech tagging on the data are performed.

**Vectorization and Feature Extraction:** Once the text data has been preprocessed Vectorization is the process of converting text into numerical vectors that can be used in machine learning models. In RASA, the CountVectorizer and TfidfVectorizer classes from the scikit-learn library are used for vectorization. The CountVectorizer converts the text into a matrix of token counts, while the TfidfVectorizer converts the text into a matrix of term frequencies multiplied by inverse document frequencies (TF-IDF).

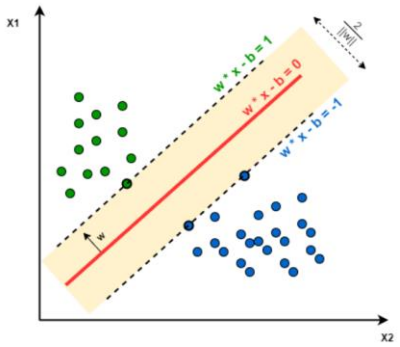
After the data is vectorized, feature extraction is the process of selecting and transforming the most relevant information from the vectorized data to be used for model training. In the context of intent classification in Rasa, feature extraction typically involves transforming the vectorized text data into a more compact representation that captures the most important information for distinguishing between different intents.

The CountVectorsFeaturizer component is a feature extractor that uses the bag of words (BoW) model to represent each input message as a vector of word frequencies. In BoW representation, each word in the text data is treated as a separate feature, and the feature vector is a count of how many times each word appears in the text.
The SpacyFeaturizer component uses the spaCy library to extract linguistic features from the input message, such as part-of-speech tags and dependency parse trees. These vector representations capture the semantic meaning of each word in the text and allow RASA to compare the similarity between different words.

This vector representation is then used as input to the support vector machine, to predict the intent of the query.

**Training the model:** Once the data is pre-processed, the next step is to train the model. SVM is a machine learning algorithm that works by finding the optimal hyperplane in a high-dimensional space that separates the different classes. In the context of intent classification, the SVM algorithm tries to find the best decision boundary between different intents.

Support Vector Machines (SVM) finds the best hyperplane by maximizing the margin between the two classes in the feature space. The margin is defined as the distance between the hyperplane and the closest data points of each class, also known as the support vectors. The goal of SVM is to find the hyperplane that maximizes this margin.



a hyperplane in a d-dimensional space can be represented as:

$$\boldsymbol{w^T\ x + b = 0}$$

where w is the normal vector to the hyperplane, x is the input data, b is the bias term, and T represents the transpose operation.

In binary classification, we have two classes (positive and negative), and we aim to find a hyperplane that maximizes the margin between the two classes. The margin is defined as the distance between the hyperplane and the closest data points from each class.

Mathematically, the margin can be expressed as:

$$\textbf{margin} = \frac{2}{||w||}$$

where ||w|| is the Euclidean norm of the weight vector w. The goal of SVM is to maximize this margin subject to the constraint that all data points are correctly classified.

For **multi-class classification** RASA uses **"one-vs-all"** approach

In this approach, SVM trains one binary classifier for each class, treating the data of that class as positive examples and the data of all other classes as negative examples.
For class i, SVM constructs a hyperplane that separates the data points of class i from the data points of all other classes. The hyperplane is defined as:

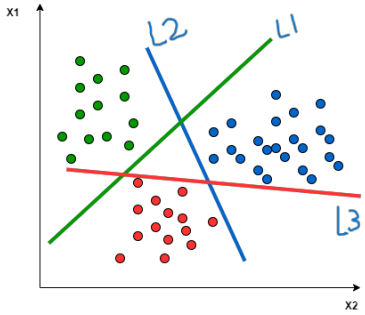
$$\boldsymbol{w_i^T x + b_i = 0}$$

where $w_i$ is the weight vector and $b_i$ is the bias term for the binary classifier that separates class i from all other classes.

To classify a new data point x, SVM runs each of the binary classifiers and computes a score for each class. The score for class i is defined as:

$$\boldsymbol{score_i = w_i^T x + b_i}$$

The class with the highest score is chosen as the predicted class.




here, the line L1 tries to maximize the gap between green points and all other points at once.

Suppose that if the confidence score is very low (close to 0), that means the user query has not matched with any pre-defined intents. We can set a threshold so that any user query which has the score less than that threshold then a default response will be triggered saying “Sorry, I didn’t understand. It is out of my knowledge boundary.”

We can handle the fallbacks in multiple ways, the above one is a standard approach. Fallbacks can also be handled by redirecting a less score user query to a human response.

4. **Entities extraction:** In this step, any entities present in the user message are extracted. Entities are specific pieces of information that the user is trying to convey.

   For example, if the user message is "Book a flight from New York to London on 10th August", then the entities extracted would be "New York", "London", and "10th August". Entities are defined in the **nlu.md** file in RASA.

   To extract entities, RASA uses a combination of rule-based and machine learning-based approaches.

   **Rule-based entity extraction:**
   In rule-based entity extraction, we define patterns for each entity we want to extract. For example, if we want to extract a date, we can define a pattern that matches dates in various formats like "today", "tomorrow", "next week", "December 1st", etc. RASA provides built-in entity extractors like DIETClassifier and CRFEntityExtractor which uses rule-based approach to extract entities. RASA by default uses a rule-based approach for entity recognition.

   **Machine learning-based entity extraction:**
   In machine learning-based entity extraction, we train a model to predict entities from text. RASA provides a pre-trained entity extractor called DIETClassifier which uses a neural network to predict entities. The neural network takes in the tokenized text and context of the conversation as input and outputs the entity label and its start and end positions in the text.

   Once the entities are extracted, they are stored in slots. These slots can be filled with user inputs.

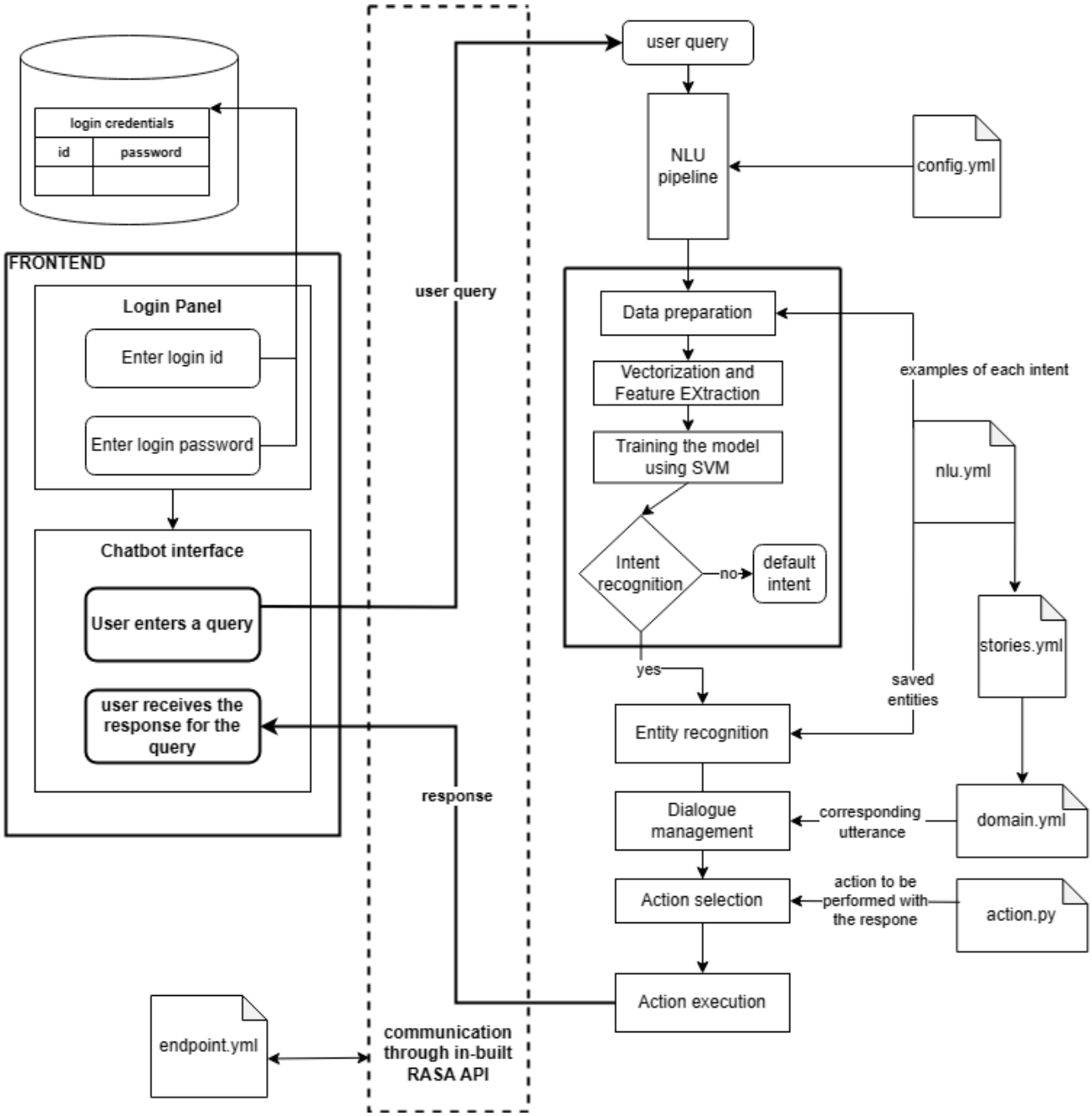
5. **Dialogue management:** Once the intent and entities of the user message have been determined, the dialogue management component of RASA decides what the chatbot should do next. It uses a dialogue policy to decide which action to take next based on the current conversation history. Dialogue policies are defined in the **config.yml** and **domain.yml** file in RASA.

6. **Action selection:** Once the dialogue management component has decided which action to take, the action server is called to execute the action. Actions are defined in the **actions.py** file in RASA. The action can be customized as per requirement and sent to the user. An action can be anything from sending a message to calling an API.

7. **Action execution:** The action server executes the chosen action and returns a response. The response is sent back to the user as a message through the Rasa NLU server API. The responses are defined in the **domain.yml** file in RASA.

8. **User receives response:** The user receives the response from the chatbot.

## ARCHITECTURE DIAGRAM



## SCOPE OF THE PROJECT

The helpdesk system can be used at organization level for providing support such as:

1. **IT support**: The chatbot can help employees troubleshoot common IT issues, such as internet related issues or software installation problems.
2. **Training and development**: The chatbot can answer employee's questions about training programs, professional development opportunities.
3. **Facilities and operations**: The chatbot can help employees find information about office locations, parking, security, or other facilities-related questions.
4. **Employee onboarding**: The chatbot can help new employees navigate the onboarding process by answering questions about company policies, benefits, and procedures.

Furthermore, the helpdesk system can be used in industries like **Banking, Education, sales and marketing, Travel and Hospitality.**

## REFERNCES

[1] Rasa Official documentation: https://rasa.com/docs/rasa/

[2] An Analytical Study and Review of open Source Chatbot framework, RASA. National Informatics Center (NIC), India.

[3] Building a Chatbot on a Closed Domain using RASA. Can Tho University, Vietnam.