DETI Pal Report

Intelligent Systems II - Conversational Agent

1st Daniel Emídio
dept. of eletectronics, telecommunications and informatics
University Of Aveiro
Aveiro, Portugal
daniel.emidio@ua.pt

2nd David Palricas
dept. of eletectronics, telecommunications and informatics
University Of Aveiro
Aveiro, Portugal
davidpalricas@ua.pt

3rd Márcio Tavares
dept. of eletectronics, telecommunications and informatics
University Of Aveiro
Aveiro, Portugal
marciotavares@ua.pt

***Abstract*—This document describes the development of a project carried out for the course Intelligent Systems II (41877), focused on building a conversational agent. Our agent was expected to process natural language in English, learn from user interactions to improve its responses, and be able to reply in a ”seemingly intelligent” manner when presented with unsupported or grammatically incorrect sentences. We decided to develop a chatbot named DETI Pal using the Rasa framework, with its main functionality being to provide information about the courses and teachers of DETI (Department of Electronics, Telecommunications and Informatics) at the University of Aveiro.**

***Index Terms*—conversational agent, natural language, chatbot, Rasa**

I. Introduction

A conversational agent, or chatbot, is an artificial intelligence (AI) based software designed to simulate a human conversation using natural language. These agents are increasingly being used in areas like customer service, healthcare, banking, and information retrieval. There are two main categories of conversational agents [1]:

- Rule-based agents, which follow pre-defined rules by matching user input to a database of responses.
- AI-powered agents, which use machine learning and natural language processing (NLP) to understand context, learn from interactions, and manage more complex conversations.

Our chatbot, DETI Pal is a chatbot that can have a normal conversation with users, provide various services and answer personal questions. It was develop using Rasa and, thanks to that, falls under both categories of conversational agents: rule-based and AI-powered. The chatbot’s primary function is to offer information about courses and teachers from the Department of Electronics, Telecommunications, and Informatics (DETI) at the University of Aveiro. However, it also includes a range of additional features, such as:

- Greeting users.
- Recognizing and responding to expressions of gratitude.
- Understanding compliments and insults.
- Politely saying goodbye.
- Answering personal questions.
- Detecting when a user is feeling sad and offering encouragement.
- Telling jokes.
- Remembering the users’ name if provided.
- Remembering users’ favourite media (e.g.., game, book).
- Answering the current time in a specified location.
- Providing points of interest in a specified location.
- Recommending TV shows and movies.

II. Rasa

Rasa was chosen as the framework for developing our conversational assistant due to its flexibility, customizability, and strong support for advanced natural language processing components. It allows full control over dialogue flow and easy integration with external services, without relying on proprietary platforms.

In this section, we present the architecture of Rasa and describe its main components: the Natural Language Understanding (NLU) module, the Dialogue Management module, and the Actions module. Together, these elements enable the assistant to manage contextual, dynamic, and personalized conversations with users.

A. Architecture

Rasa’s architecture show in Figure 1 is designed around modular components, that are combined to process user input and generate appropriate responses. The interaction with the assistant, starts when the user types its message which is received through a HTTP server responsible for sending the message to the correct internal components

The next stage begins with the **NLU (Natural Language Understanding)** module, which processes the user input to identify what the user wants (intent) and extract key details from user inputs, such as names or dates (entities).

Once the user's intent and entities have been identified, the **Dialogue Management** module takes over. Decide what the assistant should do next based on the current conversation state, the history of the interaction, and the trained dialogue policies (learned through stories and rules). This component ensures coherent, context-aware conversations that adapt to each user's behaviour. The Dialogue component can also directly access the Database to retrieve or store conversation-related information.

The **Actions** component is responsible for executing the assistant's decisions. These actions can include sending responses back to the user, calling external APIs, or triggering system events. As shown in the diagram, the Actions component can interface with External Systems to perform operations outside the RASA environment.

The architecture includes several feedback loops: the Dialogue component can communicate back to the HTTP Server directly, and both the Dialogue and Actions components can trigger further processing. Additionally, the HTTP Server can route messages directly to the Dialogue component, bypassing NLU in certain situations.

As noted in the diagram, RASA's architecture consists of three main component systems: - **RASA NLU**: Handles the natural language understanding - **RASA Core**: Manages the dialogue and conversation flow - **RASA X**: Provides tools for improving the assistant through user interaction data

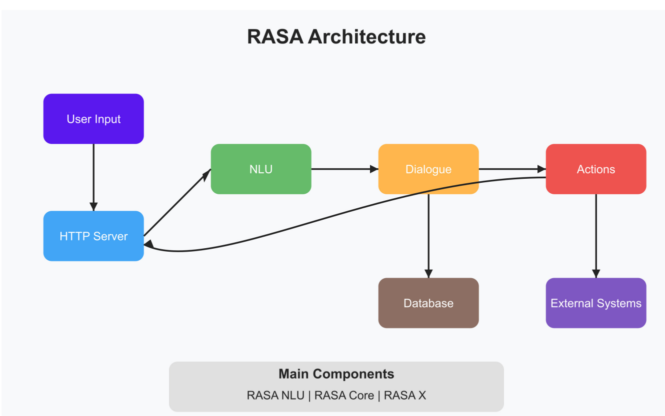

Fig. 1. Rasa Architecture

B. NLU Module

1) Intents: Intents are an important part of the NLU (Natural Language Understanding) of conversational agents. They help classify the user's goal by extracting information from their input. In Rasa, intents are defined in the NLU training file, where each intent is given a name and associated examples to train the Rasa NLP model. This model learns to recognize patterns in user inputs and matches them to the corresponding intent. In this way, it becomes easier for the conversational agent to understand what the user wants, even when there are grammatical errors or is written in a different form than the examples provided, but with a similar context.

2) Entities: Entities are another important part of NLU, as they extract key details from user inputs (e.g., locations, types, dates). Entities are associated with intents, and they are also passed in the text of an intent's examples in the format (entity_value) [entity_name]. Rasa supports multiple mechanisms for extracting entities. In our project, we use the following mechanisms:

- LookUp Table
- Regexes
- Spacy module

LookUp Tables are tables defined in the NLU files with predefined values. The entity values extracted will match a value from this table. This approach is ideal for extracting location types and genres for movies and TV shows, as the Google API and IMDB datasets have specified values for their queries.

Regex, like LookUp Tables, is defined in the NLU files. We use this mechanism to extract the professors' emails from the University of Aveiro, as they match the pattern mail@ua.pt.

SpaCy is one of the external models supported by Rasa. SpaCy is an open-source library used for advanced natural language processing. Its models contain many words in English, including special names, making it ideal for extracting location names provided by the user.

C. Dialogue Module

1) Slots: Slots are the long-term memory of the assistant in a conversation. They are used to store relevant information that may affect future interactions or personalize responses. For example, in our assistant, if the user provides their name, we store it in a slot. Later, when giving movie suggestions, the assistant can include the user's name in its response to make the interaction more personal.

Slots can be extracted in multiple ways: from intents, entities, or through custom actions. Their configuration must be defined in the domain file.

2) Rules: Rules are simple, deterministic paths that the assistant follows when a specific intent is detected. They are used for short, linear conversations where responses are always the same. The rules and their paths are defined in the rules files.

3) Stories: Stories are a type of training data used to train a Rasa assistant's dialogue management model. They allow the assistant to learn how to respond in multi-turn conversations and can help it generalize to unseen interactions. Each story must be defined in the stories files.

Stories follow a format that represents a conversation between a user and an AI assistant, where user inputs are expressed as intents (and entities when necessary), and the assistant's responses and actions are represented by action names.

Within stories, it is also possible to check whether specific slots hold certain values. This allows for more dynamic and context-aware dialogue, making the assistant's responses more relevant and personalized.

D. Actions Module

Actions are the assistant's responses or operations triggered by rules or stories. They can be either responses or custom actions.

Responses are messages sent by the assistant to the user, while custom actions are Python functions that can execute any kind of logic. Custom actions are useful when we want to generate more personalized responses or when we need to interact with external services, such as APIs or datasets.

III. Implemented Features

In this section, we describe the implementation of each feature using the Rasa framework. It is important to note that for every feature, the corresponding intents, entities, slots, responses and actions must be defined within the appropriate domain file.

A. Greeting users

The first functionality implemented in our chatbot was the ability to greet users. To achieve this, we defined a specific intent named *greet* in the file *nlu-user-name.yml*. This intent includes several examples of greeting messages, both with and without the user's name. Some of these examples are illustrated in Fig. 2.

```
nlu:
- intent: greet
  examples: |
    - hey
    - hello
    - hi
    - good morning
    - Hi, my name is [Alice](PERSON)
    - Hi, I am [John](PERSON)
    - Good morning, It's [Sarah](PERSON)
```

Fig. 2. Greet Intent

The example *Hi, my name is [Alice](PERSON)* demonstrates the extraction of an entity labeled as `PERSON`, identified using the SpaCy library. In order to enable this entity recognition, we included the *SpacyEntityExtractor* component in the NLU pipeline, configured in the *config.yml* file, as shown in Fig. 3.

```
pipeline:
  - name: SpacyNLP
    model: "en_core_web_lg"
  - name: SpacyTokenizer
  - name: SpacyFeaturizer
  - name: SpacyEntityExtractor
    dimensions: ["PERSON", "GPE"]
  - name: RegexFeaturizer
    use_lookup_tables: true
```

Fig. 3. SpacyEntityExtractor on the Rasa NLU pipeline

After configuring entity extraction, we defined a rule to determine the behavior of the chatbot whenever the *greet* intent is detected - in this case makes the custom action *action_greet*. This rule was implemented in the *rules-user-name.yml* file, as shown in Fig. 4.

```
rules:

- rule: Greet the user with or without a name
  steps:
  - intent: greet
  - action: action_greet
```

Fig. 4. Greet Rule

The implementation of the custom action `action_greet` is shown in Fig. 5. In this action, the chatbot attempts to retrieve the user's name either from the extracted *PERSON* entity or from the `user_name` slot, if it has already been set. Based on whether a name is detected, the chatbot selects an appropriate response — personalized or generic. Finally, the name is stored in the `user_name` slot for use in future interactions.

```python
class ActionGreet(Action):
    """Action to greet the user (with or without their name)."""

    def name(self):
        """Name of the action."""
        return "action_greet"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        """Greet the user with their name (if provided)."""
        # Get the user's name (entity or slot)
        user_name = next(
            tracker.get_latest_entity_values("PERSON"),
            tracker.get_slot("user_name") or None
        )

        # Select the appropriate response based on whether the name is provided or not
        response = "utter_greet_with_name" if user_name else "utter_greet_no_name"
        dispatcher.utter_message(response=response, user_name=user_name)

        # Set the slot to the user's name
        return [SlotSet("user_name", user_name)]
```

Fig. 5. Greet Action

B. Expressions of gratitude

Another feature implemented in our chatbot is the ability to recognize and respond to expressions of gratitude. To support this, we defined an intent named *thanks* in the file *nlu-compliments-insults.yml*, which includes some user inputs that convey appreciation shown in Fig. 6.

```
nlu:
- intent: thanks
  examples: |
    - ty
    - thank you
    - thanks
    - thank you so much
    - thanks for your support
```

Fig. 6. Thanks Intent

To handle this intent, we created a rule that specifies the chatbot's response whenever a user expresses gratitude. The rule triggers the predefined response `utter_thanks`, which provides a polite reply such as ”You're welcome”. This rule is defined in the *rules-compliments-insults.yml* file, as illustrated in Fig. 7.

```
rules:

- rule: Say "You're welcome" anytime the user says thanks
  steps:
  - intent: thanks
  - action: utter_thanks
```

Fig. 7. Thanks Rule

The implementation of features such as handling compliments and insults, saying goodbye, answering personal questions and detecting when a user is feeling sad follows a process similar to the one described in III-B.

C. Compliments and insults

The chatbot possess two different intents to detect and appropriately respond to both compliments and insults: something_nice and something_bad. Something_nice intent covers positive expressions and compliments, on the other hand, the something_bad intent is triggered by messages that are rude or offensive. In any case the chatbot is configured to respond in a friendly and humble manner, reinforcing a positive user experience.

```
responses:
  utter_something_nice:
  - text: "I appreciate your kind words."
  - text: "Thanks for the compliment!"
  - text: "Your words mean a lot to me."

  utter_something_bad:
  - text: "Sorry, I will do my best next time."
  - text: "I am here to help you."
  - text: "I will try to improve."

  utter_thanks:
  - text: "You're welcome!"
  - text: "No problem!"
  - text: "Anytime!"
  - text: "Glad to help!"
```

Fig. 8. Response for compliments and insults

D. Saying goodbye

The chatbot is equipped to recognize farewell expressions using the goodbye intent. This allows it to acknowledge when users are possibly leaving.

E. Answering personal questions

The DETI Pal chatbot includes functionality to handle various personal questions that users might ask during interaction. This feature helps create a more engaging conversation experience. We implemented several intents related to personal questions in the nlu-personal-questions, including:

- **ask_identity**: Recognizes questions about the chatbot's identity or name
- **ask_age**: Captures inquiries about the chatbot's age or creation date
- **ask_home**: Identifies questions about where the chatbot is from or lives
- **ask_hobby**: Detects questions about the chatbot's hobbies or interests
- **ask_favorite_color**: Recognizes questions about color preferences
- **ask_favorite_food**: Captures food preference questions
- **ask_favorite_movie**: Identifies movie preference questions
- **ask_favorite_show**: Detects TV show preference questions

Each intent is paired with a corresponding rule in the rules-personal-questions.yml file that maps to specific responses. Most intents trigger simple utterance responses (e.g., utter_ask_identity), while the ask_age intent triggers a custom action (action_give_age) that can dynamically calculate and respond with the chatbot's ”age” based on its creation date. This approach gives our chatbot a consistent personality across conversations and creates a more relatable experience for users. For example, when asked ”What is your favorite movie?”, the chatbot responds with its predefined favorite movie,that being Star Wars Episode III: Revenge of the Sith, making the interaction more natural and engaging.

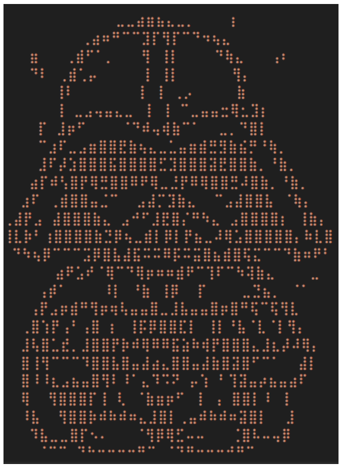

Fig. 9. Response for favorite movie

F. Detecting when a user is feeling sad

To support empathetic and emotionally aware interactions, the chatbot includes a mood_unhappy intent to recognize when users are feeling down or upset. When the chatbot detects this intent, it responds with comforting and supportive messages, the message beeing a link to a gif of a cute cat to cheer the user up.

Fig. 10. Response for sad users

G. Telling jokes

The implementation of the ”Telling Jokes” feature follows a similar approach to the one described in III-A. As with the greeting functionality, we began by defining an intent named *tell_joke* in the file *nlu-tell-joke.yml* (in contrast, this one has no entities). A rule was then added to trigger a custom action whenever this intent is detected, `action_tell_joke`, shown in Fig. 11.

```python
class ActionTellJoke(Action):
    """Action to tell a random joke."""

    def __init__(self):
        """Initialize the action and load jokes from a file."""
        super().__init__()
        self.load_jokes()

    def load_jokes(self):⋯


    def name(self):
        """Name of the action."""
        return "action_tell_joke"

    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        """Run the action to tell a random joke."""
        # Get a random joke from the list and send it to the user
        joke = random.choice(self.jokes)
        dispatcher.utter_message(joke)
        return []
```

Fig. 11. Tell Joke Action

The custom action **action_tell_joke** allows the chatbot to select and deliver jokes from a predefined collection of jokes present in jokes.txt. Using a custom action instead of a simple utterance gives us the flexibility to randomly select jokes to provide variety and track which jokes have already been told to the user.

H. Remembering users' name

An important feature to personalize the interaction between the chatbot and users is the ability to remember users' names. DETI Pal implements this functionality through a combination of entity recognition, slot storage, and custom actions. This implementation follows a process similar to the one used for greeting users (in III-A).

We created two custom actions: *ActionSetName* and *ActionShowName*.

- *ActionSetName*: This action retrieves the user's name, either from the *PERSON* entity or from the `user_name` slot and responds accordingly. To finish, it sets the `user_name` slot.
- *ActionShowName*: This action is responsible for recalling and displaying the user's name. It checks the `user_name` slot and if a name is stored, it responds with a message showing the user's name.

```
Your input ->  Say my name
Heisenberg
Your input ->  You're goddamn right!
```

Fig. 12. Response for remembering users' name

I. Remembering users' favorite media

Our chatbot can also remember the user's favorite media, such as books, movies, or music. This functionality follows a similar process of III-H.

We created two custom actions: *ActionSetFavoriteMedia* and *ActionShowFavoriteMedia*.

- *ActionSetFavoriteMedia*: This action retrieves the type and name of the user's favorite media (e.g. "game" and "Minecraft") and stores them in the *favorite_media* slot. If either the media type or name is missing, it warns the user.
- *ActionShowFavoriteMedia*: This action retrieves and displays the user's favorite media of a specified type (e.g. their favorite game).

The favorite media information is stored in the *favorite_media* slot as a JSON string.

J. Answering the current time

We incorporated the capability to provide users with current time information for different locations around the world. This practical feature demonstrates the chatbot's ability to interact with external services and provide real-time information. We utilize SpaCy's entity recognition to identify geographical political entities (GPE), such as countries, cities, and regions mentioned in the user's query. When the chatbot detects the ask_time intent, it triggers a custom action called action_get_time. This action extracts the location entity from the user's message, uses the **Nominatim** geocoding service to convert the location name into geographic coordinates and utilizes the **TimezoneFinder** library to determine the appropriate timezone based on these coordinates. Finally the **pytz** library accesses the current time in the identified timezone and returns the time information to the user.

K. Providing points of interest

Our assistant search for the 10 best (filtered by ranking) points of interest in a certain location, using the **Google Map's API** and to show to the user. To perform this feature the user must provide the type of point we has an interest and the location. The user can do that in single message or in multiple steps, to allow this a set of intents were made with also a set of stories(Figure 13). Besides the name of the point of interest, the Deti Pal provides also its address, link to google maps, rating and informing the user if its open or not.

For the assistant's response two actions were created: *ActionRetrieveLocation* and *GivePointsOfInterest*.

- *ActionRetrieveLocation*: This action retrieves the location (city or town) provided by the user and stores it in the *location* slot. If the user did not specify a location or it is invalid, DetiPal will inform the user and prompt them to provide the location again.
- *GivePointsOfInterest*: This action shows the type of point of interest the user wants in the specified location. If an error occurs, or if the location or type of point of interest is invalid, an appropriate response is shown to the user.

```yaml
stories:
- story: ask_points_of_interest_without_location_and_type_of_interest
  steps:
    - intent: ask_points_of_interest
    - action: action_retrieve_location
    - slot_was_set:
      - location: null
    - intent: provide_location
    - action: action_retrieve_location
    - slot_was_set:
      - location: not null
    - intent: choose_interest_type
    - action: action_give_points_of_interest

- story: ask_points_of_interest_with_location_and_without_type_of_interest
  steps:
    - intent: ask_points_of_interest
    - action: action_retrieve_location
    - slot_was_set:
      - location: not null
    - intent: choose_interest_type
    - action: action_give_points_of_interest

# In this story, the intent can have a location and the following action retrieves the location.
- story: ask_points_of_interest_with_location_and_type_of_interest
  steps:
    - intent: choose_interest_type
    - slot_was_set:
      - location: null
    - action: action_give_points_of_interest


- story: ask_different_points_of_interest
  steps:
    - intent: choose_interest_type
    - slot_was_set:
      - location: not null
    - action: action_give_points_of_interest
```

Fig. 13. Providing points of interest

L. Recommending TV shows and movies

The Deti Pal assistant can provide 10 randomly generated suggestions for TV shows or movies if the user provides a genre for this type of media(Figure 14). To generate these suggestions, two **IMDb** datasets are used: one to retrieve basic information (title and release year), and another to get the ratings of the media. Additionally, an API call to the **TMDb** platform is made to extract the synopsis, poster link, and platform availability. These recommendations are not only filtered by genre but also by an IMDb rating range of 8 to 10 stars. The assistant provide appropriate responses if any error such as the user didn't provide a genre or that genre doesn't exit(in **IMDB** data sets or the **TMDb** plataform).

For the assistant's response two actions were created: *ActionAskMediaGenre* and *ActionRecommendMedia*.

- *ActionAskMediaGenre*: This action is triggered if the user asks for a recommendation of a movie or a TV show without specifying the genre. DetiPal will respond with a message asking what movie genre the user is interested in if the user requested movie recommendations. Otherwise, it will ask which TV shows genre the user is interested in.
- *ActionRecommendMedia*: This action provides media recommendations to the user. If an error occurs or if the specified genre is invalid, the assistant will show an appropriate response.

```yaml
stories:
- story: ask_tv_show_suggestions_without_genre
  steps:
  - intent: ask_tv_show_suggestions
  - slot_was_set:
    - is_movie: false
    - media_genre: null
  - action: action_ask_media_genre
  - intent: provide_media_genre
  - action: action_recommend_media

- story: ask_tv_show_suggestions_with_genre
  steps:
  - intent: ask_tv_show_suggestions
  - slot_was_set:
    - is_movie: false
    - media_genre: not null
  - action: action_recommend_media

- story: ask_movie_suggestions_without_genre
  steps:
  - intent: ask_movie_suggestions
  - slot_was_set:
    - is_movie: true
    - media_genre: null
  - action: action_ask_media_genre
  - intent: provide_media_genre
  - action: action_recommend_media

- story: ask_movie_suggestions_with_genre
  steps:
  - intent: ask_movie_suggestions
  - slot_was_set:
    - is_movie: true
    - media_genre: not null
  - action: action_recommend_media
```

Fig. 14. Recommending TV shows and movies

M. Information about courses and teachers of DETI

Last but not least the functionalities to provide users with detailed information about DETI's academic offerings and faculty. DETI Pal recognizes various possible user queries related to:

- **Course Information** (course_info): Users can request details about specific courses by using course codes (e.g., "8309"), acronyms (e.g., "LECI"), or full course names (e.g., "Licenciatura em Engenharia Informática").
- **Subject Information** (disciplina_info): Users can inquire about specific subjects using subject codes, acronyms (e.g., "POO"), or full subject names (e.g., "Redes de Computadores").
- **Professor Information** (professor_info): Users can ask about faculty members by name to learn about their contact information, office location, and teaching assignments.
- **Email Lookup** (email_info): Users can inquire about the owner of a specific email address within the department.
- **Course Listing by Degree** (course_list_degree): List of all courses offered by DETI, optionally filtered by degree level (bachelor's, master's, etc.).
- **Subject Listing by Course** (disciplina_list_course): Users can request all subjects included in a specific course, with an option to filter by semester.

The actions query CSV files containing structured information about courses (curso.csv), subjects (disciplinas.csv), and professors (docentes.csv).

1) **curso.csv**: Contains course details including codes, names, acronyms, and degree levels;
2) **disciplinas.csv**: Contains subject information including codes, names, teaching staff, and associated courses;
3) **docentes.csv**: Contains faculty information including names, email addresses, and office locations.

N. Fallback message

To handle situations where the chatbot is unable to understand the user's input, we implemented a Fallback Message feature. This is useful for cases when the chatbot doesn't have a good confidence level - the user input not fit any intent. The classifier will assign the *nlu_fallback* intent in such cases.

IV. Conclusion

DETI Pal, with its features like providing information about DETI, points of interest, and media recommendations, not only recognizes users' emotions and remembers their personal preferences but also handles grammatically incorrect sentences effectively. This showcases the potential of conversational agents to enhance user experience by offering personalized, intelligent, and context-aware interactions. By being able to respond appropriately to a variety of user needs – from emotional support to information retrieval – conversational agents like DETI Pal are becoming increasingly important in both educational and commercial settings.

This project also showcased the power of Natural Language Processing (NLP), showing how advanced algorithms can understand and process human language to generate appropriate responses. The use of the Rasa framework further emphasized the flexibility and scalability of conversational AI, providing a customizable and adaptable platform that can meet different user needs. By combining NLP and machine learning, Rasa enables the development of advanced chatbots that can evolve with user interactions, ensuring ongoing relevance and engagement.

In the future, this project could be expanded to support additional languages, such as Portuguese (the language of our university), and include more emotional features, for example, recognizing when a user is angry and attempting to calm them down.

References

[1] https://deepai.org/machine-learning-glossary-and-terms/conversational-agent (17-05-2019)

[2] https://legacy-docs-oss.rasa.com/docs/rasa/
[3] https://developer.imdb.com/non-commercial-datasets/
[4] https://developers.google.com/maps/documentation/javascript/get-api-key
[5] https://developer.themoviedb.org/reference/intro/getting-started
[6] https://spacy.io/