Language Understanding Systems - Final Project
Dialog System within Rasa framework in Formula1 domain

Sebastiano Chiari
220527
`sebastiano.chiari@studenti.unitn.it`

1 Introduction

The goal of this final project for the Language Understanding Systems course was to build a dialogue systems within the *Rasa* framework in the Formula1 domain. The chatbot needs to understand and answer the user's questions, handling basic and more complex conversations. More, this dialogue system has been integrated with Amazon's voice assistant, Alexa.

All code is available on Github [1].

2 Architecture

2.1 Rasa Open Source

Rasa is an open source machine learning framework for automated text and voice-based conversations. It understands messages, holds conversations and connects to messaging channels and APIs.

2.1.1 Finding data

Being very passionate about Formula 1, I immediately thought about creating a bot that could help answer simple questions or curiosities within this domain. Unfortunately, thanks to the recent partnership the FIA (*Federation International de l'Automobile*) has with AWS (*Amazon Web Services*), there are no official APIs available. But, I was able to find the Ergast Developer API [2], an experimental web service which provides data for the Formula One series, from the beginning of the world championships in 1950, for non-commercial purposes.

These APIs are not exhaustive and, in my opinion, also difficult to use and parse, but sufficient for the purpose of this chatbot. They are available in different formats (downloadable database, json, etc.) and I decided to use the json format to facilitate the interactivity of the questions and to allow the chatbot to answer correctly and always in an up-to-date fashion.

[1] `https://github.com/sebastianochiari/LUS-final-project`
[2] `http://ergast.com/mrd/`

2.1.2 Rasa entities

Entities are structured pieces of information that can be extracted from a user's message. Entities are annotated in training examples with the entity's name and are automatically extracted by Rasa during training through the NER (Name-Entity Recognition).

In this project, the following entities can be found:

- `circuit`, name of the circuit
- `constructor`, identification name of the constructor team
- `driver`, identification name of the driver
- `driver_or_ranking`, state if the user is interested in the driver or in the global ranking for a given race
- `year`, a specific year the user is interested in

2.1.3 Rasa intents

Intents are used to understand and classify what the user probably wants to know. Intents are key structural elements in order to guide the conversation along different paths and according to predefined rules. The following intents has been implemented:

- `greet`, the user just joined the conversation and wants to greet
- `goodbye`, the user is leaving the conversation
- `affirm`, the user has provided an affirmative answer

- `deny`, the user has provided a negative answer
- `stop`, the user is not happy with the results provided
- `thankyou`, the user is happy with the results provided and thanks the chatbot
- `inform`, the user is providing more information
- `request_info`, the user wants to know what the bot is capable of doing
- `request_joke`, the user wants to hear an F1 joke
- `request_race_results`, the user want to know the result of a particular race
- `search_season_winner`, the user want to know who win a particular season
- `search_driver`, the user wants to know more about a particular driver
- `search_next_race`, the user wants to know the next race on calendar
- `request_drivers_by_constructor`, the user wants to know the drivers of a particular team (*optional*, in a specific year)
- `driver`, the user is interested in the driver
- `ranking`, the user is interested in the ranking
- `reset_slots`, the user wants to reset all the slots filled previously
- `nlu_fallback`, used in order to handle answer with very low confidence

2.1.4 Rasa actions

After each user message, the model will predict an action that the assistant should perform next, according to stories, rules and how the model has been trained.

Some of the actions are directly defined inside the `domain.yml` file and represent some basic response the bot can use to reply to basic user intents. Most of them have multiple possibilities of answering, which are causally chosen by the chatbot when the action is triggered, in order to introduce variety during the conversation.

- `utter_greet`, replies to the user with a greet
- `utter_goodbye`, says goodbye to the user
- `utter_rejoice`, shares the joy of successfully completed a task
- `utter_noworries`, replies to a compliment from the user
- `utter_sorry`, says a sorry phrase to the user
- `utter_submit`, replies with a feedback to the user when a form has been submitted
- `utter_default`, standard replies in case something went wrong
- `utter_reset_slots`, confirms that all the slots have been set to `None`
- `utter_give_info`, returns a list of all the bot is able to answer to
- `utter_ask_year`, asks the year information
- `utter_ask_driver_or_ranking`, asks the user to specify whether is interested in the driver or in the ranking information
- `utter_ask_driver`, asks the driver name
- `utter_ask_constructor`, asks the constructor name

More complicated actions, that implies APIs calls, data management, etc. can be performed and their implementation can be found inside the `actions/actions.py` file.

- `reset_all_slots`, resets all the slots to the initial value, `None`
- `reply_with_joke`, replies to the user with a random joke selected from the `data/F1jokes.txt` file
- `action_search_driver`, searches the asked driver, replying with the (if available, simple) Wikipedia summary
- `search_drivers_by_constructor`, based on the given constructor team, if the slot `year` is set to `None` it retrieves the

drivers of that team for the current year; otherwise, if the year is specified, it searches for the constructor's drivers in the selected year

- search_next_race, given the current date, it looks for the next race in calendar
- search_race_result, based on the provided slots by the user, it retrieves the podium of the selected race (if driver_or_ranking is set to /ranking) or the selected driver final position (if the driver_or_ranking slot is set to /driver)
- search_winner_by_year, given the year, it returns name, points and win races of the season champion

2.1.5 Rasa forms

Forms are the most practical way to collect pieces of information from a user in order to do something. Forms are useful when executing actions where several slots are involved.

Forms are powerful tools because they allow also to have a validation of the input (for example, when the user inputs the year, the form will request the input until the year is between 1950 and the present year). Once the form is submitted, the bot will notify the success of the operation to the user and run a custom action in order to produce the desired output.

In this chatbot, two forms are defined, in order to handle the most complicated actions:

- race_result_form, it requires the user to fill the slots year, circuit, choose one of the two options of the driver_or_ranking slot (and eventually, if the user selects the /driver option, the slot)
- drivers_by_constructor_form, collects the constructor name and, if the user provides it, the year and performs the research based on these slots

2.2 Rasa X

Rasa X is a tool for Conversation-Driven Development (CDD), the process of listening to your users and using those insights to improve your AI assistant. In particular, Rasa X has been intensively used during this project in order to build interactive stories to make the chatbot better in understanding the user's requests.

This bot has been shared with some friends through Rasa X. Different users, with different backgrounds (some of them didn't know anything about Formula 1, others were more knowledgeable about it), chatted with the bot and their conversations have been recorded and used to improve the chatbot itself. This step was crucial as it allowed to:

- add NLU data that reflects how real users talk
- better understand how users are interacting with the assistant

2.3 Alexa Integration

The Alexa Skill Kit (ASK), provided by Amazon, allows to connect a dialogue system that deals with a specific task to Alexa. Within this particular case, ASK has been used only to deploy and give to the system Alexa ASR (Automatic Speech Recognition) and TTS (Text-to-speech). On the other hand, RASA was used to build the dialogue system, providing the NLU (Natural Language Understanding), DM (Dialogue Management) and NLG (Natural Language Generation) section of the infrastructure.

3 Model configuration

In Rasa Open Source, incoming messages are processed by a sequence of components. These components are executed one after another in a so-called processing pipeline defined in the `config.yml`, which define both how the NLU model is trained and how the assistant will deal with the conversation flow.

In order to train the final model, it was used the config file with the combination of the best NLU pipeline and the best core module tested, in the following sections.

3.1 NLU pipeline

Choosing an NLU pipeline allows to customize the model and finetune it on the specific domain, in order to better extract entities and understand the intent of each sentence the user submit to the dialogue system. There are components for entity extraction, for intent classification, response selection, pre-processing, and more, which affect the end result both in training the model and in how the chatbot will behave.

Three NLU pipelines were used in order to understand how the performances would be affected: all of them are listed in the Rasa documentation and they are the default pipeline, the Spacy one and the MITIE.

All the pipelines were used in order to train and evaluate the model with different amounts of training data by using the provided `rasa test` command, which works in the following way.

- Create a global 80% train / 20% test split from `data/nlu.yml`
- Exclude a certain percentage of data from the global train split (in our case, by having a very small amount of training data available, there has been no exclusion from the global train split)
- Train models for each configuration on remaining training data
- Evaluate each model on the global test split

NLU pipeline	**F1 score**
default	0.8761
Spacy	0.8858
MITIE	0,7795

Table 1: NLU pipelines test result performed over the training stories

While the default pipeline and the Spacy one have similar performances, we can find that MITIE is not performing very well. This can be due to the fact that MITIE used a pretrained dataset , which is a generalized version useful to start testing the pipeline: with a custom pretrained dataset, results can be improved significantly.

All the results (including confusion matrices, full json reports) can be found in the Github repository.

3.2 Core training

At the foundation of the core training, there are policies. Policies are used by the assistant in order to decide which action to take at each step in a conversation.

I decided to use the main policy family available within Rasa to test how different core modules will perform. The `TEDpolicy`, Transformer Embedding Dialogue Policy is a multi-task architecture composed by several transformer encoders for next action prediction and entity recognition.

The configurations can be found in the `core_modules` folder, with variations of the main hyperparameters, such as `max_history` and `epochs`. The policies have been tested with the same structure as the NLU pipeline.

No `MemoizationPolicy` has been used during those test: this policy remembers the stories from the training data and it checks if the current conversation matches the stories in the stories.yml file. Thus, because the training and testing is going to be performed over the training data, that would result into an overfitting scenario and the test would be useless.

core policies	**correct**
suggested	**0** / 20
TED_epochs_50	**20** / 20
TED_epochs_100	**20** / 20
TED_epochs_200	**20** / 20
TED_maxhistory_0	**6** / 20
TED_maxhistory_10	**4** / 20

Table 2: Core policies test results performed over the training stories

4 Problems and future improvements

4.1 Lack of training data

Many questions can be asked in a very similar way (ex. "who won the F1 championship in 2018" or "who won the Monaco Grand Prix in 2018"). This can lead to a misclassification by the NLU module regarding the intent the user expressed (ex. search_driver_winner vs search_race_result, making it difficult to use the bot.

Surely, the best way to overcome this difficulty is to have large amounts of data and conversations with users available, together with other existing data regarding the domain of interest that would allow more effective training.

In my case, not having this available, I chose to use different ways to ask similar questions, so that it was easier for me to avoid the problem, while remaining aware that I had not overcome it at all.

4.2 The difficulty of proper names

Dealing with a domain where proper names (even very exotic every now and then) are on the agenda, Alexa ASR seems to fail every now and then, offering imaginative interpretations. This can lead to difficulty getting Alexa to recognize our real intent, which can be solved by asking the question in written form.

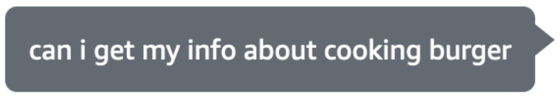
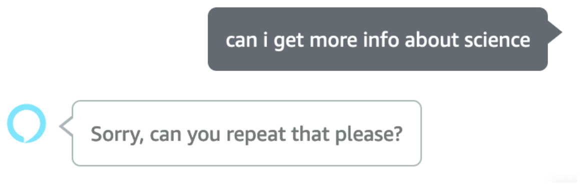

can i get my info about cooking burger

Figure 1: Nico Hulkenberg confused for a "cooking burger"


can i get more info about science

Sorry, can you repeat that please?

Figure 2: Sainz misspelled for "science"

4.3 Dealing with complicated tasks

While on the Rasa side everything seems to work fine, Alexa was particularly difficult to integrate, especially in performing complex tasks, such as the `search_race_result` action.

Unfortunately, my lack of knowledge of this environment did not allow me to fine tune this action, which cannot be completed. In the case of the search by ranking, the action seems to stop in the processing phase, not returning any results. In the case of the driver search, the dynamic addition of the field to indicate the driver's name seems to fail, not allowing the action to be completed.

This task instead from the terminal and interface of Rasa X is correctly interpreted and completed by Rasa itself: this led me to hypothesize of a missed optimization step but I was not able to find a proper solution.