Documentation project Chatbot.

Abstract:

This present document aims at explaining the work I've done as part of my 10-week internship of summer 2019 in BMAT. Once again I'd like to thank the company for its warm welcoming and the opportunity offered to expand my skills in the field of NLU.

I'll try to explain and justify my choices /results and define the current state of the project allowing the company to continue it.

Introduction:

The project could be divised in several parts:

- **The user interface:** I used the open source project "mrBot" : https://github.com/mrbot-ai/rasa-webchat. The javascript file has been modified so as to allow for interpreting url from markdown syntax and allowing for user redirection to proper FAQ page.

- **User interface 2 using FancyBox:** FancyBox used to be an open-source project. Basically, I think it improves the user experience by allowing the openning of the proper FAQ article on as a layer above vericast platfor (no need to redirect to another tab).

Old open-source available there : http://fancybox.net/
Current used version there : http://fancyapps.com/fancybox/3/ (seems to cost 29$)

I have very little front-end knowledge, but it might be possible to develop custom solution to improve user experience.

- **Rasa Framework version 1.0.9** Rasa is an open-source framework allowing to conceive chatbot using popular NLU/NLG algorithms available in libraries such as Spacy, Tensorflow or Keras.

- **A platform to manage the chatbot:** That's the aspect of the project I didn't manage to finish and that I won't be able to complete. Basically I tried to develop a platform using javascript and Flask that would allow silutaneous editing of questions and answers but which results being very buggy and unreliable. Also I may advise the use of another open-source project I wasn't convinced of during the internship but which seems to have greatly evolved https://mrbot.ai/.

The Rasa Framework:

Why use this framework ?

The Rasa framework is used by many companies (Decathlon, Orange, Valtech… just to mention a few of them) the documentation is quite complete and forum is active (All my questions were answered in less than 10hrs). It's an open-source project.

Quickstart, get the environnement ready:

Important:

- The development/deployement must be done on a Linux/Mac Os machine (Except if the asyncio package is now fully compatible with windows)
- Developement has been done on an Ubuntu 18.04 computer with python 3.6, works also on Arch with python 3.7… Every machine with Linux kernel and python 3.X shold do the job.

Installation:

Versions of python-engineio and python-socketio must be set to those version, newer versions aren't compatible with rasa 1.0.9

```
pip install python-engineio==3.8.1 python-socketio=4.1.0 rasa==1.0.9 spacy
python -m spacy download en_core_web_md
python -m spacy link en_core_web_md en
```

Get started on Rasa:

Though documentation is rather complete, I'll try to explicit the functionning of the framework:

Rasa framework must be provided a **pipeline** of NLU algorithms that it will call sequentially. I personnally kept one of the template configurations offered on the website and which happens to work well : "*pretrained_embeddings_spacy*" (Spacy being the fastest NLU library available today). Pretrained word embedding models allow an understanding of synonyms and sentence structures that aren't part of your training set.

One could easily implement its own NLU components, but apart from sentiment analysis everything is built-in in Rasa for common languages (English, Spanish, French, German…).

Rasa must also be passed **policies** configuration. Policies define the rules chatbot must follow to choose its next action. From a given message, each action receives a probability score, the most probable is usually the one chosen. Though, if first and second most

probable actions don't have a wide enough gap, or if probabilities are not sufficiently high,chatbot asks for reformulation or confirmation.
Policies currently used are:

- name: *MemoizationPolicy* : If user ask same questions as those in the training set, probability of next_action equals 1.0.
- name: *KerasPolicy* : Use a Keras NN to predict next action (otherwise developper has to implement its own ML algorithm)
- name: MappingPolicy : Allows to directly map an intent to an action (I'll define those terms later)
- name: TwoStageFallbackPolicy: Allows for reformulation of the User question in case of fail in understanding the question. After two fails, user is invited to contact client support (but other custom actions can be implemented).

Rasa vocabulary:

- *Pipeline* : Designate the set of sequential NLU algorithms to execute to analyse a question.

- *Policies*: Designate all the rules to apply to make decision for next action

- *Intent* : Quite transparent, corresponds to the intention of the user, this what the chatbot tries to understand.

- *Stories*: Stories explicit the possible sequence of intents possible. This allows contextualization of intents. (Ex: The chatbot explains a service available at BMAT and the client asks "how much would it cost ?" thanks to stories the chatbot knows the client is refering the previously mentionned product). This problem can also be solved using entities and custom actions.

- *Entities*: Entities are class of words that can be defined. If those words are found within a question *"CRFEntityExtractor"* present in the pipeline will return the list of found words and their classes. These can be used later to contextualize a question or execute a custom action.

- *Action*: Constitute the response of the chatbot to a given intent by respecting policies rules. An action is not necessarily an answer but can also be custom code executed from the "*Actions.py"* file.

At the root of your Rasa Folder you'll find 4 folders and 4 files:

> *domain.yml*: Contains all the actions name references (Be it a classcial answering action or a custom action), it also contains all the intents name references and the template answers in YAML format.

> *actions.py* : Contains all the custom actions.

> *credentials.yml* : Basically which protocol to use to communicate and its parameters (where to listen, where to answer).

> *endpoints.yml*: To be able to execute custom actions, a local server is set that uses http requests. “endpoints.md” file defines its local IP and port.

> *intent_mapping.csv*: File used for reformulation, each intent is matched with a sample question to reformulate user’s question when unsure (TwoStageFallBack policy)

> *data :* Which contains two files “*nlu.md”* and *“stories.md”:*

- *nlu.md* : Training set with Intents and their sample questions
- *stories.md* : Blocks that describe what sequence of actions to perform according to user questions and answers.

> *database*: Supposed to contain whatever data structure the user has defined in command line while launching rasa server. In practice I didn’t manage to make an SQL database so I stuck with default data structure (more on that later).

> *models*: Backup history of all trained models, when launching rasa server most recent model is used.

> *tmp*: Well tmp of current model…

Run Rasa servers :

> `source` virtual environnement
> `rasa run --log-file out.log --endpoints my_endpoints.yml` # launch rasa nlu server
> `rasa run actions` # To run in another terminal to launch custom action support
> *Open either “fancybox” or “new tab” version of* `index.html` *in UI folder*

Problems encountered:

- I can’t manage to get the TwoStageFallBack policy to work again
- Platform is buggy and unreliable, I think best option would be to look at https://mrbot.ai/.

Possible improvements

Middleware:

There are two solutions to have multilanguage support, either create a way for the front-end chat window to retrieve the addresses of wanted chatbot, or communicate with chatbots using an intermediary layer that redirects the user messages to the correct one.

It's possible to create agents (which are chatbots nlu processors). It used to be much more accessible, but basicallly you need:

- To create an "*action_endpoint*" using *EndpointConfig* in *rasa.utils.endpoints*
- To create an agent using "*Agent*" class from *rasa.core.agent*
- Use asyncio to create an async function to handle messages for each chatbot and then use the buitl-in function of Agent class "*handle_message*" to get response .

" *response = await my_agent.handle_message(message)*"

Also you need to create a socketio server (or any protocol you like depending on the final front-end solution chosen) to establish connections with users and redirect messages to correct chatbot thanks to messages headers.