International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 9 Issue 06, June-2020

An Analytical Study and Review of open Source Chatbot framework, RASA

Rakesh Kumar Sharma
Scientist-D
National Informatics Center (NIC)
India

Manoj Joshi
Scientist-E
National Informatics Center (NIC)
India

***Abstract*— In the era of chatbots, besides imitating humans they can also perform complex tasks like booking tickets for movie etc. Out of various implementations, RASA is open source implementation for NLU and DIET model. It can interact with database, api, conversational flow, interactive learning with reinforcement Neural network. In this study, various features of rasa core are studied and upto much extent it can perform complex tasks. Implementation details are studied like interaction with database, API. Tracker Store has been examined with modifying the socket.io core file adding metadata to the user message data, so that user ip and port can be captured. Furthermore, the action, interactive learning and implementation details are tested on windows Pycharm IDE**.

Keywords—Chatbot,Rasa,open source,NLP

I. INTRODUCTION

Rasa Conversational AI assistant is quiet different than earlier traditional FAQ interactions as it is based on natural conversations means like how humans interact with each other by considering what earlier the context was sent and what actions are to be taken in reference to the contexts and gracefully handling the unexpected conversation and driving the conversation when the user drifts from normal conversation path and also improve over time thus its far beyond the FAQ Interactions.

Rasa Conversational AI assistant normally consists of two components and they are Rasa NLU and Rasa Core. Rasa NLU can be just treated like ear which is taking inputs from user and Rasa Core is just like the brain which will take decisions based on user input.

II. RECENT WORK

Chatbots are software that behaves like human answering the question which are asked,Now a days , development in this field going to scale up , now chatbots not only answer simple questions but can perform complex task like fetching the results from api, booking tickets and various management tasks.For the development of a chatbot, it requires a team of many experts but to overcome the scarcity of time and resources, re-use of open source software is done.

In this paper[1], Bocklisch, T et al introduced the rasa NLU and core for the first time with open source license. The aim of this study was to provide a dialogue system based on machine learning and understanding the language to the enthusiast whoare no such expert in technology. The package[2] they developed was of minimal size and advancement is done in the package. With the efforts of 344 contributers , 244 releases of rasa have been released with total of 18023 commits.

In his work [3], Lacerda used the core of rasa and presented a new software stack called as Rasa-ptbr-boilerplate for the non specialist who doesn't much about the internals of the chatbot , considering the chatbot as blackbox.

Now, chatbots are intelligent systems which can perform complex task and has many application in robotics and natural language processing.In the recent time, there is development in the field of artificial intelligence, so the chatbots are acting as customer service agents. In this study [4], Jiao designed a functional framework which implements the principle of RASA NLU and further more he integrated the RASA NLU with the neural network methods resulting into an entity extraction system and later on recognizes the intents and related entities. This study showed that Neural network outperforms in with RASA NLU.

In the study[5], rasa chatbot is compared with Microsoft Bot, RASA, and Google Dialogflow and the various performance metrics have been tested, the extensibility and open source license makes RASA more versatile than other enterprise softwares.

This paper examines[6],the available technologies and implemented the use case prototype to give up insight that chatbot can be used with social networks so that smart chat system can be employed.

In the study[7], a social chatbot for the football has been designed which answer the questions to the Spanish football league. Chatbot is deployed with slack client, having text based interaction. It extracts the information about football players, team and their trainers.

III. FLOW CHART

1.The message from the end user is fed to the Rasa NLU (Interpreter) whose output is structured output containing the original text, the intent and entities if any, shown in Fig

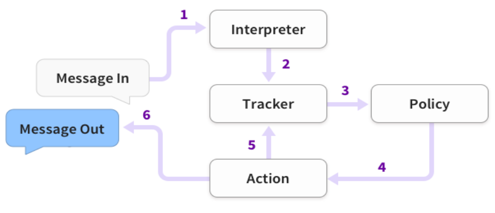


Fig. 1. Rasa architecture flow chart

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 9 Issue 06, June-2020

2. The tracker maintains the conversation state and receives the structured output from interpreter.

3. The output from tracker fed to the Policy, which acts on the current state of tracker.

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.

IV. FEATURES

A. Policies

The class rasa.core.policies.policy decides what action is to be taken during each conversation steps with bot.There are many training policies in rasa core like MemoizationPolicy, MappingPolicy, KerasPolicy, TEDPolicy, EmbeddingPolicy, Form Policy, FallbackPolicy and TwoStageFallbackPolicy.For obtaining best results and performance of our model these policies should be used in combination. These policies are configured in config.yml. Two parameters Max_History and Data Augmentation affect the performance of model so they need to be setup considering performance.

The execution of the policy will be based on the priority; highest priority policy will be executed first and so on. Priority is calculated on the basis of confidence score of each policy, which having higher score given higher priority.

The default policies are:

5. FormPolicy
4. FallbackPolicy and TwoStageFallbackPolicy
3. MemoizationPolicy and AugmentedMemoizationPolicy
2. MappingPolicy
1. TEDPolicy, EmbeddingPolicy, KerasPolicy,
and SklearnPolicy

If MappingPolicy and MemoizationPolicy predict the next action with same confidence score then the action related to MemoizationPolicy will be executed provided other parameters remain same.

B. Slots

To fill the forms, the entities value to be retained in the memory. They can store the values which are supplied by user. Slots can be of Text, Float, Boolean, categorical and unfeaturised. Based on type of entity, they are used in different scenarios.

C. Forms

It is analogus to web forms but they are filled by Bot from the inputs of user.If a user wants to ask for restauarant search from chatbot assistant and in response to that our assistant wants to avail information like Type of food, number of people, seating arrangement etc so here comes the need of Forms and policy defined for this,FormsPolicy which is having highest priority as per the rasa.

Form name to be defined in domain.yml as:

```
forms:
- restaurant_form
```

D. Interactive Supervised Learning

To improve the efficiency of bot, by learning ,called supervised reinforcement learning.In which the bot action predicted based on confidence score, we can change the predicted intent action to guide the bot.

In the url http://localhost:5006/visualization.html, we can see the states of the dialog flow, and we can proceed for the intended dialog. New stories are generated as the dialog flow is modified, so they are generated automatically by interactive mode of rasa.

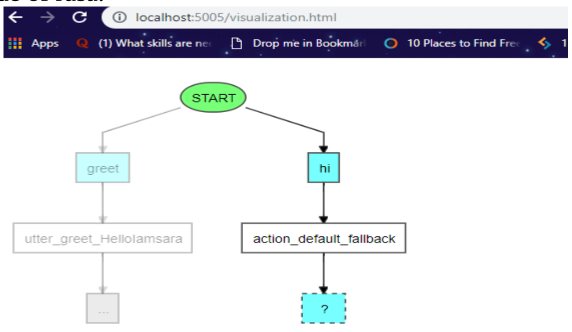


Fig. 2. States of the chatbot with supervised learning

E. Tracker Store

It is just like a kind of storage which will keep record of all the conversations with the chatbot in your database. There are many Tracker Stores provided by Rasa Core like

InMemoryTrackerStore(Default)
SQLTrackerStore
RedisTrackerStore
MongoTrackerStore
CustomTrackerStore

If we restart the rasa core entire history data in memory will be lost, this is the reason we are not using InMemoryTrackerStore. Depending upon the database you have accordingly TrackerStore can be used.

The entries of endpoints.yml are shown below:

```
action_endpoint:
url: http://localhost:5055/webhook
tracker_store:
type: SQL
dialect: "postgresql"  # the dialect used to interact with the db
url: "127.0.0.1"  # (optional) host of the sqldb, e.g.
"localhost"
port: "5432"
db: "rasa_func"  # path to your db
username: "postgres" # username used for authentication
password: "postgres"  # password used for authentication
query: # optional dictionary to be added as a query string to
the connection URL
```

V. PROCEDURE TO MAINTAIN CUSTOM LOGS

If a client/customer wants to connect rasa core, there is some requirement of Network service to be used by client/customer and some protocols are also required like http protocol,web socket etc. In addition to this for interaction of client with Rasa Core, open port must be allotted to the

International Journal of Engineering Research & Technology (IJERT)
ISSN: 2278-0181
Vol. 9 Issue 06, June-2020

individual client. To achieve this interaction with Rasa Core, a different mechanism is required by client. So how RASA Core Process will communicate with Network? The answer is provided by socketio.py which will serve the purpose here. In Nutshell, the socketio.py will provide the interprocess communication between Core and Network Socket and providing unique identification based on sender id to all the clients who are interacting with core simultaneously and thus maintaining their states.

Here we have captured only Remote_Addr and Remote_Port but we can gather much more information also.

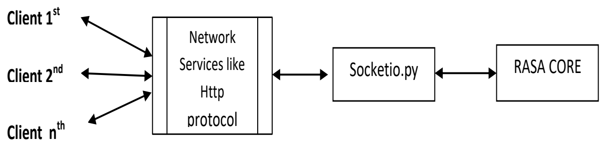


Fig. 3. Schematic Diagram of working of socketio.py

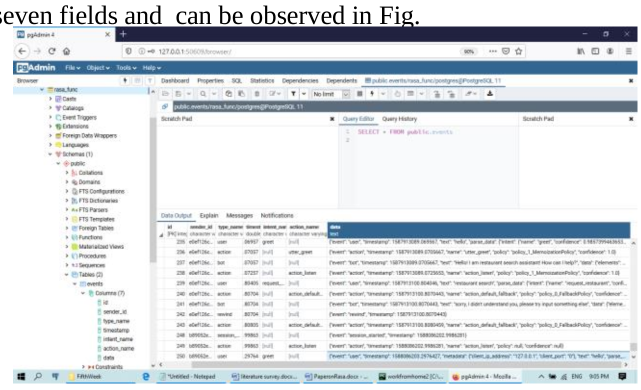
A. *Modification of Socketio.py*

In the socketio.py there is a class named SocketIOInput(InputChannel) we have taken two more members of class and these are client_ip and client_port and at the time of connection when the bot receives messages values were assigned by environ['REMOTE_ADDR'] & environ['REMOTE_PORT'] in the connection method

```
async def connect(sid: Text,environ) -> None:
    self.client_ip = environ['REMOTE_ADDR']  #capture IP
    self.client_port = environ['REMOTE_PORT']#capture Port
```

```
async def handle_message(sid: Text, data: Dict) -> Any:
            metadata={'client_ip_address':self.client_ip,'client_port':
            self.client_port,}
            message=UserMessage(data["message"],output_channel
            ,sender_id,input_channel=self.name(),metadata=metada
            ta)
 #Adds metadata collected to the message packet
```

Thus when the connection established with client, the table 'events' in 'rasa_func' database of pgAdmin4 will have the 'metadata' entry under column 'data'. The table 'events' have seven fields and can be observed in Fig.



```
SELECT * FROM public.events
```

id	sender_id	type_name	timest	intent_nar	action_name	data
[PK] integ	character v	character v	double	character	character varying	text
235	e0ef126c...	user	06957	greet	[null]	{"event": "user", "timestamp": 1587913089.06957, "text": "hello", "parse_data": {"intent": {"name": "greet", "confidence": 0.98573994636...
236	e0ef126c...	action	07057	[null]	utter_greet	{"event": "action", "timestamp": 1587913089.0705667, "name": "utter_greet", "policy": "policy_1_MemoizationPolicy", "confidence": 1.0}
237	e0ef126c...	bot	07057	[null]	[null]	{"event": "bot", "timestamp": 1587913089.0705667, "text": "Hello! I am restaurant search assistant! How can I help?", "data": {"elements": ...
238	e0ef126c...	action	07257	[null]	action_listen	{"event": "action", "timestamp": 1587913089.0725653, "name": "action_listen", "policy": "policy_1_MemoizationPolicy", "confidence": 1.0}
239	e0ef126c...	user	80405	request_...	[null]	{"event": "user", "timestamp": 1587913100.804046, "text": "restaurant search", "parse_data": {"intent": {"name": "request_restaurant", "confi...
240	e0ef126c...	action	80704	[null]	action_default...	{"event": "action", "timestamp": 1587913100.8070443, "name": "action_default_fallback", "policy": "policy_0_FallbackPolicy", "confidence": ...
241	e0ef126c...	bot	80704	[null]	[null]	{"event": "bot", "timestamp": 1587913100.8070443, "text": "sorry, I didn't understand you, please try input something else", "data": {"eleme...
242	e0ef126c...	rewind	80704	[null]	[null]	{"event": "rewind", "timestamp": 1587913100.8070443}
243	e0ef126c...	action	80805	[null]	action_default...	{"event": "action", "timestamp": 1587913100.8080459, "name": "action_default_fallback", "policy": "policy_0_FallbackPolicy", "confidence": ...
248	b89052e...	session_...	99863	[null]	[null]	{"event": "session_started", "timestamp": 1588086202.9986281}
249	b89052e...	action	99863	[null]	action_listen	{"event": "action", "timestamp": 1588086202.9986281, "name": "action_listen", "policy": null, "confidence": null}
250	b89052e...	user	29764	greet	[null]	{"event": "user", "timestamp": 1588086203.2976427, "metadata": {"client_ip_address": "127.0.0.1", "client_port": "0"}, "text": "hello", "parse...

Fig. 4. Event table details

And the metadata in data field look like this

```
{"event": "user", "timestamp": 1588086203.2976427, "metadata": {"client_ip_address": "127.0.0.1", "client_port": "0"}, "text": "hello", "parse_data": {"intent": {"name": "greet", "confidence": 0.9857399463653564}, "entities": [], "intent_ranking": [{"name": "greet", "confidence": 0.9857399463653564}, {"name": "inform", "confidence": 0.003557391930371523}, {"name": "request_restaurant", "confidence": 0.0023641688749194145}, {"name": "bot_challenge", "confidence": 0.002120556542649865}, {"name": "affirm", "confidence": 0.0019410221138969064}, {"name": "stop", "confidence": 0.0015613293508067727}, {"name": "thankyou", "confidence": 0.0015150005929172042}, {"name": "deny", "confidence": 0.0009625948732718825}, {"name": "chitchat", "confidence": 0.00023804829106666148}], "text": "hello"}, "input_channel": "socketio", "message_id": "abf0cc8d44014c9d95ad043d0aa6dea9"}
```

VI. INTEGRATING WITH API AND DATABASE

When query or data is received by rasa from the end user, rasa will predict the values of entities and intents from the message, all this handling is done by RASA NLU unit.

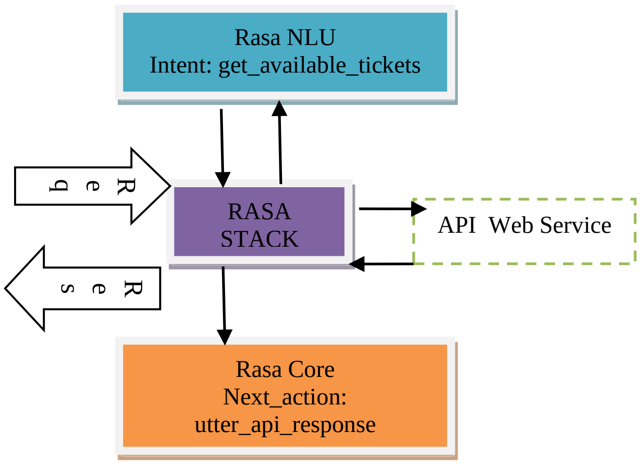


Fig. 5. API communication procedure

Intent is classified from the message and rasa stack will act on the action that is defined in the domain.yml .Based on this action, request will be done to the defined api in action.py, to get the updates requested by the user.

After this rasa core tries to predict what to do next, this decision will be based on various learning paradigms and dialog flow in the stories.yml.

A. *Defining Actions*

Intent is classified from the message and rasa stack will act on the action that is defined in the domain.yml .Based on this action, request will be done to the defined api in action.py, to get the updates requested by the user.

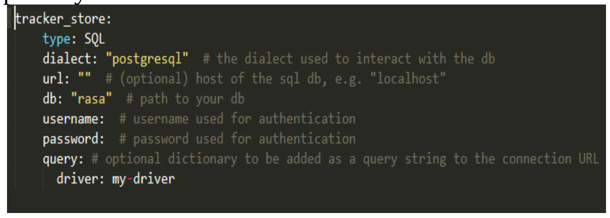
After this rasa core tries to predict what to do next, this decision will be based on various learning paradigms and dialog flow in the stories.yml. As the work of the action is based upon requirements, here we assume fetching data from API. A get request is sent to the API, and rasa action class will work on the response received from the API.

Based on the response code, processing on the response can be done to provide the expected result to the user.

```
class ApiTicketsAction(Action):
        def name(self):
                return "ticketaction"
        def run(self, dispatcher, tracker, domain):
                res = requests.get(API_URL + "destination" +
                "?apikey=" + API_KEY)
                if res.status_code == 200:
                        data = res.json()["records"]
                        out_message = "Buses available for
                        the destination
                        {}:\n1".format(data["destination"],)
                        dispatcher.utter_message(out_messag
                        e)
                        out_message =
                        "{}".format(data["buses"])
                        dispatcher.utter_message(out_messag
                        e)
                return []
```

B. *Configuration with Database*

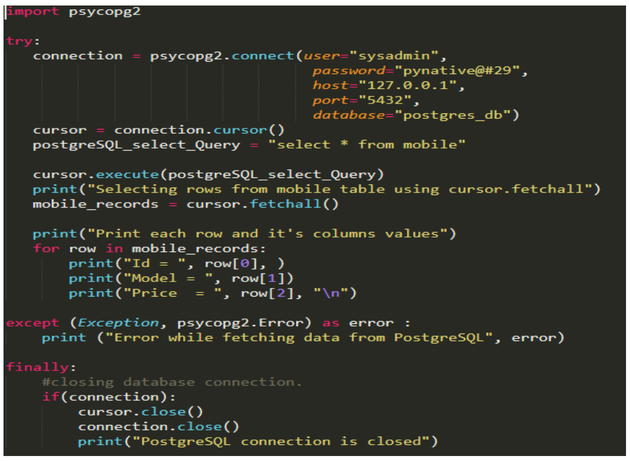
Based on the database like postgres, oracle etc required python dependencies need to be installed, like for posgres pycopg2 is the driver adapter. Its configuration is defined in endpoints.yml



```
tracker_store:
    type: SQL
    dialect: "postgresql"  # the dialect used to interact with the db
    url: ""  # (optional) host of the sql db, e.g. "localhost"
    db: "rasa"  # path to your db
    username:  # username used for authentication
    password:  # password used for authentication
    query: # optional dictionary to be added as a query string to the connection URL
      driver: my-driver
```

Fig. 6. Database adapter driver configuration in endpoints.yml

The data needed to be fetched from the database, this processing can be done in defined action as explained above. An instance of connection on which cursor is defined , is used



```
import psycopg2

try:
    connection = psycopg2.connect(user="sysadmin",
                                  password="pynative@#29",
                                  host="127.0.0.1",
                                  port="5432",
                                  database="postgres_db")
    cursor = connection.cursor()
    postgreSQL_select_Query = "select * from mobile"

    cursor.execute(postgreSQL_select_Query)
    print("Selecting rows from mobile table using cursor.fetchall")
    mobile_records = cursor.fetchall()

    print("Print each row and it's columns values")
    for row in mobile_records:
        print("Id = ", row[0], )
        print("Model = ", row[1])
        print("Price  = ", row[2], "\n")

except (Exception, psycopg2.Error) as error :
     print ("Error while fetching data from PostgreSQL", error)

finally:
     #closing database connection.
     if(connection):
         cursor.close()
         connection.close()
         print("PostgreSQL connection is closed")
```

Fig. 7. Database configuration with postgres

to execute the sql query. Based on the result from database, the response is given to end user.

VII. CONCLUSION AND FUTURE SCOPE

From the study it is concluded that rasa core features like slots, forms, supervised interactive learning, api integration, and database makes it a complete framework that can be used to perform highly complex tasks. The chatbot based on rasa has more capabilities than any open source alternative. Futher, in this paper, internals of rasa has been modified to carry out custom data logging of client ip and port.All internals and custom action has been studied which further states that rasa is a complete open source framework for the development of chatbots and for the developers who don't want to dig into the internals of natural language processing.

The future scope of this study, voice and face recognition engines can be integrated for more complex task like ATM cash withdrawal. Performance may be enhanced with use of various learning procedures of machine learning.

REFERENCES

[1] Bocklisch, T., Faulkner, J., Pawlowski, N., & Nichol, A. (2017). Rasa: Open source language understanding and dialogue management. *arXiv preprint arXiv:1712.05181*.
[2] https://github.com/RasaHQ/rasa
[3] Lacerda, A. R. T. D. (2019). Rasa-ptbr-boilerplate: FLOSS project that enables brazilianportuguese chatbot development by non-experts.
[4] Jiao, A. (2020). An Intelligent Chatbot System Based on Entity Extraction Using RASA NLU and Neural Network. *JPhCS*, *1487*(1), 012014.
[5] Singh, A., Ramasubramanian, K., &Shivam, S. (2019). Introduction to Microsoft Bot, RASA, and Google Dialogflow. In *Building an Enterprise Chatbot* (pp. 281-302). Apress, Berkeley, CA.
[6] Frommert, C., Häfner, A., Friedrich, J., & Zinke, C. (2018, September). Using chatbots to assist communication in collaborative networks. In *Working Conference on Virtual Enterprises* (pp. 257-265). Springer, Cham.
[7] Segura, C., Palau, A., Luque, J., Costa-Jussà, M. R., &Banchs, R. E. (2019). Chatbol, a chatbot for the Spanish "La Liga". In *9th International Workshop on Spoken Dialogue System Technology* (pp. 319-330). Springer, Singapore.