

Northeastern University – Silicon Valley

CS 6120 Natural Language Processing

NLP Final Project

Team Members:

Divit Vasu

Parshva Timbadia

Sadiya Bhawania

Sriram Bhardwaj

Project: Healthcare Assistant

Date: 04-29-2021

Introduction

What is NLP?

Natural language processing is a subdomain of computer science and linguistics, and AI. It involves modelling interactions between computers and human (natural) languages in order to manage and analyze data in the form of a natural language. Multiple challenges in natural language processing exist, which involve speech recognition, natural language understanding, and natural language generation at the very basic. There are various frameworks for creating ML-based chatbots like RASA, Botpress, Chatfuel, GPT3 and GPT4, Microsoft Bot Framework, DialogFlow etc.

A Brief Introduction to Chatbots

A chatbot is a program that uses voice commands, text chats, or both to mimic human communication. A chatbot, or chatterbot, is an artificial intelligence (AI) feature that can be incorporated and used in any messaging app.

Broadly, there are two types of chatbots: Rule-based Chatbot and Machine Learning Chatbot.

Rule-based Chatbots:

The conversational capabilities of a chatbot that operates under a set of rules are minimal. It can only respond to a limited number of requests and has a limited vocabulary, and its intelligence is limited to the rules and code specified. An automated banking bot, for example, is a limited bot that asks the customer a series of questions to determine what the customer wishes to achieve. One more example of such a bot would be a bot intended to serve as a knowledge source at a helpdesk or a FAQ bot.

Machine Learning-based Chatbots:

An ML-based chatbot works upon the concepts of artificial neural networks inspired by the neural nodes of the human brain. As new conversations and words are added to the bot, it is designed to self-learn (a mixture of supervised and reinforcement learning methods). As a chatbot receives sequential dialogues, the variety of requests it can respond to increases, and so does the accuracy of its responses.

What is RASA?

RASA is a machine learning platform that is open-source and very diverse in the number of functionalities it provides. It is a widely used framework to build conversational bots. So, we resorted to RASA for our project. Also, many tutorials and learning material were freely available alongside the RASA communities actively supporting forums. One can implement chatbot workflow and classify end-user interactions to intents and entities efficiently in RASA. RASA uses a dialogue flow pipeline to execute dialogues, responses (templates), and actions. Rasa Stack is a combination of Rasa X, Rasa NLU, and Rasa Core.

Motivation and Abstract

One of the significant challenges of open-source is a demanding and steep learning curve and a trial-error based implementation approach. We have built this chatbot from a demonstration point of view and have tried to implement the learnings that we have imbibed over the duration of this course. The chatbot that we have built to showcase our work is a healthcare assistant. This bot was developed to locate licensed hospitals, pharmacies for a given area and return directions to the same on Google Maps. A framework of a symptom-checker has also been included in the project to aid in self-diagnosis. However, this was not implemented in its entirety owing to time constraints. There is also a module in the project which lets a user view his insurance details, update the same or check the status of his claims. This has been demonstrated using dummy data in the SQLite database. The chatbot assists by guiding patients to a set of possible underlying conditions. Users can converse with this bot through WhatsApp via Twilio as an intermediary.

To achieve this, we:

- Researched upon the RASA framework.
- Understood its constituent files and their application and their purpose.
- Worked upon building an NLU model and wrote the necessary content.
- Worked upon modelling a dialogue flow and wrote the necessary content.
- Researched upon integrating a database with RASA.
- Researched various APIs for the symptom checker and finalized on using Infermedica.
- Researched upon deploying the bot on various platforms and choose WhatsApp for the final application.
- Researched upon and understood RASA pipelines, configurations, and the overall structure.
- Finally, we tried to improve the model's accuracy by tuning the hyper-parameters in the config file.

Software and Hardware Requirements

Hardware requirements:

CPU: Intel Core i5 or above or equivalent AMD, Xeon series recommended for large level applications.

Clock: 2.4 GHz or above

RAM: 8Gb minimum, recommended 16Gb

HDD: relative to use

Software requirements:

OS: Windows 10, Mac OS, Ubuntu (recommended)

Language: Python

Tools: Anaconda, any text editor, postman, NGrok

Libraries: Rasa core and its dependencies, Rasa NLU, GoogleMaps, SQLite3

Background

RASA Framework

RASA framework comprises of mainly two modules: RASA NLU and RASA Core. RASA NLU comprises of modelling the conversational part and the Core comprises of dialogue management.

RASA NLU is used to comprehend natural languages. This is where Rasa attempts to decipher requests in order to detect Purpose and Entity from the text input. Rasa NLU has a number of separate components for identifying intents and entities, the majority of which are interdependent.

RASA Core is crucial for providing an answer based on the direction of the conversation. Here Rasa tries to assist with the flow of contextual messages. It can anticipate dialogue as a response and trigger Rasa's Action Server depending on the User message and the response to be provided.

The Rasa bot Framework is based on the following files:

NLU- This file comprises of all the training examples for intents and entities that are required to be understood by the bot. Synonyms for particular words and regular expressions to identify phone numbers, area codes etc. can also be mentioned in this file. An intent could comprise of zero or more entities. An intent is what the user wishes to accomplish, and an entity is specific to the intent. Example: If a user wishes to find hospitals near a particular location, the intent would be to locate a hospital and the entity would be the given location. One thing to take care of in this file is to include enough and equal amounts of training examples for each of the intent. If the training examples are unequal, a class imbalance problem will arise. If the training examples are less, the model will underfit.

Actions- This file contains custom actions in the form of Python classes. These are actions that can be customized according to what is required to be performed. Example: An API call can be defined here matching to a particular intent, and the result can be returned to the user.

Domain- The Domain file defines the entire scope/universe of the bot and acts as a summariser. All the intent and entity names are defined here, alongside the names of all custom actions. It also stores slot details. Slots are RASA's way to implement conversational memory for the bot. Also, components for forms are mentioned in this file. Forms help gather information from the user iteratively. Lastly, this file also contains templates for standard responses.

Stories- This file contains all the stories. Stories are nothing but sample conversational paths that a user can take during the conversation. The more the number of stories, the more robust will be the model. The idea is to define Happy paths and sample paths wherein the bot can identify the user to be digressing. This also helps the bot model the actual conversations possible.

Config- This file contains definitions for the pipelines and the hyper-parameters that can be

tweaked according to the requirements and the task at hand. The default RASA pipeline for the English language consists of a Whitespace tokenizer, set of featurizers, a classifier and a synonym mapper. Different policies like the Fallback policy, Form policy, rule policy can be defined here. A tokenizer feeds identified tokens or lexical units into a featurizer. A featurizer then maps the input tokens to vector sets. These vector sets in turn are fed into an Intent Classifier, which in the case of RASA works upon Support Vector Machines. One can also tune the max number of n-grams that should be considered. It is seen that increasing the number of n-grams to greater than four does not affect the results much, but a higher value proves to be costly during the training phase.

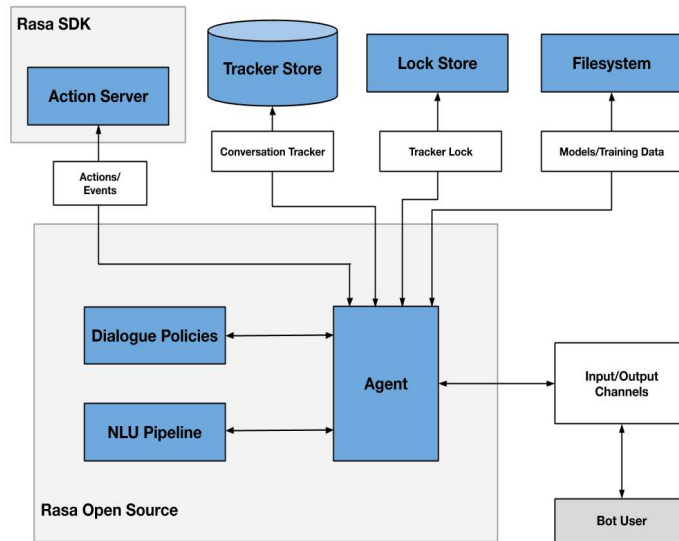
Moreover, a max-history parameter holds the number of sentences from the conversation prior to each step. The higher the value, the bot would model exactly onto the supplied stories. The number of epochs can be altered too. This nothing but the number of times the model would recurse over the training set. Maxima would occur for this value at some point in time, which can be identified empirically by performing a few training runs. A value above the maxima would not improve the training quality and add to the training cost. Moreover, the model would risk overfitting if epochs were set too high. So, a reasonable value needs to be determined on a case-to-case basis.

Rules- Rules can be used to define a series of sequences of actions that ought to be followed in any case. Example: If a user utters goodbye at any point in time, the bot should also utter a goodbye message. Such scenarios can be defined under rules. Rules have a higher priority above stories. In general, their extensive use should be avoided.

Credentials- This file can be used to store API tokens, keys and other authentication-related information. Integrations for platforms such as Twilio, Slack etc., can be defined here. Data pertaining to cross-platform communication is stored here.

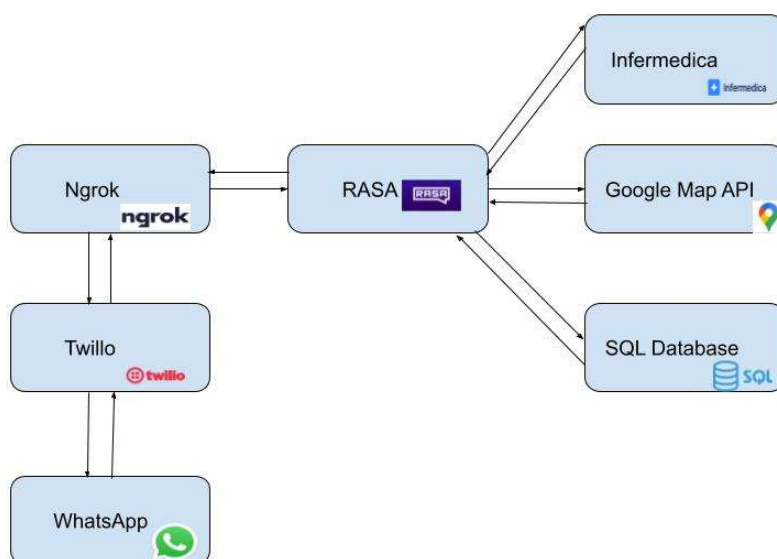
Endpoints- This file holds server endpoints. These endpoints include the URL path where the RASA server should be hosted and a database connection path. This is also used to point to a server running custom actions (defined in a python file).

RASA Architecture



The NLU section is responsible for intent classification, entity extraction, and response retrieval. Based on the context, the dialogue management aspect determines the following action in a conversation. In the diagram, this is represented by the Dialogue Policies. Conversations between Assistant and users are stored in tracker stores. Rasa employs a ticket lock system called lock store to ensure that incoming messages for a given conversation ID are processed in the correct order, and conversations are locked while messages are being processed. The above diagram has been taken as a reference from RASA's knowledge base⁶.

Our implementation Flow Chart and sub-modules



Implementation

Installation and running

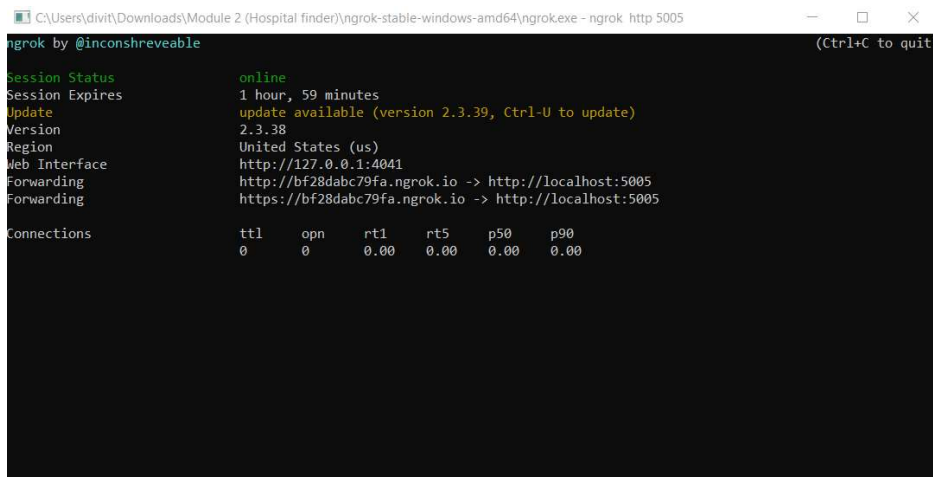
- Installation steps here have been shown for a Windows-based environment.
- First, download anaconda¹² latest distribution from its distributions page (link in references) and run the MSI installer.
- Create a virtual environment in anaconda for the installed python version and activate the same as below.

```
(base) C:\Users\divit>conda create --name project python==3.7

(base) C:\Users\divit>conda activate project

(project) C:\Users\divit>
```

- Install RASA and its dependencies with reference to supplied text file with the project. The text file contains all environmental dependencies necessary for the project to run.
- Once all the rasa files have been modelled as explained in the coming sections, the model can be trained using the 'rasa train' command. This command trains both the NLU and the Core together. The model can be invoked in an interactive learning mode after this using 'rasa interactive' command. This is a more manageable approach towards modelling stories. The rasa bot can also be invoked on the terminal using 'rasa shell' command.
- After this, an account on Twilio needs to be created, which would then be integrated to WhatsApp. Twilio would serve as our sandbox. The necessary configuration needs to be defined in the credentials.yml file.
- Once, the above is done, download ngrok¹³ from the link mentioned in the references. Ngrok serves as a proxy server and publishes the chatbot running on the local machine to the internet via a URL. Ngrok needs to be supplied with the port RASAs server has been set to run on. Ngrok will then provide a URL, which would then needs to be put into Twilio.
- The bot can now be queried using WhatsApp.



```
C:\Users\divit\Downloads\Module 2 (Hospital finder)\ngrok-stable-windows-amd64\ngrok.exe - ngrok http 5005
ngrok by @inconshreveable (Ctrl+C to quit)

Session Status      online
Session Expires     1 hour, 59 minutes
Update              update available (version 2.3.39, Ctrl-U to update)
Version              2.3.38
Region              United States (us)
Web Interface        http://127.0.0.1:4041
Forwarding           http://bf28dabc79fa.ngrok.io -> http://localhost:5005
                    https://bf28dabc79fa.ngrok.io -> http://localhost:5005

Connections
t1l    opn    rt1    rt5    p50    p90
0      0      0.00   0.00   0.00   0.00
```


Google Maps API integration

- The Google Maps API¹⁵ is a powerful tool that can be used to make a personalized map, a searchable map, check-in activities, show data flow synching with location, plan routes, to name a few examples. Google Places API can be configured to be queried by: Using a keyword search and/or filters, Ranking by distance and popularity.
- We can look for hospitals based on their location. A Place Search returns a list of places and summary information about each one; a Place Details query provides more verbose details.
- We can use the Nearby Search Request to find various places based on:
 - Position: this may be the user's current location or some other location from which one would like to find nearby locations.
 - Locations: Hospitals, and other similar establishments.
 - Radius: The radius defines how far out one wants to travel to get to the locations.
- The majority of the information about locations, such as name, address, and coordinates, will be contained in the answer item.
- We have used a text search request to get the points of interest based on a text string. "Hospitals in San Jose," for example.
- Place details request can include additional information about a location, such as a website address, contact number, weekly opening hours, feedbacks, ratings etc.
- In the code, the above-explained query works inside a custom action named 'action_hospital_location'. Hospitals, pharmacies can be found near a location using city name or zipcode as input from the user.

```
class ActionHospitalLocation(Action):
    def name(self) -> Text:
        return "action_hospital_location"

    def run(self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        poi_provided = str(tracker.get_slot("place_of_interest"))
        location_provided = str(tracker.get_slot("location"))
        print(location_provided)
        Address_found_from_API = ActionHospitalLocation.locate(poi_provided, location_provided)
        for i in Address_found_from_API:
            dispatcher.utter_message(i)
        print(Address_found_from_API)
        return [SlotSet("address", Address_found_from_API)]

    # dispatcher.utter_message(template="Here are the hospitals near {}".format(location_provided))
    # return []

    def locate(place_of_interest, location):
```

```

API_Key = 'AlzaSyBMLuCO-xkj6PV5BdUxvgh5IhbadAd'

gmaps = googlemaps.Client(key=API_Key)

long_lati = gmaps.geocode(str(location))

res_long_lati = long_lati[0]

lat = str(res_long_lati["geometry"]["location"]["lat"])

Long = str(res_long_lati["geometry"]["location"]["lng"])

response_result = gmaps.places_nearby(location=(str(lat), str(Long)), keyword=str(place_of_interest),
                                     rank_by="distance", open_now=True, type="doctor")

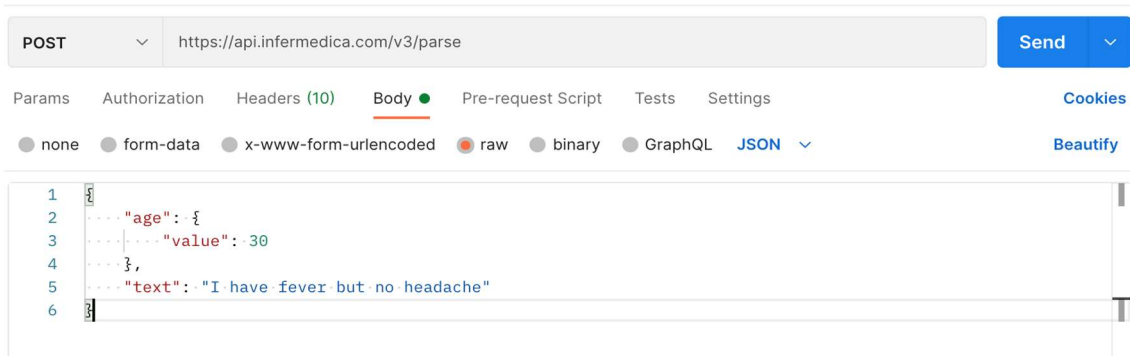
i = 1
messages = []
for place in response_result['results']:
    placeID = place['place_id']
    requirements = ['formatted_address', "formatted_phone_number", 'url', 'rating']
    place_details = gmaps.place(place_id=placeID, fields=requirements)
    info = place_details['result']
    messages.append(str(i) + ". Location Name : " + place['name'] + "\nAddress : " + place_details['result'][
        'formatted_address'] + "\nPh. no : " + place_details['result'][
        'formatted_phone_number'] + "\nRatings : " + str(
        place_details['result']['rating']) + "\nGoogle Map Link : " + place_details['result']['url'])
    i += 1
    if i == 6:
        break
return messages

```

Infermedica API integration

- We use API calls to Infermedica's API. It has a Parse endpoint that identifies what symptoms are present based on the input. This module identifies the symptoms the user is facing and provides a JSON file as the output. The JSON file contains an id which includes the unique number for a particular symptom and choice_id keeps a record of whether a specific symptom is present or not.
- For example, If the user queries something like "I have fever and headache." The parser from the API will identify the symptoms "fever" with choice_id as "present" and "headache" with choice_id as "present".

- If the user queries something like "I have a fever but no headache." The parser will identify the symptoms "fever" with choice_id as "present" and "headache" with choice_id as "absent".
- A look at the same in the Postman Application can help understand the working further.



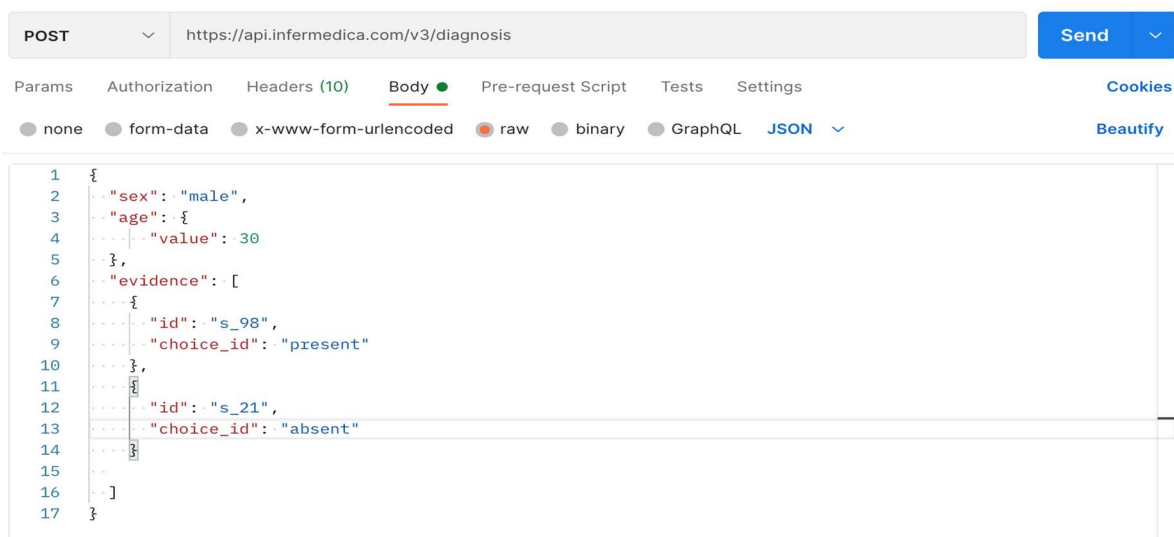
- JSON Response for the above query will be as follows.

```

{
  "mentions": [
    {
      "id": "s_98",
      "name": "Fever",
      "common_name": "Fever",
      "orth": "fever",
      "choice_id": "present",
      "type": "symptom"
    },
    {
      "id": "s_21",
      "name": "Headache",
      "common_name": "Headache",
      "orth": "headache",
      "choice_id": "absent",
      "type": "symptom"
    }
  ]
}

```

- This JSON payload helps to get the initial query from the user. The same can be done to investigate further and ask recurring questions to the user using the ids and choice_ids recorded in the first step.



- Obtaining recurring questions to probe further involves using API calls to infermedica which is being used at the diagnosis endpoint. Once diagnosis is called, a JSON file in a QA format gets dispatched. This is then combined with the rasa chatbot to seek the related prompt from the user. This prompt is again stored with its consecutive ids and choice_ids. All this information gets appended to our initial query, alongside repeating the process for at least five iterations to get accurate results.
- The JSON payload for the above explanation is as follows.

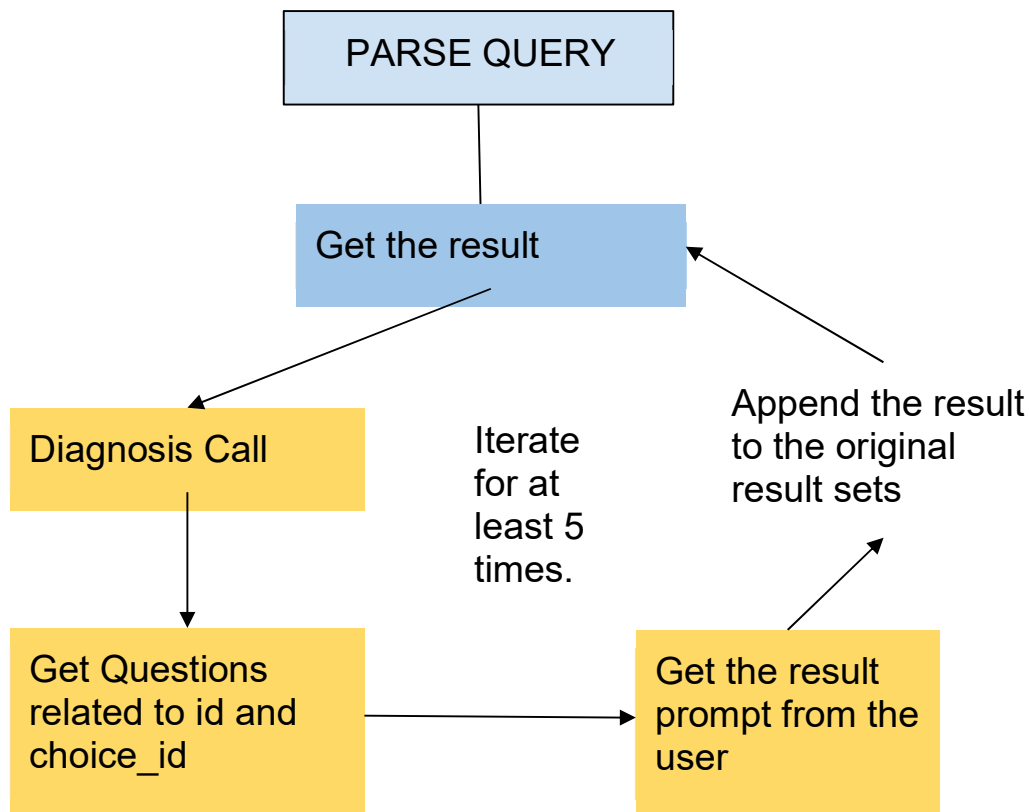
```
{
  "question": {
    "type": "group_single",
    "text": "What is your body temperature?",
    "items": [
      {
        "id": "s_99",
        "name": "Between 98.6 and 100.4 °F or 37 and 38 °C",
        "choices": [
          {
            "id": "present",
            "label": "Yes"
          },
          {
            "id": "absent",
            "label": "No"
          },
          {
            "id": "unknown",
            "label": "Don't know"
          }
        ]
      },
      {
        "id": "s_100",
        "name": "Between 100.4 and 104 °F or 38 and 40 °C",
        "choices": [
          {
            "id": "present",
            "label": "Yes"
          },
          {
            "id": "absent",
            "label": "No"
          },
          {
            "id": "unknown",
            "label": "Don't know"
          }
        ]
      },
      {
        "id": "s_2000",
        "name": "Greater than 104 °F or 40 °C",
        "choices": [
          {
            "id": "present",
            "label": "Yes"
          },
          {
            "id": "absent",
            "label": "No"
          }
        ]
      }
    ]
  }
}
```

```

{
  "id": "unknown",
  "label": "Don't know"
}
],
{
  "id": "s_1820",
  "name": "Temperature not checked",
  "choices": [
    {
      "id": "present",
      "label": "Yes"
    },
    {
      "id": "absent",
      "label": "No"
    },
    {
      "id": "unknown",
      "label": "Don't know"
    }
  ]
},
"extras": {},
},
"conditions": [
{
  "id": "c_87",
  "name": "Common cold",
  "common_name": "Common cold",
  "probability": 0.0741
}
],
"extras": {},
"has_emergency_evidence": false
}

```

- Once all the prompts from the user are collected, they are stored, and the resultant is then appended to the original query. After this, a call to diagnosis is made again.
- It can be observed that the JSON output procured for the above condition 'Common Cold' is based on only a single iteration. The result keeps on improving with every iteration.



- After iterating this about five times, the loop is exited, and then a call to yield the final diagnosis is made, which returns the most symptom.
- All the logic explained above has been incorporated as custom actions, which can be found in the **actions.py**. Snippets of classes from the same have been shown below.
- The chatbot requires a prompt from the user and then returns the list of symptoms. This is done by calling the custom action 'action_initial_query' in the 'InitialQuery' class.

```

class InitialQuery(Action):
    """
    This is the initial query which call the Parse endpoint in the API
    @returns the list of symptoms based on the query initially.
    """

    def name(self) -> Text:
        return "action_initial_query"
  
```

- From the above custom action, a list of symptoms is obtained, which is further passed on to the action 'action_diagnosis_query'.

```
class DiagnosisQuery(Action):
    """
    @returns us the list of the question to be asked for further analysis
    based on the symptoms
    """
    def name(self) -> Text:
        return "action_diagnosis_query"
```

- Questions received from the 'action_diagnosis_query' are dispatched to the user in order to record responses on the severity of symptoms.

```
class ResponseQuery(Action):
    """
    @return here the slot will be filled, based on the response by the user
    after calling the action_listen.
    """
    global modification
    def name(self) -> Text:
        return "action_response_query"
```

- The responses given by the user are stored using 'action_storage_query'. The resulting data is stored into RASA slots.

```
class StorageQuery(Action):
    """
    Store the user's responses mentioned in the above action_response_query
    """
    global choice_array
    def name(self) -> Text:
        return "action_storage_query"
```

- The storage query is called continuously alongside the 'response_query' for a total of 5 times. A reset on all slots is performed, and the 'final_query' is invoked to result in the final diagnosis.

```
class FinalQuery(Action):
    """
    Gets the disease and some of the remedies to overcome it.
    """
    def name(self) -> Text:
        return "action_final_query"
.
.
```

```

t1 = "After looking at your symptoms the most probable disease is: " +
str(disease)

    query = "Remedies for " + str(disease)

    text = "Please look at the following links for remedies : \n"

    for j in search(query, tld="co.in", num=4, stop=3, pause=2):

        text += str(j) + "\n"

    return (t1, text) #This will be dispatched in the run method!!

```

- One thing to note here is that most of the values are being stored into slots. A reset on the slots needs to be performed after every iteration, except the slot, which hold the initial symptoms. This is done by calling the custom action 'action_reset_slot'. The model understands when to reset the slots based on a slot 'flag', which is set to true by earlier invoked methods.

```

class ResetSlots(Action):

    def name(self) -> Text:

        return "action_reset_slots"

    def run(self, dispatcher: CollectingDispatcher, tracker: Tracker,
domain: Dict) -> List[Dict[Text, Any]]:

        return [SlotSet("choice", None), SlotSet("idkey", None),
SlotSet("prompts", None), SlotSet("flag", False)]

```

SQLite integration

- We are using SQLite in the backend for our database to store member information and claim information.
- For every story, action_hello_world is called, and a message to greet the user is displayed where we ask them what they need help with.

```

Your input -> hi
Hello! Please select one of the following
View member information - view
Check Claim Status - claim
Update Information - update

```

- The user provides what they need help regarding.
- For each of the above-mentioned tasks of view, claim and update: we verify username and member id. If they don't match, we don't proceed further. This a security verification we have added for health insurance.
- **View** – view member information:
 - We verify member id with the member ids in the database, and if member id doesn't exist, we give an error, else we ask for member name and check if member id matches with member name in the database.

- Valid inputs:

```
Please enter your insurance member ID.
Your input -> 1
Please enter your first name as per government issued ID.
Your input -> sadi
Ok. Thanks!
Member id:1
Name:sadi
Address: 355 suno street
Zipcode: 95126
Phone: 6692522634.
Your input ->
```

- Invalid input:
- When member id is incorrect.

```
Please enter your insurance member ID.
Your input -> 4
Please re-check your member ID. I'm assuming you missed a number.
Please enter your insurance member ID.
Your input ->
```

- When member name is invalid:

```
Please enter your insurance member ID.
Your input -> 1
Please enter your insurance member name.
Your input -> abcd
Please re-check the spelling. I'm assuming you mis-spelled.
Please enter your first name as per government issued ID.
Your input ->
```

- **Output for view:** if inputs are valid, we display user address, zip code and their phone numbers.

- **Claim** – check claim status

- Similar to 'view' task, we start with asking for member id and verifying it with our entries in the database.
- Once verified we ask the user for their claim ID.
- After input from user, we verify if the claim id exists for that member id. If it does, we show the status as "approved", "declined" or "in progress".
- If the user has no claims, it displays a message that the user doesn't have a claim and that they should contact customer care.
- Valid inputs:

```
Hello! Please select one of the following
View member information - view
Check Claim Status - claim
Update Information - update
Your input -> claim
Please enter your insurance member ID.
Your input -> 1
Please Enter your claim number.
Your input -> 123
Status for Claim 123 is In progress
Did that help you?
```

- Invalid inputs example:

```

Your input -> claim
Please enter your insurance member ID.
Your input -> 1
Please Enter your claim number.
Your input -> 101
Incorrect claim value. Please re-check.
Your input -> claim
Please enter your insurance member ID.
Your input -> 3
Please Enter your claim number.
Your input -> 123
You don't seem to have any claims at the moment.
Please contact customer care for more information.

```

- **Update** – Update member information:
 - Update task is divided into three sub-tasks i.e updating of:
 - Name
 - Address
 - Phone

```

Hello! Please select one of the following
View member information - view
Check Claim Status - claim
Update Information - update
Your input -> update
What information would you like to update:
Name
Address
Phone

```

- All of these are updated if the member id matches

- **Name:**

```

Please enter your insurance member ID.
Your input -> 1
Please enter your new member name.
Your input -> Sadiya

```

- **Output:**

```

Member id:1
Name:Sadiya
Address: 355 suno street
Zipcode: 95126
Phone: 6692522634.
Did that help you?

```

- **Address:**

```

Your input -> update
What information would you like to update:
Name
Address
Phone
Your input -> address
Please enter your new address.
Your input -> 6034 silver creek

```

○ **Output:**

```
Member id:1
Name:Sadiya
Address: 6034 silver creek
Zipcode: 95126
Phone: 6692522634.
```

○ **Phone:**




```
Your input -> update
What information would you like to update:
  Name
  Address
  Phone
Your input -> phone
Please enter your new phone number.
Your input -> 4087592634
```

○ **Output:**

```
Member id:1
Name:Sadiya
Address: 6034 silver creek
Zipcode: 95126
Phone: 4087592634.
Did that help you?
Your input -> yes!
Bye! Have a nice day!
```



- If the user affirms the help they received, the chatbot replies with a greeting and ends the chat.
- The database contains three tables.

Member_details

	Name	Data type	Primary Key	Foreign Key	Unique
1	ID	INTEGER			
2	name	VARCHAR (100)			
3	address_line1	TEXT			
4	zipcode	INTEGER (5)			
5	phone	INTEGER (10)			



- ID – stored member Id and is the primary key:
- Name – first name as per government id
- Address_line_1, address eg. 355 sunol street
- Phone – it is unique because no two people can have the same phone number.

Claims

	Name	Data type	Primary Key	Foreign Key	Unique
1	claim_number	INTEGER			
2	status	TEXT			

- Claim_number: every claim requested is a new entry in the database and claim number is a unique number associated with every claim
- Status- there are three statuses a claim request can be in – Approved, Decline and In-progress.

Claim_x_member

	Name	Data type	Primary Key	Foreign Key
1	claim_id	INTEGER		
2	member_id	INTEGER		

- Claim_id references claim table
- Member_id references member_details table.
- It is a cross table for member and claims. This is because a member can contain multiple claims.
- Claim_id stores all the claim_numbers in the claims table and member_id column represents member ids for specific claim_numbers.

🔗 Interaction with the bot

- A sample output from the terminal is as shown when the bot is being queried via WhatsApp. Once the action server and ngrok are running, the following command needs to be run: 'rasa run -m models -enable-api -cors "*" -debug'. This will publish the rasa server on the localhost.

```
(project) C:\Users\divit>cd "Downloads\Module 1 (Insurance checker)\Insurance checker"

(project) C:\Users\divit\Downloads\Module 1 (Insurance checker)\Insurance checker>rasa run --m models --enable-api --cors "*" --debug
<frozen importlib._bootstrap>:219: RuntimeWarning: greenlet.greenlet size changed, may indicate binary incompatibility. Expected 144 from C header, got 152 from PyObject
<frozen importlib._bootstrap>:219: RuntimeWarning: greenlet.greenlet size changed, may indicate binary incompatibility. Expected 144 from C header, got 152 from PyObject
<frozen importlib._bootstrap>:219: RuntimeWarning: greenlet.greenlet size changed, may indicate binary incompatibility. Expected 144 from C header, got 152 from PyObject
2021-04-28 19:52:19 rasa.cli.utils - Parameter 'endpoints' not set. Using default location 'endpoints.yml' instead.
2021-04-28 19:52:19 rasa.cli.utils - Parameter 'credentials' not set. Using default location 'credentials.yml' instead.
2021-04-28 19:52:20 rasa.core.utils - Available web server routes:
/conversations/<conversation_id>:path/messages POST add_message
/conversations/<conversation_id>:path/tracker/events POST append_events
/webhooks/rasa GET custom_webhook_RasaChatInput.health
/webhooks/rasa/webhook POST custom_webhook_RasaChatInput.receive
/webhooks/rest GET custom_webhook_RestInput.health
/webhooks/rest/webhook POST custom_webhook_RestInput.receive
/model/test/intents POST evaluate_intents
/model/test/stories POST evaluate_stories
/conversations/<conversation_id>:path/execute POST execute_action
/domain GET get_domain
/domain GET hello
/model PUT load_model
/model/parse POST parse
/conversations/<conversation_id>:path/predict POST predict
/conversations/<conversation_id>:path/tracker/events PUT replace_events
/conversations/<conversation_id>:path/story GET retrieve_story
/conversations/<conversation_id>:path/tracker GET retrieve_tracker
/status GET status
/model/predict POST tracker_predict
/model/train POST train
/conversations/<conversation_id>:path/trigger_intent POST trigger_intent
/webhooks/twilio GET twilio_webhook.health
/webhooks/twilio/webhook POST twilio_webhook.message
/model DELETE unload_model
/version GET version
2021-04-28 19:52:20 root - Starting Rasa server on http://localhost:5005

(project) C:\Users\divit\Downloads\Module 1 (Insurance checker)\Insurance checker\actions>rasa run actions
<frozen importlib._bootstrap>:219: RuntimeWarning: greenlet.greenlet size changed, may indicate binary incompatibility. Expected 144 from C header, got 152 from PyObject
<frozen importlib._bootstrap>:219: RuntimeWarning: greenlet.greenlet size changed, may indicate binary incompatibility. Expected 144 from C header, got 152 from PyObject
<frozen importlib._bootstrap>:219: RuntimeWarning: greenlet.greenlet size changed, may indicate binary incompatibility. Expected 144 from C header, got 152 from PyObject
2021-04-28 20:09:46 rasa_sdk.endpoint - Starting action endpoint server...
2021-04-28 20:09:46 rasa_sdk.executor - Registered function for 'action_hello_world'.
2021-04-28 20:09:46 rasa_sdk.executor - Registered function for 'validate_input_details_form'.
2021-04-28 20:09:46 rasa_sdk.executor - Registered function for 'validate_claim_form'.
2021-04-28 20:09:46 rasa_sdk.executor - Registered function for 'validate_name_form'.
2021-04-28 20:09:46 rasa_sdk.executor - Registered function for 'validate_address_form'.
2021-04-28 20:09:46 rasa_sdk.executor - Registered function for 'validate_phone_form'.
2021-04-28 20:09:46 rasa_sdk.endpoint - Action endpoint is up and running on http://localhost:5055

validating member id
Connection to SQLite DB successful
validating member id
Connection to SQLite DB successful
validating member name
```

- Ngrok then guides the localhost path to the created Twilio webhook as below.

```
C:\Users\divit\Downloads\Module 1 (Insurance checker)\Insurance checker\ngrok-stable-windows-amd64\ngrok.exe - ngrok http ...
ngrok by @inconshreveable (Ctrl+C)

Session Status      online
Session Expires     1 hour, 33 minutes
Update              update available (version 2.3.39, Ctrl-U to update)
Version             2.3.38
Region              United States (us)
Web Interface        http://127.0.0.1:4041
Forwarding            http://6067f46493ae.ngrok.io -> http://localhost:5005
                    https://6067f46493ae.ngrok.io -> http://localhost:5005

Connections          ttl    opn    rt1    rt5    p50    p90
                    15     0      0.00   0.01   5.96   6.94

HTTP Requests
-----
POST /webhooks/twilio/webhook 204 No Content
POST /webhooks/twilio/webhook 204 No Content
POST /webhooks/twilio/webhook 204 No Content
POST /webhooks/twilio/webhook 204 No Content
POST /webhooks/twilio/webhook 204 No Content
POST /webhooks/twilio/webhook 204 No Content
POST /webhooks/twilio/webhook 204 No Content
POST /webhooks/twilio/webhook 204 No Content
POST /webhooks/twilio/webhook 204 No Content
POST /webhooks/twilio/webhook 204 No Content
```

- The entire interaction with the bot has been recorded in a video clip.

Conclusion and Future work

This project helped us to gain a deeper understanding of NLP systems, their design and their working. We were able to associate the concepts of Natural language processing and understanding by means of implementing the chatbot from scratch. We also gained a lot of insight into the RASA framework and its configuration pipelines. Lastly, the symptom checker part has been demonstrated in RASAs interactive mode, as that required a lot of development to be redone, on account of an inherent drawback of RASA which we came across, quite late into the project, as until then most of our modules had already been built. A link from [stackoverflow¹⁴](#) has been provided under the references section, which describes the issue. RASA does not allow the use of a for loop to iterate and ask for continuous user inputs inside a custom action. This issue can be resolved by further breaking the modules and including a few more model stories. In the interactive mode (refer to video), it can be observed that the bot identifies the next step correctly in most cases. There are only a few cases that would need to be addressed by some tweaks.

The bot that we have built is more aimed towards demonstrating what we have learned overall and, thus, is not extensive in nature. As future scope of this project, we could integrate these functionalities into one single bot with a symptom checker. A real-life application of the same could involve leveraging cloud services for computing. Also, the model can be made to train with real-time data from incoming user queries to make it more robust. This would require a lot of data pre-processing, pipelining and handling, which would be compute-intensive. A rail-based path could also be provided as an option by incorporating the use of pre-defined buttons for sequential tasks. These sequential tasks could comprise appointment scheduling, showing FAQs etc. The bot could also be modelled to be used as a front-desk respondent at a hospital.

References:

1. <https://blog.rasa.com/ai-assistants-in-healthcare-an-open-source-starter-pack-for-developers-to-automate-full-conversations>
2. <https://rasa.com/docs/rasa/user-guide/installation/>
3. <https://rasa.com/docs/rasa/user-guide/rasa-tutorial/>
4. <https://www.investopedia.com/terms/c/chatbot.asp>.
5. <https://towardsdatascience.com/create-chatbot-using-rasa-part-1-67f68e89ddad>
6. <https://rasa.com/docs/rasa/arch-overview/>
7. <https://rasa.com/docs/rasa/nlu/choosing-a-pipeline/>
8. <https://rasa.com/docs/rasa/nlu/entity-extraction/>
9. <https://blog.rasa.com/the-rasa-masterclass-handbook-episode-3/>
10. <https://rasa.com/docs/rasa/nlu/components/>
11. <https://blog.rasa.com/the-rasa-masterclass-handbook-episode-7/>
12. <https://www.anaconda.com/>
13. <https://ngrok.com/download>
14. <https://stackoverflow.com/questions/63700081/how-can-i-use-action-listen-inside-a-custom-action-in-rasa>
15. <https://mapsplatform.googleblog.com/2012/05/google-places-api-search-refinements-as.html>