Assistant Bot.

Autor: Román Nahuel

Universidad Nacional del Centro, sede: Tandil.

Profesor a cargo: Amandi Analia

Materia: Programación Exploratoria

Resumen

Un *chatbot* es un software que simula y procesa conversaciones humanas (ya sea escritas o habladas), permitiendo a los humanos interactuar con dispositivos digitales como si se estuvieran comunicando con una persona real[1].

Este informe presenta el desarrollo de "*Assistant Bot"* un chatbot cuyo objetivo es ser un asistente virtual, el cual sea capaz de organizar reuniones, responder dudas y mantener una conversación fluida con uno o varios usuarios en simultaneo. En las siguientes secciones se encuentran el tipo de tecnologías con las cuales se trabajaron, cada funcionalidad del asistente y como trabaja el mismo.

[1] *¿Qué es un chatbot?*: https://www.oracle.com/ar/chatbots/what-is-a-chatbot/

Assistant Bot.

Assistant Bot es un chatbot conversacional creado para el ambiente educativo. Su objetivo principal es ser un asistente capaz de organizar reuniones al usuario y responder ante cierto tipo de situaciones tal y como él lo haría.

En esta esta etapa el bot está en una forma *demo*[2], quiere decir que solo cuenta con rasgos y acciones propias del desarrollador, como a su vez de sus horarios y sus preferencias. No obstante, posee la capacidad de interactuar con una o más personas y mantener una conversación fluida con las mismas.

1. Tecnologías.

En el desarrollo del bot se utilizaron varias tecnologías las cuales facilitan al desarrollador en la creación del mismo y le permiten al asistente tener diversas funcionalidades.

1.1 RASA.

Rasa es una herramienta para crear chatbots con *inteligencia artificial* [3] personalizados utilizando Python y comprensión del lenguaje natural (NLU). Rasa tiene dos componentes principales: *Rasa NLU* y *Rasa Core*.

Rasa NLU es una herramienta para el procesamiento del lenguaje natural de código abierto para la clasificación de intenciones, en otras palabras, identifica que es lo que el usuario está diciendo.

Rasa Core es responsable del flujo de la conversación, el manejo del contexto, la respuesta de los bots y la gestión de la sesión. Basado en *machine learning*[4] toma la

---

[2] *Demo*: Demostración didáctica del funcionamiento de una cosa, generalmente en el ámbito científico o técnico.
[3] *Inteligencia Artificial*: https://es.wikipedia.org/wiki/Inteligencia_artificial
[4] *Machine Learning*: https://es.wikipedia.org/wiki/Aprendizaje_autom%C3%A1tico

entrada estructurada de la NLU y predice la siguiente mejor acción utilizando la estructura de *red neuronal LSTM*.

1.2 PROLOG.

PROLOG[5] es un lenguaje de programación basado en el paradigma lógico. En si busca relacionar hechos existentes en una base de conocimiento a través de secuencias lógicas, para alcanzar una conclusión partiendo de predicados determinados.

En este trabajo se implemento esta tecnología como base de datos en la cual se almacena los horarios y materias que el chatbot va a respetar. Esto se logra apreciar con más claridad en la *figura 1*, donde se ve la forma en que se construyeron los hechos.

```
%Hechos
%

horario_ocupado_alumno('lunes',[[0,13],[15,17],[18,24]]).
horario_ocupado_alumno('martes',[[0,10],[12,17],[20,24]]).
horario_ocupado_alumno('miercoles',[[0,13],[14,16],[20,24]]).
horario_ocupado_alumno('jueves',[[0,19],[20,24]]).
horario_ocupado_alumno('viernes',[[0,16],[20,24]]).
horario_ocupado_alumno('sabado',[[0,9],[12,14],[19,24]]).
horario_ocupado_alumno('domingo',[[0,14],[19,24]]).

horario_ocupado_profesor('lunes',[[0,12],[18,24]]).
horario_ocupado_profesor('martes',[[0,8],[12,17],[20,24]]).
horario_ocupado_profesor('miercoles',[[0,13],[20,24]]).
horario_ocupado_profesor('jueves',[[0,19],[20,24]]).
horario_ocupado_profesor('viernes',[[0,16],[20,24]]).
horario_ocupado_profesor('sabado',[[0,8],[20,24]]).
horario_ocupado_profesor('domingo',[[0,11],[20,24]]).


materia_cursada('programacion exploratoria','BIEN','GUSTA',['Adrian ','Valentin.R','Luca.P','Nahuel.V','Nicolas']).
materia_cursada('analisis y diseno de algoritmos 2','BIEN','GUSTA',['Eliseo.V']).
materia_cursada('comunicacion de datos 1','BIEN','NO GUSTA',['Eliseo.V','Tomas.R']).
materia_cursada('probabilidad y estadistica','BIEN','GUSTA',[]).

gustos('estudio',['redactar informes','la planificacion del trabajo','y la programacion']).
gustos('grupales',['el respeto','la planificacion','y la organizacion']).
gustos('deportes',['voley','correr','y el futbol']).
gustos('personales',['el debate','la politica','el emprendimiento','y la int.artificial']).

```

Figura 1.

Dentro de los hechos anteriormente mostrados solo tres tienen, por el momento, gran relevancia para el bot. En primer lugar, *horario_ocupado_alumno* contiene los horarios en los cuales el asistente no podrá organizar reuniones con los usuarios que sean alumnos, mientras que *horario_ocupado_profesor* son los horarios en los que el asistente no estará disponible para organizar eventos para con los usuarios que son

[5] *Más información sobre PROLOG*: https://virtual.cuautitlan.unam.mx/intar/?page_id=212

profesores. Por otro lado, el hecho *materia_cursada* contiene por cada hecho una materia la cual el bot reconocerá como materia que "esta cursando" y a su vez información sobre como viene con la misma, si es de su agrado o no y con que personas tiene un grupo de trabajo.

1.3 Ngrok.

Ngrok[6] permite exponer a internet una URL generada dinámicamente, la cual apunta a un servicio web que se está ejecutando en una máquina local. Por lo general, suele utilizarse durante el desarrollo de un proyecto cuando surge la necesidad de integrarlo con aplicaciones que ya están funcionando en la nube.

2. Implementación.

En esta sección busca explicar la manera en la que esta implementado el *Assist bot* y mostrar de forma textual el funcionamiento que posee. A su vez, y para complementar se encontrarán imágenes del código.

2.1 Intentos, entidades y slots.

Los intent[7] o intentos son un conjunto de expresiones dichas por el usuario que significan una acción concreta. Por ejemplo, el *intent* "saluda" puede contener expresiones tales como "hola", "buen día", etc. Estas intenciones están hechas con el fin de que el chatbot "aprenda" a reconocer lo que el usuario intenta decir y poder actuar en consecuencia.

Para el la versión actual del bot se implementaron un total de veinticuatro intents. En la *figura 2* se puede apreciar la estructura de un intent(file *nlu.yml*), mientras que

---

[6] *Más información sobre Ngrok*: https://ngrok.com/
[7] *Más información sobre intents*: https://botfront.io/docs/rasa/nlu/

en la *figura* siguiente observamos en forma minimizada todos los intents(file *domain.yml*).

```
- intent: saludo
  examples: |
    - hey
    - hello
    - hi
    - Hola
    - holiss
    - Holis
    - holi
    - holu
    - Holus
    - hola
    - holaa
    - Holaa
    - buenas
    - Buenas
    - buenass
    - Buen día
    - buen dia
    - Buenos días
    - buenos dias
    - Buenas tardes
```

Figura 2.

```
intents:
  - saludo
  - despedida
  - afirmacion
  - negacion
  - satisfactorio
  - insatisfactorio
  - bien
  - cortante
  - enojo
  - triste
  - preguntas_casete
  - contactos
  - soy_profesor
  - soy_alumno
  - reunion
  - horario
  - materias_cursadas
  - consejo
  - ayuda
  - buscar_informacion
  - como_estas
  - choque
  - murio
  - fallo
```

Figura 3.

Por otro lado, tenemos las *entidades* o *entities*[8]. Una entity se trata de información que es extraída del mensaje recibido por el usuario. Si una intención tiene el significado general de una declaración del usuario, a veces necesitará información adicional. En la implementación del bot se utilizaron un total de cuatro entidades.

[8] *Más información sobre Entities*: https://botfront.io/docs/rasa/nlu/

```
29    entities:
30      - nombre
31      - titulo
32      - dia
33      - hora
```

Figura 4. File: *domain.yml*

2.1.1 Entity Nombre.

Entidad utilizada para almacenar nombres de los usuarios. Para el correcto funcionamiento se cargaron más de *144* nombres diferentes.

```
 3  nlu:
 4  - lookup: nombre
 5    examples: |
 6      - Sophia
 7      - Emma
 8      - Isabella
 9      - olivia
10      - ava
11      - emily
12      - Abigail
13      - mia
14      - Madison
15      - Elizabeth
16      - sofia
17      - Giulia
18      - martina
19      - Giorgia
20      - sara
21      - emma
22      - Aurora
23      - Chiara
24      - Alice
25      - Alessia
```

Figura 5. File: *nlu.yml*

```
606      - mi nombre es [Maria](nombre) soy Profesor
607      - mi nombre es [Matias](nombre) soy ayudante
608      - mi nombre es [nelson](nombre) soy Ayudante
609      - soy [Santiago](nombre) soy el Ayudante de programacion
610      - soy [Analia](nombre), la Profesora de programacion
611      - el profesor de, me llamo [sofia](nombre)
612      - mi nombre es [Adrian](nombre) soy profesor
```

Figura 6. Ejemplo de uso en intent “soy_profesor”. File: *nlu.yml*.

2.1.2 Entity Titulo.

Entidad utilizada para almacenar los títulos de la información que solicita el usuario al bot. Para este reconocimiento se utilizó expresiones regulares[9].

```
150   - regex: titulo
151     examples: |
152       - \w{3-10}|\w{3-10}\s\w{3-10}|\w{3-10}\s\w{1-10}\s\w{3-10}
```

Figura 7. File: *nlu.yml*

```
765 - intent: buscar_informacion
766   examples: |
767     - no encuentro información sobre [Algoritmo de Kruscal](titulo)
768     - No encuentro informacion de [combinatoria](titulo)
769     - quisiera saber sobre la [sumatoria](titulo)
770     - busco informacion sobre [Redes](titulo)
771     - Estoy buscando información sobre [tecnicas Cuantitativas](titulo)
772     - no entiendo el [machine Learning](titulo)
773     - podrias buscar informacion sobre [perros](titulo)
```

Figura 8. Ejemplo de uso. File: *nlu.yml*.

2.1.3 Entity Día.

Entidad utilizada para el almacenamiento del día que indique el usuario al momento de organizar una reunión con el chatbot.

```
154   - lookup: dia
155     examples: |
156       - Lunes
157       - lunes
158       - LUNES
159       - Martes
160       - martes
161       - MARTES
162       - Miercoles
163       - miercoles
164       - MIERCOLES
165       - Jueves
166       - jueves
167       - JUEVES
168       - Viernes
169       - viernes
170       - VIERNES
```

Figura 9. File: *nlu.yml*.

[9] *Expresiones regulares:* https://es.wikipedia.org/wiki/Expresi%C3%B3n_regular

```
661      - el [jueves](dia) estas libre?
662      - el [MIERCOLES](dia) podes?
663      - arreglamos para el [Jueves](dia)?
664      - quedamos para el [viernes](dia)
665      - ¿Podes el [Jueves](dia)?
666      - estas libre el [Sabado](dia)?
667      - yo puedo [sabado](dia)
```

Figura 10. Ejemplo de uso en intent *horario*. File: *nlu.yml*.

2.1.4 Entity hora.

Esta entidad es utilizada para el almacenamiento del horario indicado por el usuario al momento de organizar una reunión. Para el correcto reconocimiento se utilizó expresiones regulares.

```
178  - regex: hora
179    examples: |
180      - [01]?[0-9]|2[0-4]|^([01]?[0-9]):[0-5][0-9]$|^(2[0-3]):[0-5][0-9]$
```

Figura 11. File: *nlu.yml*.

```
649  - intent: horario
650    examples: |
651      - Te parece el [lunes](dia) a las [14:00](hora)?
652      - te parece el [Martes](dia) por la tarde?
653      - Qué te parece a las [19](hora)hs el [JUEVES](dia)?
654      - A las [15](hora) del (domingo)[dia] te parece bien?
655      - El [miercoles](dia) [12:35](hora) te va?
656      - [17](hora)hs el [Lunes](dia) podes?
657      - podes el [martes](dia) a las [14](hora)hs?
```

Figura 12. Ejemplo de uso. File: *nlu.yml*.

Las entidades tienen la particularidad de almacenar datos mientras la conversación siga en una misma *story*(sección 2.2), en el caso de que se cambie de *story* estos datos almacenados se perderán. Para que esto no ocurra se implementan los *slots*. Los slots son la memoria de un bot. Actúan como un almacén de clave-valor que se puede utilizar para almacenar la información que no se deseamos perder a lo largo de la conversación. *Assistant bot* tiene en uso cuatro slots que le darán contexto en la conversación (*figura 13*).

```yaml
slots:
  name:
    type: text
    influence_conversation: true
    mappings:
    - type: from_entity
      entity: nombre

  professor:
    type: bool
    influence_conversation: true
    mappings:
    - type: from_intent
      intent: soy_profesor
      value: true
    - type: from_intent
      intent: soy_alumno
      value: false

  day:
    type: text
    influence_conversation: false
    mappings:
    - type: from_entity
      entity: dia

  hour:
    type: float
    influence_conversation: false
    mappings:
    - type: from_entity
      entity: hora
```

Figura 13. File: *Domain.yml*.

En la figura anterior se aprecia cuales son los slots utilizados por el chatbot, los mismos son relevantes dentro de la conversación limitada de esta versión, donde se almacenará el nombre del usuario con el que este hablando, si es profesor o no y en el caso de organizar una reunión el horario y el día.

2.2 API de Telegram.

Rasa nos brinda la posibilidad de conectarnos con la aplicación de mensajería Telegram y poder desplegar nuestro bot en dicha plataforma a través su API[10] *Telegram Bot API*[11]. Para lograr este cometido primero se debe registrar al chatbot dentro de la

[10] *API*: https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces

[11] *Telegram Bot API*: https://core.telegram.org/bots/api

plataforma (*figura 14*), para luego con las credenciales que nos brinda la misma configurar nuestro bot(*figura 15*).

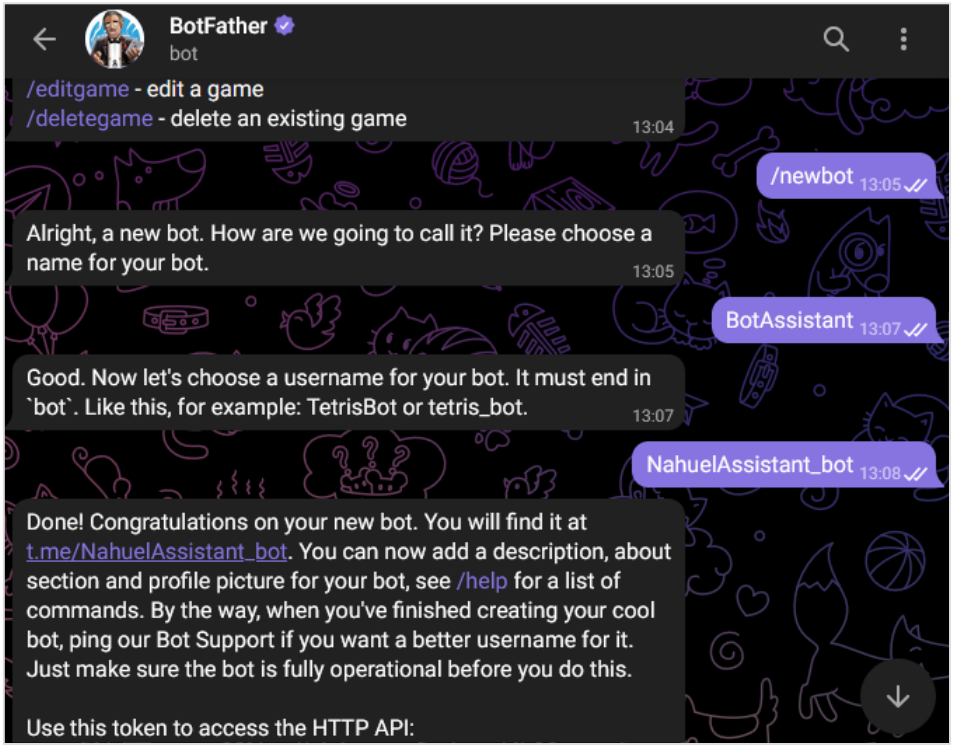

Figura 14. Registro de chatbot con BotFather.

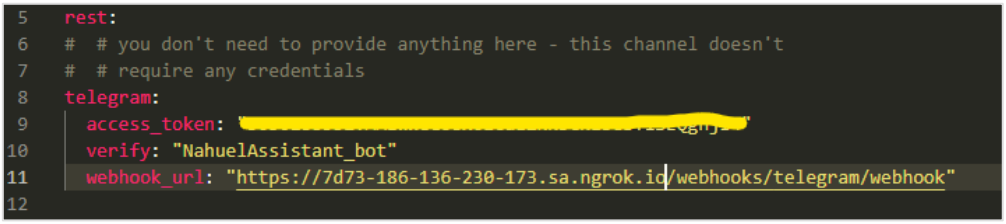

Figura 15. Configuración del canal. File: *credentials.yml*.

Telegram no solo permite la comunicación a través de su plataforma, también provee información de utilidad que viene junto con los mensajes recibidos, desde *id* del chat, hasta si se esta hablando con un bot o una persona. Esta información puede y es utilizada por el asistente virtual. Para esto se configura el archivo “channel.py” que se encuentra dentro del *virtual enviroment* utilizado en la dirección *\Lib\site-packages\rasa\core\channels*. En la *figura 16* se observa la configuración utilizada,

mientras que en la *figura 17* se aprecia parte de la información que brindan los mensajes de Telegram en formato *JSON*[12].

```python
    def get_metadata(self, request: Request) -> Optional[Dict[Text, Any]]:
        metadata= request.json
        return metadata
```

Figura 16. File: *channel.py*.

```
'message': {

    'message_id': 135,

    'from': {

         'id': 1509575415,
         'is_bot': False,
         'first_name': 'Nahuel',
         'last_name': 'Román',
         'language_code': 'es'
    },

    'chat': {

         'id': 1509575415,
         'first_name': 'Nahuel',
         'last_name': 'Román',
         'type': 'private' //tambien puede ser ⟶ 'group'

    },

    'date': 1666200891,

    'reply_to_message': { // ⟶ esto solo si hace replica en el mensaje

              'message_id': 134,
              'from': {

                   'id': 5650188532,
                   'is_bot': True,
                   'first_name': 'BotAssistant',
                   'username': 'NahuelAssistant_bot'
              },
              'chat': {
                   'id': 1509575415,
```

Figura 17.

2.3 Stories y Actions.

Las historias o *stories* son un tipo de dato de entrenamiento que se utilizan para poder entrenar el modelo de gestión de dialogo del asistente. Son representaciones de conversaciones con un usuario y chatbot, donde la participación del usuario se expresa como *intents*(sección *2.1*), mientras que las respuestas del asistente se

---

[12] *JSON*: https://es.wikipedia.org/wiki/JSON

expresan como *actions*. *Assistant Bot* cuenta con un total de veinte historias diferentes.

Una *action* es, como lo indica el nombre, una acción que tomara el bot en base a la predicción del modelo luego de cada mensaje de usuario. Hay varios tipos de acciones que pueden utilizarse, en este proyecto se aplicaron acciones de tipo *response*[13] y acciones personalizadas con *python*[14].

Cada storie y action implementada se separaron en cinco categorías: *Reconocimiento, Preguntas Simples, Ayuda , Reuniones, Busqueda de información* e *inteligencia emocional*.

2.3.1 Reconocimiento.

Esta sección engloba el inicio de la conversación, en donde el chatbot reconocerá a su interlocutor y en el caso de no hacerlo buscará registrarlo para una futura conversación. Cuenta con un total de tres stories y una única action.

```
5   - story: inic_conversation
6     steps:
7       - intent: saludo
8       - action: person
9
10  - story: i_am_proffesor
11    steps:
12      - intent: soy_profesor
13      - action: person
14
15  - story: i_am_classmate
16    steps:
17      - intent: soy_alumno
18      - action: person
```

Figura 18. Stories. File: *stories.yml*.

[13] *Response*: https://rasa.com/docs/rasa/responses
[14] *Python*: https://www.python.org/

```python
def run(self,dispatcher: CollectingDispatcher, tracker:Tracker ,domain: Dict[Text,Any])-> List[Dict[Text, Any]]:
    message = tracker.latest_message
    last_intent = str(tracker.get_intent_of_latest_message())
    id= message["metadata"]["message"]["from"]["id"]
    chat = message["metadata"]["message"]["chat"]["type"]
    entities=" "
    try:
        ent=message["metadata"]["message"]["entities"]
        entities=str(ent[0]["type"])
    except:
        pass

    if((last_intent=="consejo")&(str(chat)!="group")):
        return self.tip(dispatcher,id)
    elif((last_intent=="preguntas_casete")&((str(chat)=="private")|(str(entities)=="mention"))):
        return self.i_am_bot(dispatcher)
    elif((last_intent!="consejo")&(last_intent!="preguntas_casete")):
        return self.saludo(dispatcher,tracker,message,id)
    else:
        return[]
```

Figura 19. Action *person*. File: *actions.py*

```python
def saludo(self,dispatcher,tracker,message,id)-> List[Dict[Text, Any]]:
    register = TrabajarArchivo.cargarArchivo()
    if((register!={})&(register.get(str(id),None)!=None)):
        name = register[str(id)]["name"]
        if(register[str(id)]["professor"]):
            dispatcher.utter_message(text=f"Hola, "+str(name)+", cómo estas? Qué puedo hacer por vos?")
            return[SlotSet(key="name",value=str(name)), SlotSet(key="professor",value=True)]
        else:
            dispatcher.utter_message(text=f"Buenass, "+str(name))
            return[SlotSet(key="name",value=str(name)), SlotSet(key="professor",value=False)]
    else:
        name= next(tracker.get_latest_entity_values("nombre"), None)
        professor= tracker.get_slot("professor")
        if(professor == None):
            dispatcher.utter_message(text=f"Hola, cómo estas? che, perdon no te tengo agendado, quién sos?")
        else:
            self.make_contact(message,name,professor,register)
            register = TrabajarArchivo.cargarArchivo();
            name = register[str(id)]["name"]
            if(professor):
                dispatcher.utter_message(text=f"sii, "+str(name)+", qué puedo hacer por vos?")
                return[SlotSet(key="name",value=str(name)), SlotSet(key="professor",value=professor)]
            else:
                dispatcher.utter_message(text=f"Ahh, que onda "+str(name)+"?")
            return[SlotSet(key="name",value=str(name)), SlotSet(key="professor",value=professor)]
    return[]
```

Figura 20. Action *person*, función *saludo*. File: *actions.py*

Para que el asistente virtual tenga registro de con quien interactúa y datos sobre su ultima conversación se utiliza un archivo donde se almacenara toda esta información en formato tipo *JSON*. Explicado a detalle en la *sección 2.4*.

2.3.2 Preguntas simples.

En esta categoría se encuentran los tipos de preguntas frecuentes que nos podemos encontrar en una conversación de tipo educativa. En este caso *Assistant bot* esta programado para responder mensajes tales como: "*¿Qué materias estas cursando?*", "¿Sos un robot?", "Me darías un consejo", "Me pasarías algún número de contacto de

la facultad", etc. En este caso nos encontramos con un total de cuatro stories, dos actions personalizadas y una action de tipo response.

```yaml
- story: know_who_are_you
  steps:
    - intent: preguntas_casete
    - action: person

- story: contact
  steps:
  - intent: contactos
  - action: utter_contacts

- story: subjects
  steps:
    - intent: materias_cursadas
    - action: subject

- story: tips
  steps:
    - intent: consejo
    - action: person
```

Figura 21. Stories. File: *stories.yml*.

```python
class materias(Action):
    def name(self)->Text:···

    def run(self,dispatcher: CollectingDispatcher, tracker:Tracker, domain: Dict[Text,Any])-> List[
        message = tracker.latest_message
        chat = message["metadata"]["message"]["chat"]["type"]
        entities=" "
        try:
            ent=message["metadata"]["message"]["entities"]
            entities=str(ent[0]["type"])
        except:
            pass

        if((str(chat)=="private")|(str(entities)=="mention")):
            prolog=Prolog()
            prolog.consult('/RASA/chat-bot Alumno/actions/reglas_nahuel.pl')
            dispatcher.utter_message(text=f"Estoy cursando:")
            for materia in prolog.query("materia_cursada(X,_,_,_)"):
                dispatcher.utter_message(text=f"- "+str(materia["X"])+"")
        return[]
```

Figura 22. Action *subject*. File: *actions.py*.

```yaml
utter_contacts:
- text: "Fijate que en el link que te paso estan todos los contactos de exa. Link: https://ex
```

Figura 23. Response *utter_contacts*. File: *domain.yml*.

En las stories "*tips*" y "*kwon_how_are_you*" se implementó la misma action de la *figura 19,* utilizando otros métodos de la misma.

```python
    def i_am_bot(self,dispatcher)->List[Dict[Text, Any]]:
        dispatcher.utter_message(text=f"Soy Nahuel-bot, el Asistente virtual de Nahuel
        return []

    def tip(self,dispatcher,id)-> List[Dict[Text, Any]]:
        register = TrabajarArchivo.cargarArchivo()
        if((register!={})&(register.get(str(id),None)!=None)):
            if(register[str(id)]["professor"]):
                dispatcher.utter_message(text=f"que consejo daria... Y creo que un buen
            else:
                dispatcher.utter_message(text=f"que consejo daria... Y creo que un buen
        else:
            dispatcher.utter_message(text=f"Perdon, no te tengo agendado, quien sos?")
        return[]
```

Figura 24. Action *person*, funciones *i_am_bot*, *tip*. File: *action.py*.

2.3.3 Ayuda.

Esta categoría es encargada de continuar el flujo de la conversación cuando el usuario solicita ayuda de algún tipo, cuenta con una única historia y una única action.

```yaml
- story: help
  steps:
  - intent: ayuda
  - action: help_you
```

Figura 25. Storie *help*. File: *stories.yml*.

```python
class help(Action):
    def name(self):···

    def cutting(self,tracker,dispatcher,last_intent):···

    def run(self,dispatcher: CollectingDispatcher, tracker:Tracker ,domain: Di
        last_intent = str(tracker.get_intent_of_latest_message())

        if((last_intent=="cortante")|(last_intent=="satisfactorio")):
            self.cutting(tracker,dispatcher,str(last_intent))
        elif(last_intent=="insatisfactorio"):
            dispatcher.utter_message(text=f"que mal, como puedo solucionar esa
        else:
            dispatcher.utter_message(text=f"Hay algo en lo que pueda ayudar?")

        return []
```

Figura 26. Action *help_you*. File: *actions.py*.

2.3.4 Reuniones.

Esta categoría consta de una única storie encargada del flujo de la conversación al momento que el usuario requiera organizar una reunión. Tiene consigo dos actions.

```
- story: meeting
  steps:
  - intent: reunion
  - action: coordinate
  - intent: horario
  - action: meet
  - or:
    - intent: afirmacion
    - intent: negacion
  - action: meet
```

Figura 27. Story. File: *stories.yml*.

La action *coordinate* tiene como función principal verificar si el usuario que solicito la reunión es un usuario del cual ya hay información, de ser así ver si es profesor o alumno y en función de eso responder. En caso de que no haya información sobre el mismo, no se procederá con la organización de la reunión.

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

    def run(self,dispatcher: CollectingDispatcher, tracker:Tracker, domain: Dict[Text,Any])-> List[Dict
        message = tracker.latest_message
        id= message["metadata"]["message"]["from"]["id"]
        register = TrabajarArchivo.cargarArchivo();
        if((register=={})|(register.get(str(id),None)==None)):
            dispatcher.utter_message(text=f"che, perdon no te tengo agendado, quién sos?")
        else:
            professor = tracker.get_slot("professor")
            if(professor):
                dispatcher.utter_message(text=f"Okey, qué día y a qué hora te convendria?")
            else:
                dispatcher.utter_message(text=f"dale dale, qué día podes?")

        return[]
```

Figura 28. Action *coordinate*. File: *actions.py*.

En el caso de la action *meet*, esta accede a los datos almacenados en los slots *professor, day* y *hour*, observa si el usuario ya tiene una reunión ya programada ese mismo día y accede a los horarios cargados consultando la base de conocimientos (sección *1.2*). En el caso que ese mismo día ya tenga una reunión programada con la misma persona, se procederá a pedir otro día de reunión al usuario. Si no tiene reuniones previas, se consulta la disponibilidad horaria, se acepta la reunión o se ofrece un nuevo horario.

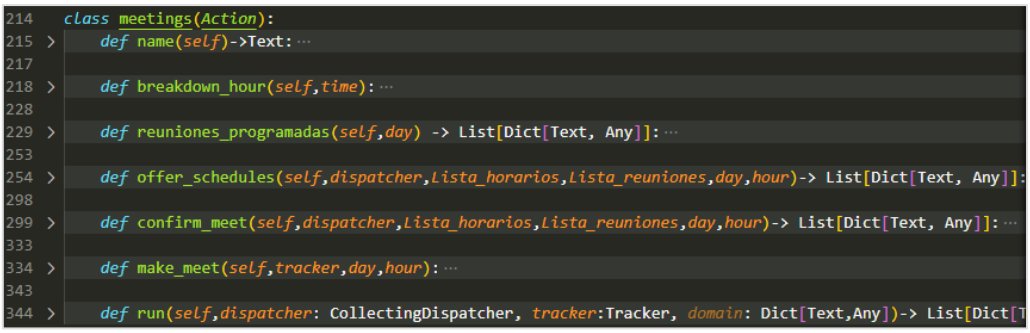

Figura 29. Action *meet*. File: *actions.py*.

2.3.5 Búsqueda de información.

La categoría *búsqueda de información* consta de una única storie y una única action.

El objetivo de esta conversación es buscar en la web información que el usuario desea o necesita saber. Para esto se implementó la utilización de las librerías *googlesearch*[15] y *Wikipedia*[16].

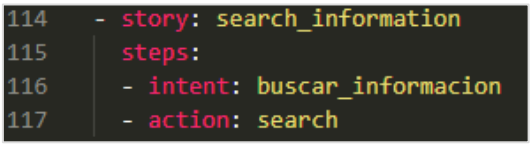

Figura 30. Storie *search*. File: *stories.yml*.

Figura 31. Action *search*. File: *actions.py*.

---

[15] Librería *googlesearch*: https://pypi.org/project/googlesearch-python/

[16] Librería *Wikipedia*: https://pypi.org/project/wikipedia/

2.3.6 Inteligencia Emocional.

La computación afectiva es un área dentro de la IA que busca mejorar la interacción entre una persona y bot, brindándole a este ultimo un diseño capaz de reconocer emociones y que condicionen su comportamiento[17]. En esta sección muestra como esta implementada esta área dentro del asistente virtual y cómo reacciona en consecuencia.

En primer lugar, el bot reconocen dos emociones *enojo* y *tristeza*, pero a su vez se le da el contexto para que reconozca que alguien está "bien", tiene en cuenta cuando una persona contesta de manera amable y cuando de una manera tajante o hiriente. También entiende tres tipos explicaciones posibles sobre el *por qué del sentimiento o la actitud.* Con un total de diez stories, siete response y una action, *Assistant bot* tiene las herramientas para actuar en las situaciones antes mencionadas.

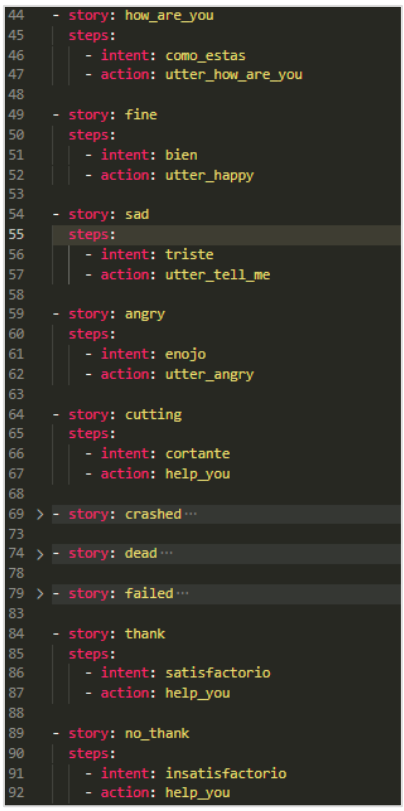

Figura 32. Stories. File: *stories.yml*.

[17] *Introducción a la computación afectiva. Universidad Nacional Experimental del Táchira*

```
87    utter_i_dont_understand:
88    - text: "Perdon, no entendí que quisiste decir..."
89
90    utter_tell_me:
91    - text: "por favor, contame por qué..."
92
93    utter_angry:
94    - text: "Perdon hay algo que te dije qué te haya molestado? Te noto un poco molesto.
95
96    utter_sympathy:
97    - text: "Noo que cagada che, lo bueno es que estas bien. Cualquier cosa que necesites
98
99    utter_compassion:
100   - text: "Realmente no sé qué decir, pero lamento mucho tu pérdida. Lo siento, esto es
101
102   utter_support:
103   - text: "Tranquilo. Haber fallado en esto no significa que seas un fracaso, solo es
```

Figura 33. Responses. File: *domain.yml*.

La action utilizada es la misma de la *figura 26*, con la diferencia que ahora se extiende su uso.

Por cada usuario que interactúa con el bot se tiene a su vez un contador con las veces en que este mismo ha respondido de manera tajante. Cuando se llega a la tercera vez con esta actitud se procede a preguntar al usuario si “sucede algo”. A su vez, si el usuario contesta de buena manera este contador disminuirá.

```
144       def cutting(self,tracker,dispatcher,last_intent):
145           message = tracker.latest_message
146           id = message["metadata"]["message"]["from"]["id"]
147           register = TrabajarArchivo.cargarArchivo()
148           nCut = int(register[str(id)]["cut"])
149           if(last_intent=="cortante"):
150               nCut+=1
151               if(nCut>=3):
152                   dispatcher.utter_message(text=f"Estas bien "+str(register[str(id)]["name"])+"? Perdon p
153                   register[str(id)]["cut"] = 0
154               else:
155                   dispatcher.utter_message(text=f"Bueno, que puedo hacer por vos?")
156                   register[str(id)]["cut"] = nCut
157           else:
158               dispatcher.utter_message(text=f"Geniall... Hay algo más en lo que pueda ayudar?")
159               if(nCut!=0):
160                   nCut=nCut-1
161                   register[str(id)]["cut"] = nCut
162
163           TrabajarArchivo.guardar(register)
```

Figura 34. Action *help_you*, método *cutting*. File: *actions.py*.

2.4 Información del Usuario.

Un asistente virtual debe saber con qué persona está hablando, con que personas habló y qué sabe hasta el momento de las mismas para así brindar un trato personalizado y más realista. Para abordar esta problemática este proyecto almacena información de cada usuario con el que habla en un archivo en formato *JSON*. La información almacenada consta de: *id de usuario, nombre, profesor (*booleano*), bot(*booleano*), cortante* (contador mencionado en la sección *2.3.6*) y *reunión (*lista con las reuniones con sus respectivos días y horarios*)*.

En la *figura 35* se muestra la clase *TrabajarArchivo* con la cual se crea el archivo en caso que aún no exista y se accede a la información que tiene almacenada utilizando el diccionario de Python[18].

```
40  class TrabajarArchivo():
41      @staticmethod
42      def guardar(Guardar):
43          with open(".\\actions\\registro","w") as archivo_descarga:
44              json.dump(Guardar, archivo_descarga)
45          archivo_descarga.close()
46
47      @staticmethod
48      def cargarArchivo():
49          if os.path.isfile(".\\actions\\registro"):
50              with open(".\\actions\\registro","r") as archivo_carga:
51                  ret=json.load(archivo_carga)
52                  archivo_carga.close()
53          else:
54              ret={}
55          return ret
```

Figura 35. File: *actions.py*.

[18] Diccionario de python: https://www.freecodecamp.org/espanol/news/compresion-de-diccionario-en-python-explicado-con-ejemplos/

2.5 Manejo de grupos.

Assistant bot no solo tiene la capacidad de hablar con una única persona, si no que puede interactuar de forma limitada en un grupo. Esto lo logra a través de condiciones explicitas en las actions, donde se detecta si el ultimo mensaje recibido es proveniente de un grupo o de un chat privado, y se toma acción al respecto. Al estar en un grupo se limita más la interacción del bot, la story *tips* ya no tendrá efecto, mientras que las stories *know_who_are_you* y *subjets*, solo funcionaran en el caso de que dentro de esa conversación grupal se haga mención al bot y el usuario comience alguna de las conversaciones pertenecientes a las historias antes mencionadas.

En la *figura 35* se puede ver en la action *subject* cómo funcionan estas condiciones. Si se quiere ver el código con más contexto en la *figura 22* se ve el código con más detalle.

```
if((str(chat)=="private")|(str(entities)=="mention")):
    prolog=Prolog()
    prolog.consult('/RASA/chat-bot Alumno/actions/reglas_nahuel.pl')
    dispatcher.utter_message(text=f"Estoy cursando:")
    for materia in prolog.query("materia_cursada(X,_,_,_)"):
        dispatcher.utter_message(text=f"- "+str(materia["X"])+"")
return[]
```

Figura 35.

2.6 Rules.

Las *reglas o rules* son un tipo de datos de entrenamiento que describen fragmentos breves de conversaciones que siempre deben seguir el mismo camino.

Dentro del chatbot aquí tratado se tiene una simple regla la cual indica que siempre que el bot interprete un *intent* de despedida, se debe proceder a terminar la conversación (*figura 36*).

```
1  version: "3.1"
2
3  rules:
4
5  - rule: Adios
6    steps:
7    - intent: despedida
8    - action: utter_goodbye
```

Figura 36. File: *rule.ympl*.

2.7 Políticas.

Rasa nos permite la utilización de políticas o policies las cuales utiliza un bot para decidir qué acción tomar en cada paso de una conversación.

Assistant bot tiene implementado, por el momento, las policies que se encuentran por defecto (*figura 37*), dentro de las cuales destacan: *Memorization Policy* y *Ted Policy*.

```
41  policies:
42  # # No configuration for policies was prov
43  # # If you'd like to customize them, uncom
44  # # See https://rasa.com/docs/rasa/policie
45
46  #D:\anaconda\envs\installingrasa\Lib\site-
47  #  - name: "test_policy.TestPolicy"
48  #    priority: 1
49    - name: MemoizationPolicy
50    - name: RulePolicy
51    - name: UnexpecTEDIntentPolicy
52      max_history: 5
53      epochs: 100
54    - name: TEDPolicy
55      max_history: 5
56      epochs: 100
57      constrain_similarities: true
```

Figura 37. Policies. File: *config.yml*.

3. Mejoras y soluciones.

El chatbot presentado en este informe no es en una versión final, por ende, se encuentra en una etapa de desarrollo y esta abierto a posibles mejoras.

3.1 Agregar Policy personalizada.

Al trabajar con varias stories, es muy posible que en alguna de ellas se repita algún *intent* (ejemplo: *afirmación, negación, etc*) o se trabajen sobre expresiones muy parecidas. Por lo tanto, es necesario implementar una policy que se encargue de estas situaciones y decida qué acción tomar.

Las condiciones impuestas dentro de las *actions* para la comunicación grupal e individual del asistente funcionan correctamente, pero lo optimo seria tener este tipo de condiciones dentro de una policy.

3.2 Almacenamiento de información.

Dentro del proyecto toda la información recolectada por el chatbot se almacena en un archivo, esto sugiere un problema de escalabilidad, ya que, si se opta por almacenar más información y el bot comienza a interactuar con más usuarios, el archivo se vuelve ineficiente. Para solucionar este inconveniente se podría considerar la utilización de una base de datos.

4. Conclusiones.

Assitant bot es un asistente virtual con un gran potencial, demostró un correcto funcionamiento y buena fluidez en conversaciones cortas individuales y grupales. Las mejoras planteadas en la sección anterior potenciarían su eficacia a la hora de hablar y ser un intermediario capaz de organizar una agenda sin ningún tipo de problema.