AGENTE INTELIGENTE PARA TELEGRAM: FRAMEWORK RASA

Utilizando Machine Learning

Alumno: Ferrero Adriel Ram

Materia: Programación Exploratoria

Profesora: Analía Amandi

UNICEN - Tandil

24/10/2022

INTRODUCCIÓN

En este informe, se muestra y explica el desarrollo de un agente inteligente, implementado utilizando la herramienta RASA. RASA es un framework que permite crear agentes conversacionales utilizando un tipo de Inteligencia Artificial, Machine Learning, y comprensión de lenguaje natural (NLU).
El agente fue llevado a practica en Telegram, una app de mensajería, que permite el uso de su API de manera gratuita. El mismo, fue basado en el autor de este documento, teniendo en cuenta aspectos tales como la personalidad, forma de hablar, entre otros.

DESARROLLO

La función básica del agente, es lograr una comunicación un tanto fluida, y que logre procesar el contexto de la situación. Para lograr esto, se utilizaron slots, archivos de formato json, y distintos lenguajes de programación, como Python.

Para utilizar el reconocimiento de lenguaje natural, se implementan distintos 'intents', utilizando ejemplos, para el entrenamiento de machine Learning, de buena calidad. Esto se logra, otorgando variedad en cuanto al léxico. Por ejemplo, para el intent saludo, se tiene en cuenta las distintas maneras que el usuario pueda llegar a expresarse, tanto formal como informal.

Imagen de perfil del agente en la plataforma Telegram. Generada utilizando Dall-E

```
- intent: saludo
  examples: |
    - hola
    - buenas
    - buen dia
    - buenas tardes
    - buenas noches
    - hola adriel

- intent: affirm
  examples: |
    - si
    - se
    - tal cual
    - por supuesto
    - suena bien
    - correcto
    - obvio
    - bueno

- intent: deny
  examples: |
    - no!
    - no mucho
    - nunca
    - no lo creo
    - no me gusta
    - para nada
    - nop
```

Para lograr que el agente sea capaz de tener una conversación coherente, debe poder recordar información sobre el usuario, como podría serlo, su estado de ánimo.

```
- intent: mood_triste
  examples: |
    - estoy triste
    - estoy mal [vos](bot_mood_asked)
    - no estoy bien
    - estoy masomenos
    - me siento mal
    - me siento triste
    - no tengo un buen dia [vos como estas]{"entity":"bot_mood_asked","value":"vos"}
    - yo mal

- intent: mood_feliz
  examples: |
    - estoy bien [vos](bot_mood_asked)
    - muy bien
    - yo? bien [vos como estas]{"entity":"bot_mood_asked","value":"vos"}
    - bastante bien [y vos]{"entity":"bot_mood_asked","value":"vos"}
    - yo bien
    - estoy feliz
```

```
- regex: bot_mood_asked
  examples: |
    - ((([aA-zZ]\w+ *){1,})+(, )*){1,}
```

Si el usuario expresa que esta ‘feliz’ o ‘triste’, el agente lo detecta utilizando estos intents. Luego, el agente procederá a almacenar esa información.

```
- story: sad
  steps:
  - intent: mood_triste
  - action: action_user_mood
  - action: utter_cheer_up
  - action: action_bot_mood_asked

- story: feliz
  steps:
  - intent: mood_feliz
  - action: action_user_mood
  - action: utter_feliz
  - action: action_bot_mood_asked
```

En la imagen se observa el llamado a la action ‘action_user_mood’, donde se almacena la respectiva información.

```python
class MoodUsuarioDado(Action):
    def name(self) -> Text:
        return "action_user_mood"
    def run(self, dispatcher: CollectingDispatcher,tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            #no funciona en grupos
            intent =  tracker.latest_message['intent'].get('name')
            if intent == 'mood_feliz' or 'mood_triste':
                return [SlotSet("user_mood_set",True)]
            else:
                return [SlotSet("user_mood_set",False)]
```

```yaml
- story: como estas
  steps:
  - intent: como_estas
  - slot_was_set:
    - user_mood_set: true
  - action: utter_como_estoy

- story: como estas + vos
  steps:
  - intent: como_estas
  - slot_was_set:
    - user_mood_set: false
  - action: utter_como_estoy_y_vos
```

Para lograr un mejor funcionamiento, se implementa la posibilidad de que el usuario, además de expresar esta información, pregunte sobre el agente, en un mismo mensaje. Para esto, se utiliza una entity de tipo bool, y un slot para guardar esa información durante toda la conversación (como se observa en imagen de intents: mood_feliz y mood_triste). Se observan contempladas distintas maneras que pueda preguntar sobre el bot. Se encuentra la action ‘action_bot_mood_asked’ al final de las respectivas stories (‘sad’ y ‘feliz), que verifican si el usuario consulto sobre el estado de ánimo o no.

```python
class MoodUsuarioPregMood(Action):
    def name(self) -> Text:
        return "action_bot_mood_asked"
    def run(self, dispatcher: CollectingDispatcher,tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            asked= next(tracker.get_latest_entity_values("bot_mood_asked"), None)
            if asked != None:
                dispatcher.utter_message(response = "utter_como_estoy")
```

```yaml
utter_como_estoy:
- condition:
  - type: slot
    name: logged_in
    value: true
  text: "Muy bien {name}!"
- condition:
  - type: slot
    name: logged_in
    value: true
  text: "Bien {name}!"
- text: "Bien!"
- text: "Muy bien!"
```

Esta función logra que el usuario experimente una conversación mas real, tendiendo a sentirse más cómodo. En una situación donde el usuario comparte sobre su estado de ánimo, y la otra persona no lo recuerda, aumentan las probabilidades de que empeore la experiencia para el mismo.
Como se observa en ‘utter_como_estoy’ se realiza un trato personal, respondiendo con el nombre del usuario, guardado en slot ‘name’.

Para obtener el nombre, se implementa lo siguiente.

```python
class ActionSessionStart(Action):
    def name(self) -> Text:
        return "action_session_start"
    async def run(
      self, dispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        #print(tracker.latest_message["metadata"])
        print('hola')
        #Si es grupo:
        if (tracker.latest_message["metadata"]["message"]["chat"]["type"] == 'group') or (tracker.latest_message["metadata"]["message"]["chat"]["type"] == 'supergroup'):
            chat_id = tracker.latest_message["metadata"]["message"]["chat"]["id"]
            N = TelegramAPI.getChatMemberCount(chat_id)
            cantidad = N
            print('cantidad:',cantidad)
            #slot CantUsersConfirmed se inicia en 0 (aclarado en domain.yml)
            return [SessionStarted(), ActionExecuted("action_listen"),SlotSet("is_group",True),SlotSet("CantUsers",cantidad)]
        else:
```

```
else:
    print('saru')
    conocidos = OperarArchivo.cargarArchivo()
    telegram_user_id = tracker.latest_message["metadata"]["message"]["from"]["id"]
    if telegram_user_id in conocidos:
        conocidos[telegram_user_id]['first_time'] = False
        if conocidos[telegram_user_id]['name_set'] == True :
            nombre = conocidos[telegram_user_id]['name']
        else:
            nombre = tracker.latest_message["metadata"]["message"]["from"]["first_name"]
    else:
        print('ola')
        conocidos[telegram_user_id] = {}
        conocidos[telegram_user_id]['name_set'] = False
        conocidos[telegram_user_id]['first_time'] = True
        nombre = tracker.latest_message["metadata"]["message"]["from"]["first_name"]
    OperarArchivo.guardar(conocidos)
    return [SessionStarted(), ActionExecuted("action_listen"),SlotSet("is_group",False),SlotSet("name",nombre),SlotSet("logged_in",True),SlotSet("user_mood_set",False)]
    # Ademas SlotSet de logged_in, y user_mood_set
```

Esta action, es una action default de RASA, que es llamada al comienzo de la conversación/sesión. Esto permite poder recaudar información sobre la misma, una primera vez, que sea utilizada para el resto de la conversación.

Una de las funciones de esta action es corroborar si es la primera vez que se habla con esa persona. Para esto, si la conversación no es un grupo, obtiene el id del chat, mediante la metadata de Telegram. Luego, busca ese id en el archivo ‘conocidos. Json’. Si no está, lo agrega, con los campos ‘name_set’ en false, haciendo referencia a que nunca dijo su nombre, y ‘first_time’ en true, ya que es la primera vez que habla con el agente. En cuanto al nombre, en el slot ‘name’ almacena el nombre que tenga el usuario en su perfil de Telegram.

En caso el usuario se encuentre en el archivo, si el bool ‘name_set’ se encontraba en True, el usuario ya se había presentado anteriormente, y el bot utilizara ese nombre para hablarle. Esto es conveniente, ya que el usuario podría preferir que lo llame de cierta manera, y no con el nombre de usuario de Telegram.

Esto permite el bot pueda tener un trato personal, mas allá el usuario se haya presentado, pero también ofrece esa posibilidad.

```
- rule: presentacion
  steps:
  - intent: presentacion_intent
  - action: action_presentacion
  - action: utter_un_placer
```

```
- intent: presentacion_intent
  examples: |
    - soy [adriel](nombre)
    - mi nombre es [Analia](nombre)
    - me llamo [estani](nombre)
    - yo soy [Santi](nombre)
    - me podes llamar [nelson](nombre)
```

Se utiliza una entidad ‘nombre’, extraída utilizando Lookup Tables.

```
- lookup: nombre
  examples: |
    - matias
    - eduardo
    - santiago
    - Santi
    - nelson
    - analia
    - mati
    - edu
    - adriel
```

```
    - Maria
    - marithe
    - estani
    - adri
    - paqui
    - estanislao
    - nahue
    - nahuel
    - luca
    - lucas
    - valen
    - valentin
```

```
pipeline:
# No configuration for the NLU pipeline was provided. The following default pipeline was used to train your model.
# If you'd like to customize it, uncomment and adjust the pipeline.
# See https://rasa.com/docs/rasa/tuning-your-model for more information.
  - name: WhitespaceTokenizer
  - name: RegexFeaturizer
  - name: LexicalSyntacticFeaturizer
  - name: CountVectorsFeaturizer
  - name: CountVectorsFeaturizer
    analyzer: char_wb
    min_ngram: 1
    max_ngram: 4
  - name: RegexEntityExtractor
    case_sensitive: False
    use_lookup_tables: True
    use_regexes: True
    use_word_boundaries: True
  - name: "DucklingEntityExtractor"
    # url of the running duckling server
    url: "http://localhost:8000"
    # dimensions to extract
    dimensions: ["time"]
    # allows you to configure the locale, by default the language is
    # used
    locale: "es_ES"
    # if not set the default timezone of Duckling is going to be used
    # needed to calculate dates from relative expressions like "tomorrow"
    timezone: "America/Argentina/Buenos_Aires"
    # Timeout for receiving response from http url of the running duckling server
    # if not set the default timeout of duckling http url is set to 3 seconds.
    timeout : 3
  - name: DIETClassifier
    epochs: 100
    constrain_similarities: true
  - name: EntitySynonymMapper
  - name: ResponseSelector
    epochs: 100
    constrain_similarities: true
  - name: FallbackClassifier
    threshold: 0.7
    ambiguity_threshold: 0.1
```

Se agrega el componente RegexEntityExtractor, para lograr extraer el nombre del mensaje, utilizando la LookupTable.

```python
class ActionPresentacion(Action):
    #no implementado para grupos
  def name(self) -> Text:
      return "action_presentacion"
  def run(self, dispatcher: CollectingDispatcher,tracker: Tracker,
          domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
           nombre = next(tracker.get_latest_entity_values("nombre"), None)
           grupo = tracker.get_slot("is_group")
           if not grupo:
               if nombre != None:
                   telegram_user_id = tracker.latest_message["metadata"]["message"]["from"]["id"]
                   conocidos = OperarArchivo.cargarArchivo()
                   if telegram_user_id in conocidos:
                   #nada mas para evitar errores, pero supuse el id ya se encuentra en "conocidos"
                       conocidos[telegram_user_id]['name_set']= True
                       #si ya habia dicho su nombre, entonces name_set ya estaba en true, pero lo reescribo,al igual que el nombre.
                       conocidos[telegram_user_id]['name']= nombre
                       OperarArchivo.guardar(conocidos)
                   return [SlotSet("name",nombre),SlotSet("logged_in",True)]
```

Esta custom action guarda el nombre con el que se presenta el usuario, en el archivo json. Además, actualiza el slot ‘name’ y ‘logged_in’.

```yaml
utter_un_placer:
  - condition:
    - type: slot
      name: logged_in
      value: true
    text: "Un placer {name}!"
  - text: "Un placer!"
```

Comenzando la conversación, si el usuario saluda, el agente responderá con un primer mensaje de bienvenida.

```yaml
- rule: Saludar por primera vez
  conversation_start: true
  steps:
  - intent: saludo
  - action: action_primer_saludo

- rule: Saludar el resto de veces
  conversation_start: false
  steps:
  - intent: saludo
  - action: utter_hola
```

Si la conversación ya había comenzado anteriormente, no responde con un mensaje de bienvenida.

```yaml
utter_hola:
- condition:
    - type: slot
      name: logged_in
      value: true
  text: "Hola {name}"
- text: "Hola"
```

En caso sea el inicio de la conversación, se llama a la action ‘action_primer_saludo’

```python
class PrimerSaludo(Action):
    def name(self) -> Text:
        return "action_primer_saludo"
    def run(self, dispatcher: CollectingDispatcher,tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            # no funciona en grupos
            grupo = tracker.get_slot("is_group")
            if not grupo:
                intent =  tracker.latest_message['intent'].get('name')
                telegram_user_id = tracker.latest_message["metadata"]["message"]["from"]["id"]
                conocidos = OperarArchivo.cargarArchivo()
                nombre = tracker.get_slot("name")
                if telegram_user_id in conocidos:
                    if conocidos[telegram_user_id]['first_time'] == False:
                        dispatcher.utter_message(text=f"Hola denuevo {nombre}! como estas?")
                    else:
                        dispatcher.utter_message(text=f"Hola {nombre}! Soy Rambot, el asistente virtual de Adriel Ferrero")
                OperarArchivo.guardar(conocidos)
```

En esta action, el agente difiere en su respuesta, dependiendo si es la primera vez que habla con el usuario, o ya habían tenido una conversación. En este último caso, el agente responde ‘Hola denuevo’ agregando el nombre. Caso contrario, da las bienvenidas y se presenta.

Es un hecho que el agente no logra cubrir todo tipo de conversaciones, pero de no hacerlo, debe ser capaz de expresarlo. Para esto, en caso que no se detecte ningún intent, responde con ‘utter_fallback’

```yaml
- rule: fallback behaviour simple
  steps:
  - intent: nlu_fallback
  - action: utter_fallback
```

```
utter_fallback:
- text: "perdon, no te entendí"
```

Finalizando la conversación, el usuario se despide, por lo que el agente también se despedirá

```
- rule: Despedida
  steps:
  - intent: despedida_intent
  - action: utter_despedida
```

```
utter_despedida:
- condition:
    - type: slot
      name: logged_in
      value: true
  text: "chau {name}!"
- condition:
    - type: slot
      name: logged_in
      value: true
  text: "Nos vemos {name}!"
- condition:
    - type: slot
      name: logged_in
      value: true
  text: "Suerte {name}!"
- condition:
    - type: slot
      name: logged_in
      value: true
  text: "Que te vaya bien {name}!"
- text: "chau!"
- text: "Nos vemos!"
- text: "Suerte!"
- text: "Que te vaya bien!"
```

El caso el usuario interactúe con el agente preguntándole sobre la carrera, se contempla con las siguientes stories

```
- story: materias cursando
  steps:
  - intent: materias_cursando
  - action: action_cursando
```

Donde el intent es

```
- intent: materias_cursando
  examples: |
    - estas cursando alguna materia?
    - que materias estas cursando?
    - en que materias estas?
    - que estas cursando?
```

```python
class ActionCursando(Action):
    def name(self) -> Text:
        return "action_cursando"
    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text,Any]) -> List[Dict[Text,Any]]:
        with PrologMQI(port=8000) as mqi:
            with mqi.create_thread() as prolog_thread:
                prolog_thread.query_async("consult('C:/Rasa_Projects/rasa_project_nuevo/actions/base_datos_prolog.pl')", find_all=False)
                prolog_thread.query_async(f"cursando(X).", find_all=False)
                result = prolog_thread.query_async_result()
                # me devuelve una lista de python, con 1 elemento "X" que es otra lista, con todas las materias q curso
                # otra manera seria, que no esten en una lista, sino que sean hechos, y en python itero, y devuelve uno por uno.
                aList = result[0]
                if len(aList['X']) > 0:
                    Laux1 = str(aList['X'][0])
                    prolog_thread.query_async(f"materia({Laux1},Z,_,_)")
                    result = prolog_thread.query_async_result()
                    print(result)
                    cursando = str(result[0]['Z'])
                    for i in range(1,len(aList['X'])):
                        Laux2 = str(aList['X'][i])
                        prolog_thread.query_async(f"materia({Laux2},Y,_,_)")
                        result = prolog_thread.query_async_result()
                        materiaaux = str(result[0]['Y'])
                        cursando = cursando + ', ' + materiaaux
                    dispatcher.utter_message(text=f'{"estoy cursando " + cursando}')
                else:
                        dispatcher.utter_message(text=f'{"no estoy cursando nada"}')
```

Esta action utiliza Prolog, y una base de datos, donde se encuentran las materias, con sus respectivos códigos, nombres, cuatrimestre y año, representadas como hechos. Además, una lista ‘cursando’ donde figuran los códigos de las materias en las que el autor está actualmente inscripto.

Python toma la lista, y por cada materia que se esta cursando, llama a prolog, con el predicado materia, pasando ese código, y devuelve el nombre. El String cursando se va modificando, agregando cada materia. Finalmente, responde al usuario que materias

está cursando.

```
materia(6111,"Introduccion a la programacion I",1,1).
materia(6112,"Analisis Matematico I",1,1).
materia(6113,"Algebra I",1,1).
materia(6114,"Quimica",1,1).
materia(6121,"Ciencias de la Computacion I",1,2).
materia(6122,"Introduccion a la Programacion II",1,2).
materia(6123,"Algebra Lineal",1,2).
materia(6124,"Fisica General",1,2).
materia(6125,"Matematica Discreta",1,2).
materia(6211,"Ciencias de la Computacion II",2,1).
materia(6221,"Analisis y disenio de algoritmos II",2,2).
materia(6225,"Ingles",2,2).
materia(6224,"Eletronica Digital",2,2).
materia(6223,"Probabilidad y Estadistica",2,2).
materia(6222,"Comunicacion de Datos I",2,2).
materia(6321,"Programacion exploratoria",3,2).
```

```
cursando([6221,6222,6223,6224,6225,6321]).
```

Se tiene en cuenta otro flujo de conversación, donde, además, el usuario quiera saber que materias de un año en específico, está cursando.

```
- story: en que año estas
  steps:
  - intent: estado_carrera
  - action: utter_estado_carrera
  - intent: carrera_year
  - action: utter_carrera_year
  - intent: materia_cursando_year
  - action: action_cursando_year
```

```
- intent: estado_carrera
  examples: |
    - como venis con la carrera?
    - estas atrasado?
    - te atraso alguna materia?
    - venis bien con la carrera?
```

```
utter_estado_carrera:
- text: " por ahora vengo bien, vengo al dia."
```

```
- intent: carrera_year
  examples: |
    - en que año estas?
    - por que año vas?
    - que año estas cursando?
    - de que año sos?
```

```
 utter_carrera_year:
 - text: "Estoy en 2do, pero cursando una materia de tercero"
```

```
- intent: materia_cursando_year
  examples: |
    - que materias de [segundo]{"entity":"year","value":"2"} estas cursando?
    - en que materias de [tercero]{"entity":"year","value":"3"} estas?
    - estas cursando alguna materia de [primero]{"entity":"year","value":"1"}?
    - ya estas cursando una materia de [cuarto]{"entity":"year","value":"4"}?
    - de [quinto]{"entity":"year","value":"5"} cursas alguna?
```

Como se observa en el código, se utiliza una entity ‘year’ que se extrae utilizando el componente DIETclassifer.

Luego, RASA procede a llamar a ‘action_cursando_year’, donde por cada materia cursando, además del código, se le pasa a prolog el año del plan de estudios al que debe pertenecer esa materia.

```python
class ActionCursandoYear(Action):
    def name(self) -> Text:
        return "action_cursando_year"
    def run(self, dispatcher: CollectingDispatcher,
            tracker: Tracker,
            domain: Dict[Text,Any]) -> List[Dict[Text,Any]]:
            year = next(tracker.get_latest_entity_values("year"), None)
            with PrologMQI(port=8000) as mqi:
                with mqi.create_thread() as prolog_thread:
                    prolog_thread.query_async("consult('C:/Rasa_Projects/rasa_project_nuevo/actions/base_datos_prolog.pl')", find_all=False)
                    prolog_thread.query_async(f"cursando(X).", find_all=False)
                    result = prolog_thread.query_async_result()
                    # me devuelve una lista de python, con 1 elemento "X" que es otra lista, con todas las materias q curso
                    # otra manera seria, mas facil en python, pero mas difcil en prolog, q devuelva por backtracking 1 x 1
                    aList = result[0]
                    if len(aList['X']) > 0:
                        cursando ="estoy cursando "
                        Laux1 = str(aList['X'][0])
                        c = 0       #lo uso despues, para que la primera materia q imprima no le ponga la ','
                        for i in range(0,len(aList['X'])):
                            Laux2 = str(aList['X'][i])
                            prolog_thread.query_async(f"materia({Laux2},Y,{year},_)")
                            result = prolog_thread.query_async_result()
                            if result != False:
                                materiaaux = str(result[0]['Y'])
                                if c != 1:
                                    cursando = cursando +' ' + materiaaux
                                    c = 1
                                else:
                                    cursando = cursando + ',' + materiaaux
                        if cursando == "estoy cursando ":
                                dispatcher.utter_message(text=f'{"no estoy cursando ninguna de ese año"}')
                        else:
                            dispatcher.utter_message(text=f'{cursando}')
                    else:
                            dispatcher.utter_message(text=f'{"no estoy cursando ninguna"}')
```

Organización de reuniones grupales

El agente conversacional es capaz de comunicarse en un grupo, donde se organice una reunión a futuro. Para esto, analiza, detecta, evalúa y pide un horario.

Con el fin de que los integrantes del grupo puedan hablar de manera flexible o cómoda, para que el agente logre detectar los horarios correctamente, se utilizó el componente DucklingEntityExtractor, como puede observarse en el pipeline.

```yaml
- name: "DucklingEntityExtractor"
  # url of the running duckling server
  url: "http://localhost:8000"
  # dimensions to extract
  dimensions: ["time"]
  # allows you to configure the locale, by default the language is
  # used
  locale: "es_ES"
  # if not set the default timezone of Duckling is going to be used
  # needed to calculate dates from relative expressions like "tomorrow"
  timezone: "America/Argentina/Buenos_Aires"
  # Timeout for receiving response from http url of the running duckling server
  # if not set the default timeout of duckling http url is set to 3 seconds.
  timeout : 3
```

Para guardar la información obtenida de este EntityExtractor se utilizan los slots mes, día y hora.

```
  hora:
    type: text
    mappings:
    - type: custom
      influence_conversation: true
      conditions:
      - active_loop: evento_form
  dia:
    type: text
    mappings:
    - type: custom
      influence_conversation: true
      conditions:
      - active_loop: evento_form
  mes:
    type: text
    mappings:
    - type: custom
      influence_conversation: true
      conditions:
      - active_loop: evento_form
```

Con el fin de detectar una propuesta, o afirmación de disponibilidad, se crea el intent ‘propone_reunion’.

```
- intent: propone_reunion
  examples: |
    - Les parece hacer un meet?
    - Quieren hacer meet el miercoles?
    - Si quieren nos reunimos a las 20 horas
    - nos reunimos el viernes a las 15?
    - pueden meet mañana?
    - podemos reunirnos hoy
    - Puedo reunirme el martes 18
    - estoy libre a las 16
    - bueno, te parece bien el viernes?
```

```
- time
```

La entidad ‘time’ es extraída, por el DucklingEntityExtractor automáticamente. Este extractor no requiere de señalización de la entity explícitamente, solo que la entidad se llame ‘time’.

Una particularidad del extractor, es que devuelve un campo ‘grain’ que indica que se detecta, pudiendo ser: mes, día, hora, o minuto. Además, solo devuelve uno de estos, priorizando el más acotado. Por ejemplo, ante un mensaje “¿pueden reunirse el viernes?” en ‘grain’ se encontrará ‘day’, ante un mensaje “¿pueden reunirse en diciembre?” se encontrará ‘month’, y ante un mensaje “¿pueden reunirse a las 15?” se encontrará ‘hour’. Esta detección se complejiza bastante, cuando se especifica dos de estos slots. Por ejemplo: “¿pueden reunirse mañana a las 15?” se especifica tanto día como hora, pero en el campo ‘grain’ se encontrará ‘hour’. En el caso que se especifique tanto mes como día, el campo ‘grain’ se llena con ‘day’. Se observa un orden jerárquico de prioridad, donde:

- Hora
- Dia
- Mes

El valor de el campo ‘grain’ hace referencia al mas alto, si se especificó en el mensaje. Teniendo esto en cuenta, el agente asume que, si el usuario especificó hora, o bien también especificó día y mes, o bien no lo hizo y se refiere a la mas próxima. Por ejemplo: “¿quieren hacer un meet a las 10?” en este caso, el bot asume se refiere a las 10 am próxima, podría ser las 10 am del mismo día (en caso no haya pasado ese horario) o las 10 am del día siguiente. El DucklingEntityExtractor asume, al no especificar slots de menor nivel jerárquico, que se refiere al próximo horario, cronológicamente.

Retomando la action ‘action_session_start’, se muestra la posibilidad de que se esté hablando en un grupo, en cuyo caso el slot ‘is_group’ será True y se obtendrá la cantidad de integrantes del grupo

```python
class ActionSessionStart(Action):
    def name(self) -> Text:
        return "action_session_start"
    async def run(
      self, dispatcher, tracker: Tracker, domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
        #print(tracker.latest_message["metadata"])
        print('hola')
        #Si es grupo:
        if (tracker.latest_message["metadata"]["message"]["chat"]["type"] == 'group') or (tracker.latest_message["metadata"]["message"]["chat"]["type"] == 'supergroup'):
            chat_id = tracker.latest_message["metadata"]["message"]["chat"]["id"]
            N = TelegramAPI.getChatMemberCount(chat_id)
            cantidad = N
            print('cantidad:',cantidad)
            #slot CantUsersConfirmed se inicia en 0 (aclarado en domain.yml)
            return [SessionStarted(), ActionExecuted("action_listen"),SlotSet("is_group",True),SlotSet("CantUsers",cantidad)]
        else:
```

Para obtener la cantidad de integrantes, se utiliza un request a la API de Telegram.

```python
class TelegramAPI():

    @staticmethod
    def getMe():
        request = requests.get('https://api.telegram.org/bot5670991553:AAGwyyMvJC4LaWRk7oCDUr3w3Aj5Fz39W0E/getMe').json()  #llamar al api
        if request['ok'] == False:
            print('Hubo un error...')
            print(request['description'])
        else:
        # Llamado a api de telegram, correcto
            return request['result']
    # utilizado para prueba.
    @staticmethod
    def getChatMemberCount(chat_id):
        request = requests.get('https://api.telegram.org/bot5670991553:AAGwyyMvJC4LaWRk7oCDUr3w3Aj5Fz39W0E/getChatMemberCount?chat_id={chat_id}').json()
        if request['ok'] == False:
            print('Hubo un error...')
            print(request['description'])
        else:
        # Llamado a api de telegram, correcto
            return request['result']
```

Donde la request http se crea con el token del bot de Telegram, el método ‘getChatMemberCount’ y el parámetro, la id del chat.
Esta información es relevante ya que, como se explicará posteriormente, para finalmente crear la reunión, la mayoría de los integrantes (n/2 +1) debe de haber confirmado su disponibilidad.

La reunión organizada, será agendada en el calendario de Google, pero para esto, primero el agente debe garantizar que posee toda la información requerida. RASA brinda Forms para casos como este, donde el form se encarga de pedir al usuario, la información requerida, hasta que esta sea brindada.

```yaml
- rule: activo form evento
  condition:
  - slot_was_set:
    - is_group: true
  steps:
  - intent: propone_reunion
  - action: evento_form
  - active_loop: evento_form

- rule: Completa form
  condition:
  - active_loop: evento_form
  - slot_was_set:
    - is_group: true
  steps:
  - action: evento_form
  - active_loop: null
  - slot_was_set:
    - requested_slot: null
  - action: action_crea_evento
```

Utilizando la regla ‘activo form evento’, al detectar una propuesta para una reunión, se Inicia el form. La regla ‘completa form’ contempla la finalización de un form, que ocurre si todos los slots del mismo, han sido obtenidos.

```yaml
forms:
  evento_form:
    required_slots:
      - mes
      - dia
      - hora
      - UsersConfirmed
```

```yaml
- story: inicio form evento
  steps:
  - slot_was_set:
    - is_group: true
  - intent: propone_reunion
  - action: evento_form
  - active_loop: evento_form
  - slot_was_set:
    - requested_slot: mes
  #extract_slot
  - slot_was_set:
    - mes: mayo
  - slot_was_set:
    - requested_slot: dia
  #extract_slot
  - slot_was_set:
    - dia: 10
  - slot_was_set:
    - requested_slot: hora
  #extract_slot
  - slot_was_set:
    - hora: 15
  # se valida horario solo (form validate) y confirma si estoy ocupado o no
  - slot_was_set:
    - requested_slot: UsersConfirmed
  #el form automaticamente valida UsersConfirmed como corresponde
  - slot_was_set:
    - UsersConfirmed: 1
  # la respuesta del form al no haberse seteado el slot es action_listen
  - slot_was_set:
    - requested_slot: null
  - active_loop: null
  - action: action_crea_evento
```

En esta story, se contemplan los slots que se deben requerir, y si han sido obtenidos. Internamente, el form actualiza el ‘requested_slot’ actual cada vez se mapee el slot. Con el fin de asegurarse que los slots sean llenados en el orden jerárquico ya establecido (inversamente, es decir de inferior a superior), los slots pueden influenciar la conversación, por lo que en el dominio figura ‘influence_conversation = True’

```
hora:
  type: text
  mappings:
  - type: custom
    influence_conversation: true
    conditions:
    - active_loop: evento_form
dia:
  type: text
  mappings:
  - type: custom
    influence_conversation: true
    conditions:
    - active_loop: evento_form
mes:
  type: text
  mappings:
  - type: custom
    influence_conversation: true
    conditions:
    - active_loop: evento_form
```

Además de estos slots, como puede observarse en la story del form, se utiliza el slot CantUsersConfirmed, logrando asegurar que la mayoría de los integrantes confirmen la reunión, antes de completar el form.

```
UsersConfirmed:
  type: text
  influence_conversation: true
  mappings:
  - type: custom
    condition:
    - active_loop: evento_form
```

Este valor se actualizará conforme a los integrantes confirmen si están disponibles.

```
- story: usuario confirma
  steps:
  - slot_was_set:
    - is_group: true
  - active_loop: evento_form
  - intent: affirm
  - action: action_usuario_puede
```

Siendo la action

```
class UserConfirmaEvento(Action):
    def name(self) -> Text:
        return "action_usuario_puede"
    def run(self, dispatcher: CollectingDispatcher,tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            confirmados = tracker.get_slot("CantUsersConfirmed") + 1
            print('confirmados:',confirmados)
            return (SlotSet("CantUsersConfirmed",confirmados))
```

En el caso que algún integrante no pueda en ese horario, se reinician todos los slots

```
- story: usuario niega
  steps:
  - slot_was_set:
    - is_group: true
  - active_loop: evento_form
  - intent: deny
  - action: action_reset_event_slots
```

```
class ReseteaSlotsFormEvento(Action):
    def name(self) -> Text:
        return "action_reset_event_slots"
    def run(self, dispatcher: CollectingDispatcher,tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            # resetea slots: mes,dia,hora y UsersConfirmed
            print('Slots del form reseteados!')
            return (SlotSet("mes",None),SlotSet("dia",None),SlotSet("hora",None),SlotSet("UsersConfirmed",None))
```

El form, automáticamente, se encarga de mapear y validar los slots. Esto lo logra utilizando las actions correspondientes.
Los slots mes y día no requieren de validación, pues solo validará el slot hora, ya que cuando se llene el mismo, también se tiene mes y día.
El form llamara a ‘validate_hora’ una vez este slot haya sido extraído.

```python
class ValidarForm(FormValidationAction):
    #slot_value es el valor que tomara el slot, luego de validarlo (valor temporal, se valida antes de mapearlo al slot)
    def name(self) -> Text:
        return "validate_evento_form"
    def validate_hora(
        self,
        slot_value: Any,
        dispatcher: CollectingDispatcher,
        tracker: Tracker,
        domain: DomainDict,
    ) -> Dict[Text, Any]:
            #por como implemente el mapeado de horario, se que cuando se llene el slot hora, ya tengo todos los slots.
            # es valido si: en mi calendario de google no hay nada q se interponga.
            # tiene en cuenta si los eventos son de mas de 1 hora ( suponiendo eventos en el calendario de mas de 1 hora,
            #   ,agregados por un humano, ya que el bot agrega solo de 1 hora)
            mes = tracker.get_slot("mes")
            dia = tracker.get_slot("dia")
            hora = slot_value
            creds = None
            if (mes == None) or (dia == None):
                SCOPES = ['https://www.googleapis.com/auth/calendar.readonly']#calendar.readonly porq solo voy a leer
                # el archivo token.json almacena el acceso del usuario y actualiza tokens,
                #  es creada automaticamente cuando la actualizacion del flow se completa
                # por primera vez
                if os.path.exists('token.json'):
                    creds = Credentials.from_authorized_user_file('token.json', SCOPES)
                # si no hay credenciales validas disponibles, el usuario loguea.
                if not creds or not creds.valid:
                    if creds and creds.expired and creds.refresh_token:
                        creds.refresh(Request())
                    else:
                        flow = InstalledAppFlow.from_client_secrets_file(
                            ".\\actions\\credentials.json", SCOPES)
                        creds = flow.run_local_server(port=0)
                    # Guarda las credenciales para la proxima ejecucion
                    with open('token.json', 'w') as token:
                        token.write(creds.to_json())

                try:#bloque que si no da error, lo ejecuta
                    service = build('calendar', 'v3', credentials=creds)
```

```python
try:#bloque que si no da error, lo ejecuta
    service = build('calendar', 'v3', credentials=creds)

    horariomin = datetime.datetime(2022,mes,dia,00,00).isoformat()
    horariomax = datetime.datetime(2022,mes,dia,23,59).isoformat()
    print('Mis horarios ocupados son:')
    events_result = service.events().list(calendarId='d5f9f6052fffdb67cd18405f31e9bb9ab028678ce3128484d2a8239a51f49bc8@group.calendar.google.com', timeMax=horariomax, timeMin=horariomin,
                                          maxResults=1000, singleEvents=True,#1000 EVENTOS PROXIMOS, que sobre.
                                          orderBy='startTime').execute()
    events = events_result.get('items', [])
```

```
            if not events:
                print('No hay eventos')
                dispatcher.utter_message(text=f"Yo puedo")
                return{"hora":slot_value}

            Puedo = True
            for event in events:
                if Puedo == True:
                    # para cada evento fijarme si interviene con el horario dado
                    inicial = event['start'].get('dateTime')
                    final = event['end'].get('dateTime')
                    # de todos modos, solo crea eventos de 1 hora.
                    if inicial < hora :
                        if final > hora:
                            Puedo = False
                    elif inicial > hora :
                        if inicial < (hora + 1) :
                            Puedo = False
                    else:
                        #inicia al mismo tiempo que otro evento
                        Puedo = False
            if Puedo == True:
                dispatcher.utter_message(text=f"Yo puedo")
                return{"hora":slot_value}
            else:
                dispatcher.utter_message(text=f"Yo no puedo")
                # reiniciar todos los slots (horario invalido)
                return {"mes": None, "dia":None, "hora":None, "UsersConfirmed":None}

        except HttpError as error:
            print('Ocurrio un error: %s' % error)
    else:
        print('validar horario, algun slot en None')
```

Donde se utiliza la API del calendario de Google. Para esto, primero se verifica la autorización, utilizando las credenciales guardadas en el proyecto, y SCOPES para solo leer el calendario.
De esta manera, se realiza un request de los eventos del mismo día que se organizó la reunión. Por cada evento, se pregunta si alguno interseca, y en ese caso, el horario organizado en la conversación, es inválido.

En cuanto a la extracción de cada slot, se utilizan las definiciones extract_slot dentro de la clase ValidarForm (validate_evento_form).

```python
async def extract_mes(
    self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict
) -> Dict[Text, Any]:
    last_intent = tracker.latest_message['intent'].get('name')
    if last_intent == 'propone_reunion':
        horario = tracker.latest_message['entities']
        mes = tracker.get_slot("mes")
        dia = tracker.get_slot("dia")
        pos = 0
        print('extract_mes')
        length = len(horario)
        print('length:',length)
        print('lista: ',horario)
        i = 0
        while (i < length) and (horario[pos]['extractor'] != 'DucklingEntityExtractor'):
            pos += 1
        print(pos)
        if i < length:
        #si la entity se encuentra en el ultimo mensaje, sino, no hago nada.
            info = horario[pos]['additional_info']['grain']
            print ('info:',info)
            if info == 'month':
                month = ExtraeHorario.mes(horario[pos]['value'])
                return {"mes": month}
            if info == 'day':
                #obtengo slot mes, y si no estaba seteado, lo seteo con el obtenido ahora.
                day = ExtraeHorario.dia(horario[pos]['value'])
                if mes == None:
                    month = ExtraeHorario.mes(horario[pos]['value'])
                    return {"mes": month, "dia":day}
                else:
                    return {"dia":day}
            if (info == 'hour') or (info == 'minute'):
                hour = ExtraeHorario.hora(horario[pos]['value'])
                if mes == None:
                    month = ExtraeHorario.mes(horario[pos]['value'])
                    if dia == None:
                        day = ExtraeHorario.dia(horario[pos]['value'])
                        return {"mes": month, "dia":day,"hora":hour}
                    else:
                        return {"mes": month, "hora":hour}
                else:
                    if dia == None:
                        day = ExtraeHorario.dia(horario[pos]['value'])
                        return {"dia":day, "hora":hour}
                    else:
                        return {"hora":hour}
    else:
        return{"mes":None}
```

```python
async def extract_dia(
    self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict
) -> Dict[Text, Any]:
    last_intent = tracker.latest_message['intent'].get('name')
    if last_intent == 'propone_reunion':
        horario = tracker.latest_message['entities']
        mes = tracker.get_slot("mes")
        dia = tracker.get_slot("dia")
        pos = 0
        print('extract_dia')
        length = len(horario)
        print('length:',length)
        print('horario:',horario)
        i = 0
        while (i < length) and (horario[pos]['extractor'] != 'DucklingEntityExtractor'):
            pos += 1
        print('pos:',pos)
        if i < length:
        #si la entity se encuentra en el ultimo mensaje, sino, no hago nada.
            info = horario[pos]['additional_info']['grain']
            print ('info:',info)
            if info == 'month':
                month = ExtraeHorario.mes(horario[pos]['value'])
                return {"mes": month}
            if info == 'day':
                #obtengo slot mes, y si no estaba seteado, lo seteo con el obtenido ahora.
                day = ExtraeHorario.dia(horario[pos]['value'])
                if mes == None:
                    month = ExtraeHorario.mes(horario[pos]['value'])
                    return {"mes": month, "dia":day}
                else:
                    return {"dia":day}
            if (info == 'hour') or (info == 'minute'):
                hour = ExtraeHorario.hora(horario[pos]['value'])
                if mes == None:
                    month = ExtraeHorario.mes(horario[pos]['value'])
                    if dia == None:
                        day = ExtraeHorario.dia(horario[pos]['value'])
                        return {"mes": month, "dia":day,"hora":hour}
                    else:
                        return {"mes": month, "hora":hour}
                else:
                    if dia == None:
                        day = ExtraeHorario.dia(horario[pos]['value'])
                        return {"dia":day, "hora":hour}
                    else:
                        return {"hora":hour}
    else:
        return{"dia":None}
```

```python
async def extract_hora(
    self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict
) -> Dict[Text, Any]:
    last_intent = tracker.latest_message['intent'].get('name')
    if last_intent == 'propone_reunion':
        horario = tracker.latest_message['entities']
        mes = tracker.get_slot("mes")
        dia = tracker.get_slot("dia")
        pos = 0
        print ('extact_hora')
        length = len(horario)
        print('length',length)
        i = 0
        while (i < length) and (horario[pos]['extractor'] != 'DucklingEntityExtractor'):
            pos += 1
        print('pos:',pos)
        if i < length:
        #si la entity se encuentra en el ultimo mensaje, sino, no hago nada.
            info = horario[pos]['additional_info']['grain']
            print ('info:',info)
            if info == 'month':
                month = ExtraeHorario.mes(horario[pos]['value'])
                return {"mes": month}
            if info == 'day':
                #obtengo slot mes, y si no estaba seteado, lo seteo con el obtenido ahora.
                day = ExtraeHorario.dia(horario[pos]['value'])
                if mes == None:
                    month = ExtraeHorario.mes(horario[pos]['value'])
                    return {"mes": month, "dia":day}
                else:
                    return {"dia":day}
            if (info == 'hour') or (info == 'minute'):
                hour = ExtraeHorario.hora(horario[pos]['value'])
                if mes == None:
                    month = ExtraeHorario.mes(horario[pos]['value'])
                    if dia == None:
                        day = ExtraeHorario.dia(horario[pos]['value'])
                        return {"mes": month, "dia":day,"hora":hour}
                    else:
                        return {"mes": month, "hora":hour}
                else:
                    if dia == None:
                        day = ExtraeHorario.dia(horario[pos]['value'])
                        return {"dia":day, "hora":hour}
                    else:
                        return {"hora":hour}
    else:
        return{"hora":None}
```

Como puede observarse, la extracción de los tres slots, tienen código en común, por lo que podría realizarse un método para mejorar la legibilidad del código.

Esta extracción, solo se realiza si el ultimo mensaje del usuario fue un intent ‘propone_reunion’ asegurando no tomar valores no correspondientes.
Luego, dependiendo de lo que se encuentre en el campo ‘grain’ y los slots cargados sobre la propuesta actual, se decide que slots establecer.

Por último, UsersConfirmed valdrá 1 si la mayoría de los integrantes confirma horario.

```
async def extract_UsersConfirmed(
    self, dispatcher: CollectingDispatcher, tracker: Tracker, domain: Dict
) -> Dict[Text, Any]:
        cantidad = tracker.get_slot("CantUsers")
        confirmados = tracker.get_slot("CantUsersConfirmed")
        print('cantidad:',cantidad)
        print('confirmados:',confirmados)
        if (cantidad != None) and (confirmados != None):
            if  confirmados >= (cantidad/2 + 1):
                return{"UsersConfirmed": 1}
            else:
                return{"UsersConfirmed":None}
        else:
            print('alguno de estos slotes tiene el valor None')
```

Nótese CantUsers es extraído en action_session_start.

Cuando un slot requerido por el form no se haya extraído, el agente reiterará, pidiéndolo. Para esto, el form, automáticamente, llama a utter_ask_slot

```
utter_ask_mes:
- text: "Cuando nos reunimos?"

utter_ask_dia:
- text: " Bien, que dia pueden?"

utter_ask_hora:
- text: " A q hora?"
```

Para el slot UsersConfirm, se utiliza action_ask_slot pudiendo utilizar una custom action.

```
class ActionAskUsersConfirmed(Action):
    # se llama cuando se requiere este slot.
    # anterior a esto, se valido el slot, y no estaba cargado (None) entonces el form llama esta action
    # si no se valida, el bot espera a q confirme la mayoria de personas.
    def name(self):
        return 'action_ask_UsersConfirmed'
    def run(self, dispatcher, tracker, domain):
        return [ActionExecuted("action_listen")]
```

En este caso, si aun quedan integrantes por confirmar la reunión, el agente permanecerá en silencio. Una vez confirme la suficiente cantidad, se valida el form, y si el horario es correcto, se procede a crear el evento en el calendario de Google.

```
class CrearEventoCalendarioGoogle(Action):
    def name(self) -> Text:
        return "action_crea_evento"
    def run(self, dispatcher: CollectingDispatcher,tracker: Tracker,
            domain: Dict[Text, Any]) -> List[Dict[Text, Any]]:
            mes = tracker.get_slot("mes")
            dia = tracker.get_slot("dia")
            hora = tracker.get_slot("hora")
            creds = None
            if (mes == None) or (dia == None) or (hora == None):
                SCOPES = ['https://www.googleapis.com/auth/calendar']#permiso para modificar todo calendario
                #calendar.events para poder crear eventos
                # el archivo token.json almacena el acceso del usuario y actualiza tokens,
                #  es creada automaticamente cuando la actualizacion del flow se completa
                # por primera vez
                if os.path.exists('token.json'):
                    creds = Credentials.from_authorized_user_file('token.json', SCOPES)
                # si no hay credenciales validas disponibles, el usuario loguea.
                if not creds or not creds.valid:
                    if creds and creds.expired and creds.refresh_token:
                        creds.refresh(Request())
                    else:
                        flow = InstalledAppFlow.from_client_secrets_file(
                            ".\\actions\\credentials.json", SCOPES)
                        creds = flow.run_local_server(port=0)
                    # Guarda las credenciales para la proxima ejecucion
                    with open('token.json', 'w') as token:
                        token.write(creds.to_json())
```

Esta vez, el SCOPES es sobre el calendario, tanto para leerlo como para poder modificarlo.

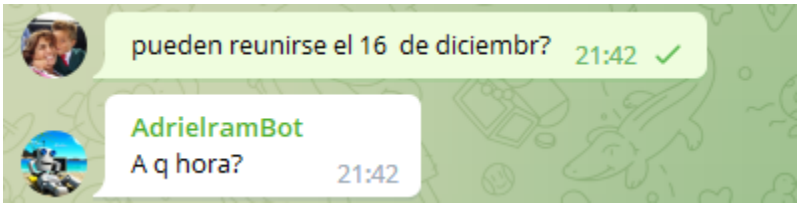
```
    try:#bloque que si no da error, lo ejecuta
        service = build('calendar', 'v3', credentials=creds)
        #a esta action solo debe entrar cuano tenga todos los slots llenos, por ende ni pregunto su valor
        start = datetime.datetime(2022,mes,dia,hora).isoformat()#PARA PODER PONER MINUTOS, MODIFICAR MAPEADO DE HORARIO, Y separar la hora de los minutos
        end = datetime.datetime(2022,mes,dia,hora+1).isoformat()# por ahora, solo se puede una hora en punto. Tambien asumo el evento dura 1 hora
        event_result = service.events().insert(calendarId='d5f9f6052fffdb67cd18405f31e9bb9ab028678ce3128484d2a8239a51f49bc8@group.calendar.google.com',
            body={
            "summary": 'EventoRasa',
            "description": 'Evento de chatbot',
            "start": {"dateTime": start, "timeZone": 'America/Argentina/Buenos_Aires'},
            "end": {"dateTime": end, "timeZone": 'America/Argentina/Buenos_Aires'},
        }
).execute()
        dispatcher.utter_message(text=f"Buenisimo! nos vemos el {dia} a las {hora} entonces")
        # no aclaro mes, se entiende igual
        # si dia = hoy deberia no decirlo..
    except HttpError as error:
        print('An error occurred: %s' % error)
else:
    print('algun slot de crear_evento esta en None')
```

La reunión será de 1 hora, pero es posible implementación, que varíe de acuerdo a las propuestas de los integrantes.

A continuación, se observan algunos ejemplos de la organización de reuniones, y el contexto del dialogo que el agente logra.

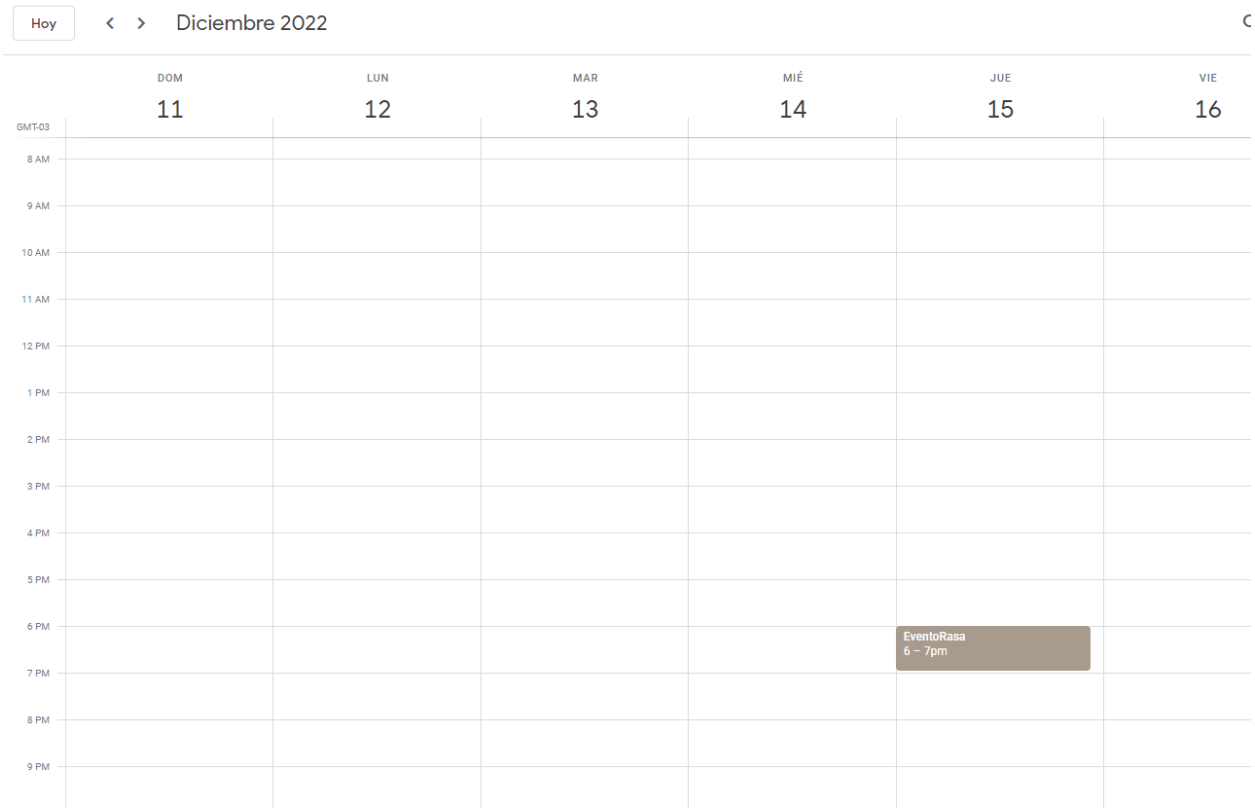
Si solo se propone una reunión, y no se otorga mes, el agente procede a preguntar por el horario

Se puede observar, el agente pregunta solo por lo faltante, teniendo en cuenta el contexto del dialogo.



Si se aclara tanto el mes como el día, solo pregunta hora.

Obsérvese el evento creado en el calendario de Google.



CONCLUSIÓN

RASA tiene un potencial incluso mayor al alcanzado por este bot. Esto se debe, entre otras cosas, a que trabaja con aprendizaje automático. Machine Learning es una tecnología que permite generar patrones para elaborar precisiones con cierta certeza, otorgando flexibilidad y precisión a la hora de desarrollar un chatbot. Además, este framework permite acciones personalizadas, utilizando Python, y como se explicó anteriormente, integrándolo con Prolog.

BIBLIOGRAFÍA

https://rasa.com/docs/

https://forum.rasa.com