1. RASA NLU

While AIML provides flexibility in directing the flow of conversation and incorporating user input, **lots of conversational data** is required in making the bot highly functional, which would require more time and resources in surveying the types of questions that will be asked by consumers. Moreover, the learning curve to implement large scale models is quite steep. Although lots of open-source repositories are available to get a basic model running, the issue of lack of regional language corpora still exists for the same.

Hence, another option was explored which was **RASA Open Source.** Rasa Open Source supplies the building blocks for creating virtual assistants. With Rasa, all teams can create **personalized**, **automated** interactions with customers, at scale. It provides infrastructure and tools necessary for efficient chatbot assistants.

RASA Open Source provides three main functions:

- Natural Language Understanding
- Dialogue Management
- API Integrations

People with a basic understanding of Natural Language Processing can easily create a chatbot model once they go through the documentation which is provided in their website:
https://rasa.com/docs/rasa/

1.1. Components of RASA

- *Rasa X*: It's a Browser based GUI tool which will allow us to train Machine learning model by using GUI based interactive mode. It is an optional tool in the Rasa Software Stack.

- *Rasa NLU*: This is where our main focus lies wherein rasa tries to understand User messages in order to detect **Intent** and **Entity** in our message. Rasa NLU has different components for recognizing intents and entities, most of which have some additional dependencies.

  - SpaCy (need to be installed separately)
  - Tensoflow (available by default with Rasa)

- *Rasa Core*: This is where Rasa tries to help us with contextual message flow. Based on user messages, it can predict dialogue as a reply and can trigger Rasa Action Server.

For integrating Tamil into Rasa, we will be focusing only on **Rasa NLU** and **Rasa Core**.

1.2. Rasa Configuration

When Rasa is initialized, we obtain the following project structure:

- *__init__.py*: An empty file that helps python find our actions.

- *actions.py*: This is where the code for our custom actions is located. In case we want Rasa to call external server via REST API or API call, we can define our Custom Actions here.

- *config.yml*: This is where the configuration of our NLU and Core models resides. If we are dealing with Tensorflow or SpaCy, we need to define these pipelines here.

- *Data/nlu.md*: This is our NLU training data. Here we can define Intents like "Order a Pizza" or "Book an Uber". We need to add related Sentences for that Intent.

- *Data/stories.md*: This is where we store our stories which is required for Rasa Core. The Rasa Core controls the flow of the conversation between us and the chatbot, so for that flow, we need to train chatbot using these stories. This is essentially "Dialog Flow".

- *Domain.yml*: This file combines Different Intents which the chatbot can detect and list of Bot replies. We can define our Custom Action Server Python method name so that Rasa can call that python method for us.

- *endpoints.yml*: It contains details for connecting to channels like FB messenger. This is mainly used for production setup. We can configure Databases like Redis so that Rasa can store tracking information.

Since we can't cover every aspect of Rasa, we will be focusing only on the NLU concepts.

2. Creating the Rasa Chatbot

2.1. Model Configuration

The configuration file defines the **components** and **policies** that our model will use to make predictions based on user input. The **language** and **pipeline keys** specify the components used by the model to make NLU predictions. The policies key defines the policies used by the model to predict the next action.

The following `config.yml` file provides optimum accuracy for the Tamil Language.

```
# Configuration for Rasa NLU.
# https://rasa.com/docs/rasa/nlu/components/
language: ta   # Setting Tamil Language


pipeline:
  - name: WhitespaceTokenizer
  - name: RegexFeaturizer
    # Text will be processed with case sensitive as default
    "case_sensitive": True
```

```
# use match word boundaries for lookup table
    "use_word_boundaries": True
  - name: LexicalSyntacticFeaturizer
  - name: CountVectorsFeaturizer
  - name: CountVectorsFeaturizer
    analyzer: char_wb
    min_ngram: 1
    max_ngram: 4
  - name: DIETClassifier
    epochs: 100
    constrain_similarities: true
  - name: EntitySynonymMapper
  - name: ResponseSelector
    epochs: 100
    constrain_similarities: true
  - name: FallbackClassifier
    threshold: 0.3
    ambiguity_threshold: 0.1

# Configuration for Rasa Core.
policies:
  - name: MemoizationPolicy
    max_history: 7
  - name: TEDPolicy
    max_history: 5
    epochs: 200
    constrain_similarities: true
  - name: RulePolicy
```

The configurations are explained below:

Tokenizers:

Splits text into tokens.

- *WhitespaceTokenizer*: A tokenizer that splits on and discards only whitespace characters.

Featurizer:

Text featurizers are divided into two different categories: **sparse featurizers** and **dense featurizers**. Sparse featurizers are featurizers that return **feature vectors** with a lot of missing values, e.g., zeros. As those feature vectors would normally take up a lot of memory, we store them as sparse features. Sparse features only store the values that are non-zero and their positions in the vector. Thus, we **save a lot of memory** and are able to train on larger datasets.

- *RegexFeaturizer*: Creates a vector representation of user message using regular expressions.
- *LexicalSyntacticFeaturizer*: Creates lexical and syntactic features for a user message to support entity extraction.
- *CountVectorsFeaturizer*: Creates bag-of-words representation of user messages, intents, and responses.

Intent Classifiers:

Intent classifiers assign one of the intents defined in the domain file to incoming user messages.

- *DIETClassifier*: Dual Intent Entity Transformer (DIET) used for intent classification and entity extraction.
- *FallbackClassifier*: Classifies a message with the intent nlu_fallback if the NLU intent classification scores are ambiguous.

Entity Extractors:

Entity extractors extract entities, such as person names or locations, from the user message.

- *EntitySynonymMapper*: Maps synonymous entity values to the same value.

Selectors:

Selectors predict a bot response from a set of candidate responses.

- *ResponseSelector*: embeds user inputs and candidate response into the same space.

Policies:

Chatbot Assistants use policies to decide which actions to take at each step in a conversation. There are machine-learning and rule-based policies that it can use in tandem.

- *MemoizationPolicy*: The MemoizationPolicy remembers the stories from our training data. It checks if the current conversation **matches** the stories in our stories.yml file. If so, it will

predict the next action from the matching stories of your training data with a confidence of 1.0. If no matching conversation is found, the policy predicts None with confidence 0.0.

- *TEDPolicy*: The Transformer Embedding Dialogue (TED) Policy is a **multi-task architecture** for next action prediction and entity recognition. The architecture consists of several **transformer encoders** which are shared for both tasks. A sequence of entity labels is predicted through a **Conditional Random Field (CRF)** tagging layer on top of the user sequence transformer encoder output corresponding to the input sequence of tokens. For the next action prediction, the dialogue transformer encoder output and the system action labels are **embedded** into a single **semantic vector space**. We use the dot-product loss to maximize the similarity with the target label and minimize similarities with negative samples.

Note: Definitions obtained from https://rasa.com/docs/rasa/components
https://rasa.com/docs/rasa/policies

2.2. Domain

The domain defines the **universe** in which our chatbot operates. It specifies the intents, entities, slots, responses, forms, and actions our bot should know about. It also defines a configuration for conversation sessions.

Here is an example of **`domain.yml`** file of a Tamil Chatbot that assists telecommunication users in purchasing cell phone packages:

domain.yml

(Due to a large number of lines of code, a pdf version has been provided to view the codebase of the file.)

The elements of the domain file are explained below:

- **Intents:**

  The intents key in our domain file lists all intents used in our NLU data and conversation training data.

- **Entities:**

  The entities section lists all entities that can be extracted by any entity extractor in our NLU pipeline.

- **Slots:**

  Slots are our bot's memory. They act as a key-value store which can be used to save information the user provided (for e.g., their home city) as well as information gathered about the outside world (e.g., the result of a database query).

- **Responses:**

  Responses are actions that send a message to a user without running any custom code or returning events. These responses can be defined directly in the domain file under the responses key and can include rich content such as **buttons** and **attachments**.

- **Forms:**

  Forms are a special type of action meant to help our bot collect information from a user.

- **Actions:**

  Actions are the events our bot can execute. For example, an action could:

  - Respond to a user,
  - Make an external API call, or
  - Query a database

- **Session Configuration:**
  A conversation session represents the dialogue between the chatbot and the user. We can define the period of inactivity after which a new conversation session is triggered in the domain under the **`session_config`** key.

  Available parameters are:
  - **`session_expiration_time`** that defines the time of inactivity in minutes after which a new session will begin.
  - **`carry_over_slots_to_new_session`** that determines whether existing set slots should be carried over to new sessions.

  In the domain.yml file, if a user sends their first message after 60 minutes of inactivity, a new conversation session is triggered, and that any existing slots are carried over into the new session. Setting the value of **`session_expiration_time`** to 0 means that sessions will not end.

- **Config:**
  The config key in the domain file maintains the **`store_entities_as_slots`** parameter. When an entity is recognized by the NLU model and the entity name matches a slot name, **`store_entities_as_slots`** defines whether the entity value should be placed in that slot. By default, entities will auto-fill slots of the same name.

  Note: Definitions obtained from https://rasa.com/docs/rasa/domain

2.3. Stories
Stories are a type of training data used to train our chatbot’s **dialogue management model**. Stories can be used to train models that are able to generalize to unseen conversation paths. In other words, a story is a representation of a conversation between a user and an AI assistant, converted into a specific format where user inputs are expressed as intents (and entities when necessary), while the assistant's responses and actions are expressed as **action names.**

Taking the same sample case as we took while discussing domains, here is an example of a dialogue in the Rasa story format:

stories.yml

(Due to a large number of lines of code, a pdf version has been provided to view the codebase of the file.)

The elements of a stories file are discussed below:

- **User Messages:**

  While writing stories, we do not have to deal with the specific contents of the messages that the users send. Instead, we can take advantage of the **output** from the NLU pipeline, which lets us use just the combination of an **intent** and **entities** to refer to all the possible messages the users can send to mean the same thing.

  It is important to include the entities here as well because the policies learn to predict the next action based on a combination of both the intent and entities.

- **Actions:**

  All actions that are executed by the bot, including the responses are listed in stories under the **action key**. We can use a response from our domain as an action by listing it as one in a story. Similarly, we can indicate that a story should call a custom action by including the name of the custom action from the actions list in our domain.

- **Events:**

  During training, Rasa Open Source does not call the action server. This means that our bot's dialogue management model doesn't know which events a custom action will return. Because of this, events such as setting a **slot** or **activating/deactivating** a **form** have to be explicitly written out as part of the stories.

Note: Definitions obtained from https://rasa.com/docs/rasa/stories

2.4. Rules

Rules are a type of training data used to train our chatbot’s **dialogue management model**. Rules describe short pieces of conversations that should always follow the same path.

Rules are great to handle **small specific conversation patterns**, but unlike stories, rules don't have the power to generalize to unseen conversation paths. We can combine rules and stories to make our chatbot robust and able to handle real user behavior.

Note that before we start writing rules, we have to make sure that the **Rule Policy** is added to our model configuration:

```
policies:
- ... # Other policies
- name: RulePolicy
```

The following is an example of the **`rules.yml`** file that supplements the **`stories.yml`** file as discussed earlier.

```
version: "2.0"

# standard rules implemented
rules:

  - rule: Say goodbye anytime the user says goodbye
    steps:
      - intent: goodbye
      - action: utter_goodbye

  - rule: Say 'I am a bot' anytime the user challenges
    steps:
      - intent: bot_challenge
      - action: utter_iamabot

  - rule: Greeting Rule
    steps:
      - intent: greet
      - action: utter_greet
```

Note: Definitions obtained from https://rasa.com/docs/rasa/rules/

2.5. Training Data Format

NLU training data consists of **example user utterances** categorized by **intent**. Training examples can also include entities. Entities are structured pieces of information that can be extracted from a user's message. We can also add extra information such as regular expressions and **lookup tables** to our training data to help the model identify intents and entities correctly.

The following file **`nlu.yml`** contains the training data for the NLU. Both English and Tamil words have been added to take into account the fact that people tend to mix languages when in conversation.

nlu.yml

(Due to a large number of lines of code, a pdf version has been provided to view the codebase of the file.)

NLU training data is defined under the **`nlu`** key. Items that can be added under this key are:

- Training examples grouped by user intent e.g., optionally with annotated entities.

```
nlu:
- intent: check_balance
  examples: |
    - What's my [credit](account) balance?
    - What's the balance on my [credit card
account]{"entity":"account","value":"credit"}
```

- *Synonyms*: Synonyms normalize your training data by mapping an extracted entity to a value other than the literal text extracted.

```
nlu:
- synonym: credit
  examples: |
    - credit card account
    - credit account
```

- *Regular expressions*: You can use regular expressions to improve intent classification and entity extraction using the *RegexFeaturizer* and *RegexEntityExtractor* components.

```
nlu:
- regex: account_number
  examples: |
    - \d{10,12}
```

- *Lookup tables*: Lookup tables are lists of words used to generate case-insensitive regular expression patterns.

```
nlu:
- lookup: banks
  examples: |
    - JPMC
    - Comerica
    - Bank of America
```

Conversation Training Data:
Stories and rules are both representations of conversations between a user and a conversational chatbot assistant. They are used to train the **dialogue management model**. Stories are used to

train a machine learning model to identify patterns in conversations and generalize to unseen conversation paths. Rules describe small pieces of conversations that should always follow the same path and are used to train the *RulePolicy*.

Note: Definitions obtained from https://rasa.com/docs/rasa/training-data-format/

3. Running the Chatbot

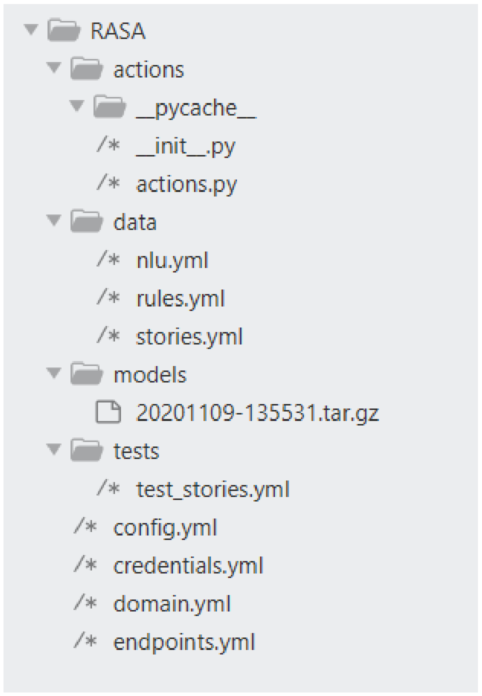
As a recap, our focus in developing our chatbot model using Rasa is towards the following files:

- Step 1: nlu.yml
- Step 2: actions.py
- Step 3: domain.yml
- Step 4: stories.yml

Once all our information about our chatbot is entered in these files, we can star training our chatbot.

```
Rasa train
```

Once done, we will be left with the following file structure:



```
RASA
  actions
    __pycache__
    __init__.py
    actions.py
  data
    nlu.yml
    rules.yml
    stories.yml
  models
    20201109-135531.tar.gz
  tests
    test_stories.yml
  config.yml
  credentials.yml
  domain.yml
  endpoints.yml
```

To run our chatbot, we have two methods:

- **In the Shell:**

For running Rasa inside the shell, just type the command in a terminal opened in the Rasa folder:

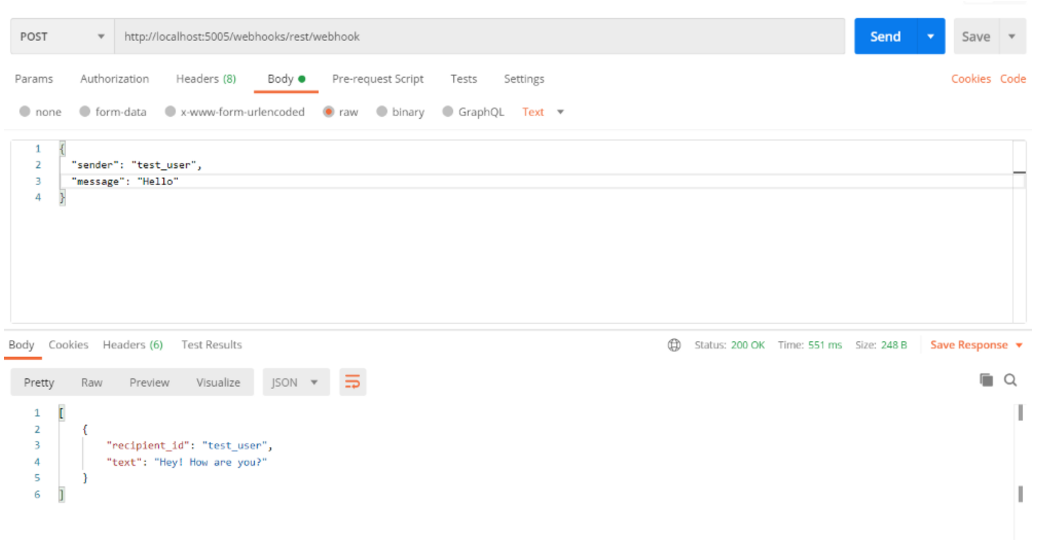
```
Rasa shell
```

This command will start the Rasa server and we will be able to talk to the chatbot right in the terminal.

- **On the Localhost:**
  The other way is to run Rasa on the localhost server. For that, just run the following command from a terminal opened in the Rasa folder:

```
rasa run
```

This would run Rasa on our local system and **expose** a REST endpoint at 5000 port in the localhost. In order to communicate with the REST endpoint exposed by the Rasa server, we can use cURL commands (for Linux) or Postman. Postman is better because of its ease-of-use.



http://localhost:5005/webhooks/rest/webhook

```
{
 "sender": "test_user",
 "message": "Hello"
}
```

Status: 200 OK Time: 551 ms Size: 248 B

```
[
    {
        "recipient_id": "test_user",
        "text": "Hey! How are you?"
    }
]
```

We have to make a **POST** request to the URL:

http://localhost:5005/webhooks/rest/webhook

- In the **Body** of the request, select the **raw** option in Postman.
- In JSON format, we have to specify two keys: **sender** and **message**.
- Put a dummy value in the sender and the message will be our query to the chatbot.
- We will be returned a JSON with keys: **recipient_id** and **text**. **text** will have the reply of the chatbot to our query and **recipient_id** will be the same as the **sender** key we sent in the request.

- **Other Options:**
  We can integrate this chatbot into a website, Facebook Messenger, Slack, Telegram, Twilio, Microsoft Bot Framework, Cisco Webex Teams, RocketChat, Mattermost, and Google Hangouts Chat. We can also connect the chatbot to any other platform but we would have to create our own **custom connector**. Information regarding API integration can be obtained from the Rasa Documentation.