Chatbot

@ Bavalpreet Singh (Unlicensed) @ Sahibpreet Singh you can start documenting here

Chatbot through RASA

Introduction:

A chatbot is an application that can initiate and continue a conversation using auditory and/or textual methods as a human would do. A chatbot can be either a simple rule-based engine or an intelligent application leveraging Natural Language Understanding(Contextual AI bots).

1.1 **Uses**

Customer support

Frequently Asked Questions

Addressing Grievances

Appointment Booking

Automation of routine tasks

Address a query

Prerequisites

The prerequisites for developing and understanding a chatbot using Rasa are:

Python 3.6/3.7

For windows - Microsoft VC++ Compiler

Installation

You can install Rasa Open Source using pip (requires Python 3.6, 3.7 or 3.8).

pip3 install rasa

You can create a new project by running:

rasa init

If environment is not setup:

Ubuntu

1. **Conda -** install anaconda(https://docs.conda.io/projects/conda/en/4.6.0/_downloads/52a95608c49671267e40c689e0bc00ca/conda-cheatsheet.pdf)
    a. Create virtual environment - where name is name of your environment
        i. `conda create --name py35 python=3.5`
    b. Start the environment - `conda activate <name>`
    c. Install rasa - `pip install rasa`
    d. Create a new project by `rasa init`
2. **Venv**
    a. Rasa Till data Just support Python 3.6 . So install Python 3.6 in your Local computer
    b. `virtualenv --python=python3.6 <name of the env>`
    c. `cd <name of env>`

d. `source bin/activate`

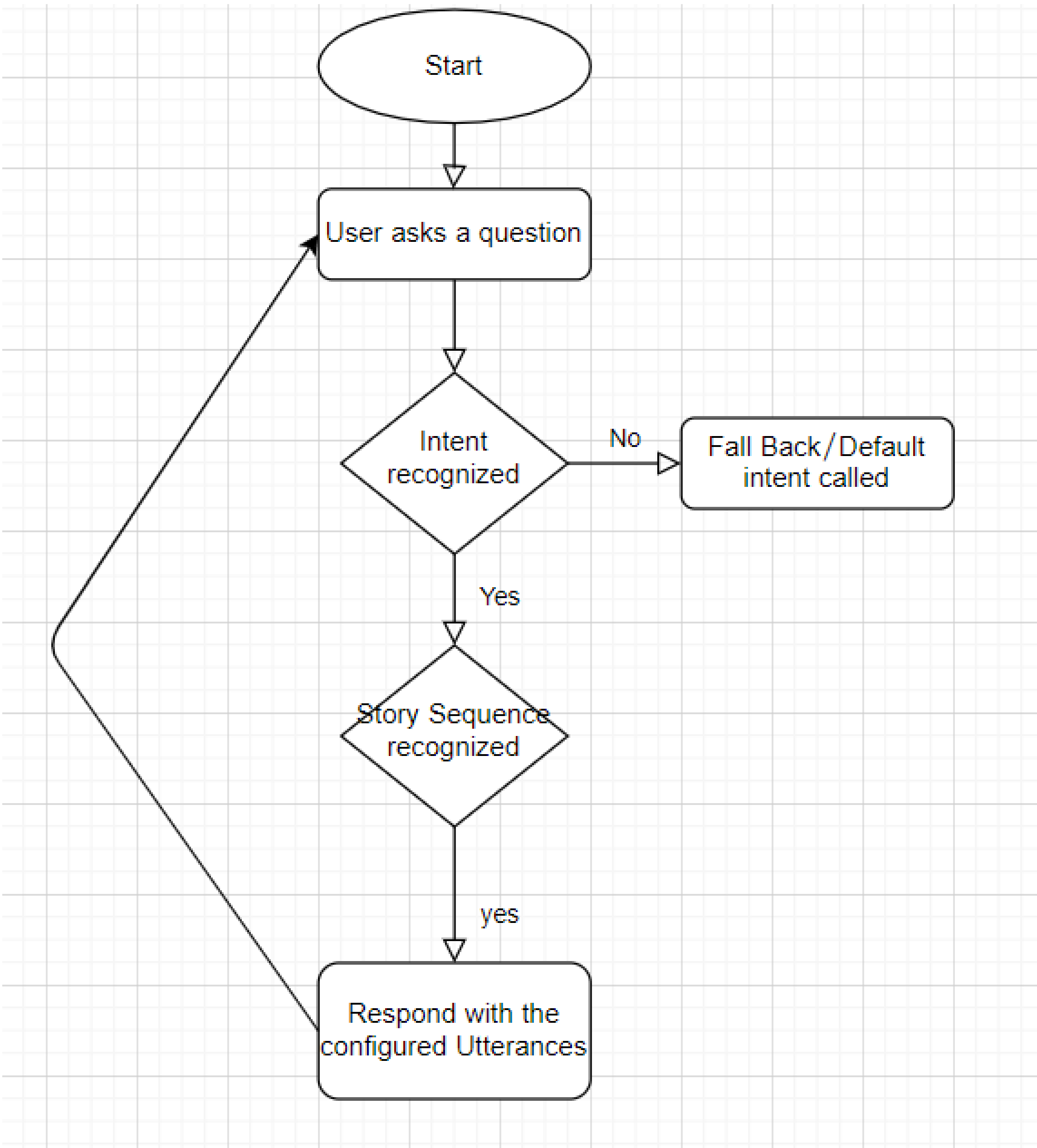
For all the subsequent actions choose Y(for training the model etc).

1. Your Virtual environment is ready. For further you can refer rasa cheat sheet- https://rasa.com/docs/rasa/command-line-interface

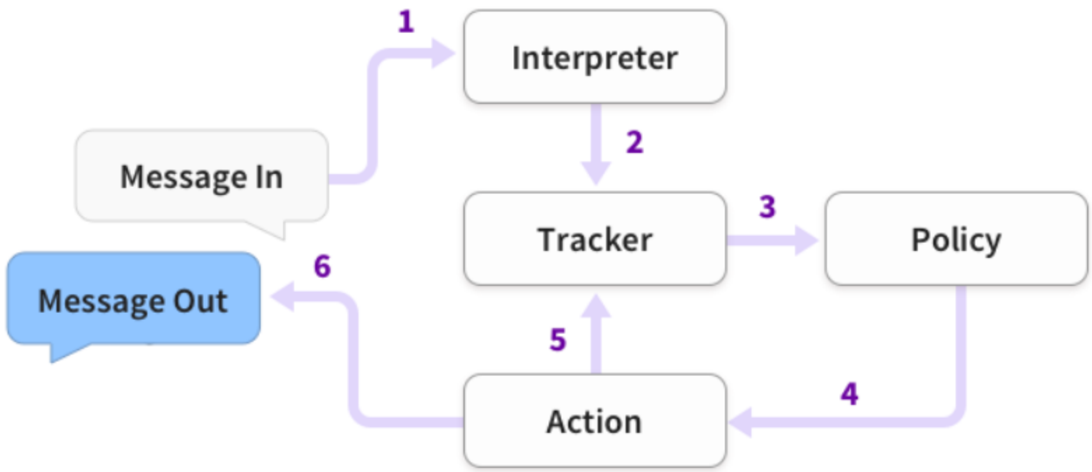
The Problem Statement

The goal here is to build a Contextual bot which can answer queries related to boats.

The application flow



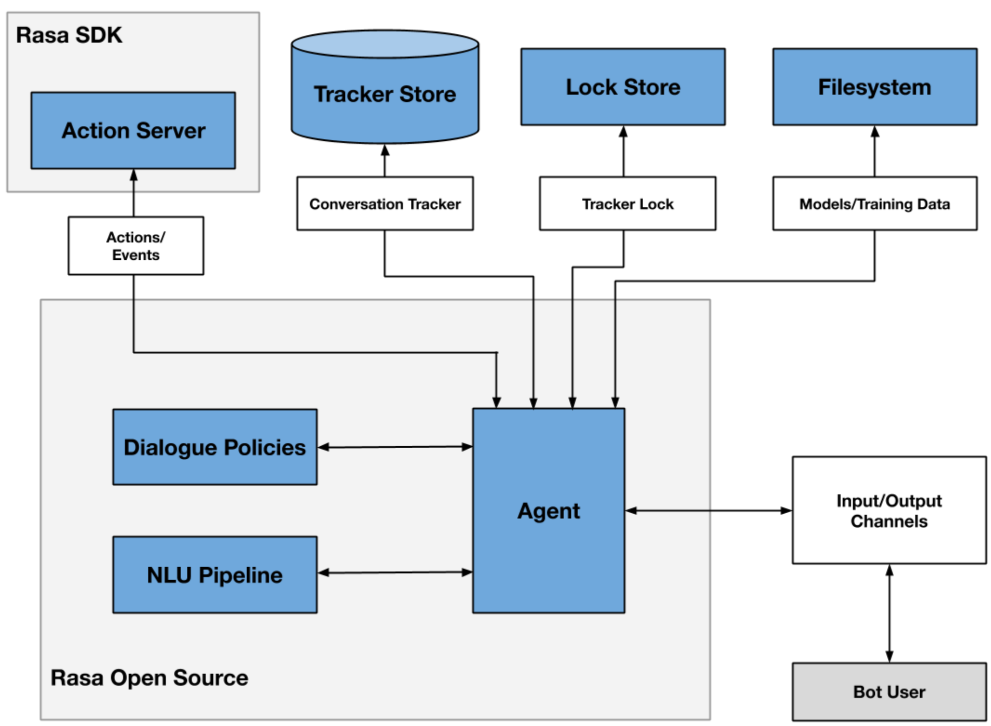
1. **Rasa Architecture Overview**
    a. Overview of working



The steps are:

i. The message is received and passed to an `Interpreter`, which converts it into a dictionary including the original text, the intent, and any entities that were found. This part is handled by NLU.
ii. The `Tracker` is the object which keeps track of conversation state. It receives the info that a new message has come in.
iii. The policy receives the current state of the tracker.
iv. The policy chooses which action to take next.
v. The chosen action is logged by the tracker.
vi. A response is sent to the user.

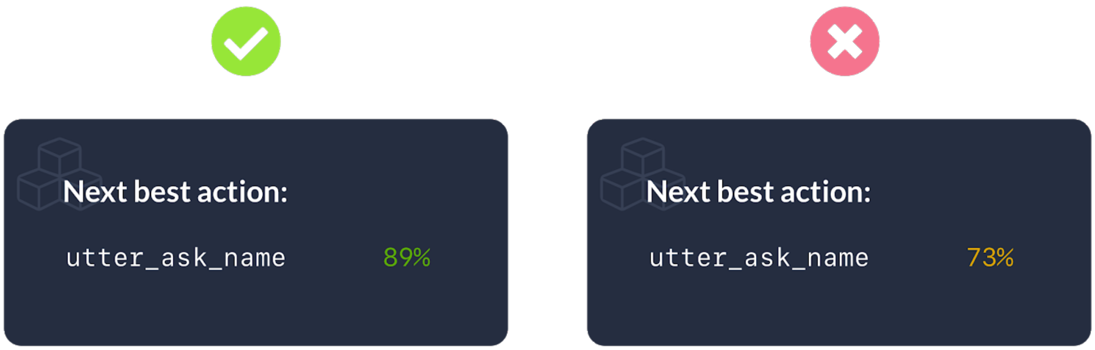
b. In little detail



NLU is the part that handles intent classification, entity extraction, and response retrieval. It's shown below as the *NLU Pipeline* b ecause it processes user utterances using an NLU model that is generated by the trained pipeline.

The dialogue management component decides the next action in a conversation based on the context. This is displayed as the *D ialogue Policies* in the diagram.

2. **POLICIES -**
Policies are components that train the dialogue model, and they play a very important role in determining its behavior. Some policies are quite simple, like those that mirror the conversations they've been trained on, and some are quite complex, like those that rely on sophisticated machine learning to predict the next action based on the context of the conversation. Just like the NLU training pipeline dialogue policies are also configured in the config.yml file, which you'll find in your main project directory. Unlike the NLU training pipeline, which runs components sequentially, dialogue policies run in parallel. At each conversational turn, each policy in the configuration makes its own prediction about the next best action. The policy that predicts the next action with the highest confidence level determines the assistant's next action.



In the case that two policies predict with equal confidence (for example, the Memoization and Rule Policies might both predict with confidence 1), the priority of the policies is considered. Rasa Open Source policies have default priorities that are set to ensure the expected outcome in the case of a tie. They look like this, where higher numbers have higher priority:

- 6 - `RulePolicy`
- 3 - `MemoizationPolicy` or `AugmentedMemoizationPolicy`
- 1 - `TEDPolicy`

In general, it is not recommended to have more than one policy per priority level in your configuration. If you have 2 policies with the same priority and they predict with the same confidence, the resulting action will be chosen randomly. Generally speaking, you should only have one policy per priority level to avoid conflicts, and some policies, like Fallback and TwoStageFallback, explicitly cannot be used together. We'll discuss configuration in greater detail when we cover each policy in depth. For now, keep in mind that multiple policies can be used together, and the highest confidence, highest priority policy predicts the assistant's next action.

3. Rules

Rules are a type of training data used to train your assistant's dialogue management model. Rules describe short pieces of conversations that should always follow the same path.

Don't overuse rules. Rules are great to handle small specific conversation patterns, but unlike stories, rules don't have the power to generalize to unseen conversation paths. Combine rules and stories to make your assistant robust and able to handle real user behavior.

Writing a Rule

Before you start writing rules, you have to make sure that the Rule Policy is added to your model configuration:

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

Rules can then be added to the `rules` section of your training data.

```yaml
version: "2.0"

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: respond to generalQs
  steps:
  - intent: generalQ
  - action: utter_generalQ

- rule: respond to chitchat
  steps:
  - intent: chitchat
  - action: utter_chitchat
```

4. Stories

Stories are a type of training data used to train your assistant's dialogue management model. Stories can be used to train models that are able to generalize to unseen conversation paths.

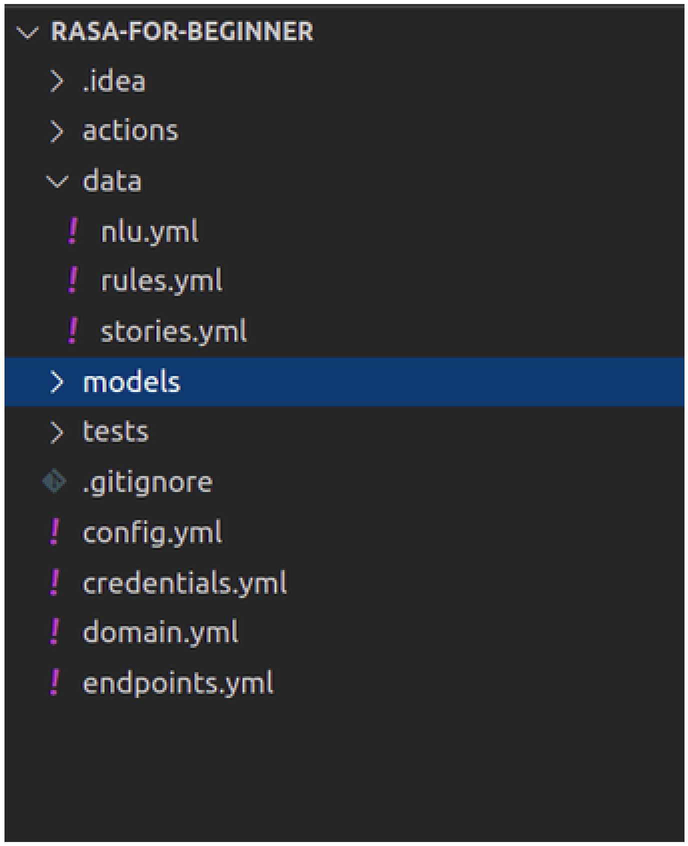
Format

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.

Here's an example of a dialogue in the Rasa story format:

Photo

1. **File Structure in rasa**



High-Level Structure

Each file can contain one or more **keys** with corresponding training data. One file can contain multiple keys, but each key can only appear once in a single file. The available keys are:

- `version`
- `nlu`
- `stories`
- `rules`

You should specify the `version` key in all YAML training data files.

THE | SYMBOL

As shown in the above examples, the `user` and `examples` keys are followed by `|` (pipe) symbol. In YAML `|` identifies multi-line strings with preserved indentation. This helps to keep special symbols like `"`, `'` and others still available in the training examples.

1. **Nlu.yml**
   This file basically contains all the intents with their specific examples. 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.

   NLU training data is defined under the `nlu` key. Items that can be added under this key are:
   - Training Examples, synonyms etc

data > ! nlu.yml

```yaml
version: "2.0"
nlu:
- intent: no_repair_required
  examples: |
    - Now my boat is working fine
    - I have no issues with my boat now.
    - each part of my boat is working fine
    - yeah it's good now
    - My [hull](boat_part) is good now
    - [hull](boat_part) is working fine
    - I have no issue with core
    - [core](boat_part) is working compltely fine
    - hey my [hull](boat_part) is good now
    - perfectly fine working
    - Hmmm no issues with any boat part
    - yeah it's good to go
    - [core](boat_part) is good now
    - [core](boat_part) is good
    - [hull](boat_part) is good
```

Training examples are grouped by intent and listed under the `examples` key.

- If you want to specify retrieval intents, then your NLU examples will look as follows:

```yaml
- intent: generalQ/teak-surfing-query
  examples: |
    - what is teak surfing?
    - how to perform teak surfing?
    - how teak surfing is done?
    - can you explain what exactly teak surfing is?
    - please elaborate about teak surfing?
```

  - All retrieval intents have a suffix added to them which identifies a particular response key for your assistant. In the above example, `teak-surfing-query` is the suffixes. The suffix is separated from the retrieval intent name by a `/` del imiter.

    SPECIAL MEANING OF `/`

    As shown in the above examples, the `/` symbol is reserved as a delimiter to separate retrieval intents from their associated response keys. Make sure not to use it in the name of your intents.

- Entities

  Entities are structured pieces of information that can be extracted from a user's message.

2. **Conversation Training Data**

   Stories and rules are both representations of conversations between a user and a conversational 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.

   a. **stories.yml**
      Stories are composed of:
      - `story`: The story's name. The name is arbitrary and not used in training; you can use it as a human-readable reference for the story.
      - a list of `steps`: The user messages and actions that make up the story

data > ! stories.yml

```yaml
version: "2.0"

stories:

- story: say goodbye
  steps:
  - intent: goodbye
  - action: utter_goodbye
  - intent: thankyou
  - action: utter_thankyou

- story: ask repair_hull
  steps:
  - intent: greet
  - action: action_greet
  - intent: specificq
  - action: action_specificq
  - action: utter_did_that_help
  - intent: no_repair_required
  - action: utter_no_repair_required
  - intent: affirm
  - action: utter_happy
  - intent: thankyou
  - action: utter_thankyou
```

Till now in our project each step can be one of the following:

- A user message, represented by **intent** and **entities**.
- A bot action.
- A slot was set event.

  User Messages

  All user messages are specified with the `intent:` key and an optional `entities:` key.
  While writing stories, you can take advantage of the output from the NLU pipeline, which uses a combination of an intent and entities to refer to all possible messages the users can send with the same meaning.

  User messages follows the following format:

  - ```yaml
    - story: ask repair_core
      steps:
      - intent: greet #required
      - entities: #optional
        entity_name: xyz
    ```

  - example 2
    ```

```yaml
- story: interactive_story_2
  steps:
  - intent: greet
  - action: action_greet
  - intent: specificq
    entities:
    - boat_part: hull
  - slot_was_set:
    - boat_part:
      - hull
  - action: action_specificq
```

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 (you can, however, change this behavior using the `use_entities` attribute).

Actions

All actions executed by the bot are specified with the `action:` key followed by the name of the action. While writing stories, you will encounter two types of actions:

i. Responses

: start with `utter_` and send a specific message to the user. e.g.

```yaml
! stories.yml
version: "2.0"

stories:

- story: say goodbye
  steps:
  - intent: goodbye
  - action: utter_goodbye
  - intent: thankyou
  - action: utter_thankyou
```

ii. Custom actions

: start with `action_`, run arbitrary code and send any number of messages (or none).

```yaml
- story: ask repair_hull
  steps:
  - intent: greet
  - action: action_greet
  - intent: specificq
  - action: action_specificq
```

- Slots

A slot event is specified under the key `slot_was_set:` with the slot name and optionally the slot's value.

Slots act as the bots memory. Slots are **set** by entities or by custom actions and **referenced** by stories in `slot_was_set` steps. For example:

```
- story: interactive_story_2
  steps:
  - intent: greet
  - action: action_greet
  - intent: specificq
    entities:
    - boat_part: hull
  - slot_was_set:
    - boat_part:
      - hull
```

This means the story requires that the current value for the `boat_part` slot be `hull` for the conversation to continue as specified.

HOW SLOTS WORK

Stories do not **set** slots. The slot must be set by an entity or custom action **before** the `slot_was_set` step.

b. Rules.yml

Rules are listed under the `rules` key and look similar to stories. A rule also has a `steps` key, which contains a list of the same steps as stories do.

data > ! rules.yml

```
version: "2.0"
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: respond to generalQs
  steps:
  - intent: generalQ
  - action: utter_generalQ

- rule: respond to chitchat
  steps:
  - intent: chitchat
  - action: utter_chitchat
```

3. domain.yml

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

domain.yml

```yaml
session_config:
  session_expiration_time: 60
  carry_over_slots_to_new_session: true
intents:
- no_repair_required
- greet
- goodbye
- thankyou
- affirm
- deny
- out_of_scope
- specificq
- chitchat
- generalQ
- bot_challenge
- nlu_fallback
entities:
- boat_manufacturer
- boat_part
- engine_series
- engine_manufacturer
- boat_length
- boat_model
- year_of_manufacturing
- consumable
- process
- material
slots:
  boat_length:
    type: list
    initial_value: []
    influence_conversation: false
  boat_manufacturer:
    type: list
    initial_value: []
    influence_conversation: false
```

1.
2.

domain.yml

```yaml
responses:
  utter_please_rephrase:
  - text: I'm sorry, I didn't quite understand that. Could you rephrase?
  utter_did_that_help:
  - text: Do you Need any other help.
  - text: Can i help you some other way.
  - text: Thanks can I help you some other way.
  - text: Hope I could help you
  utter_goodbye:
  - text: Bye tc :)
  - text: Bbye it was nice talking to you.
  - text: see you soon.
  utter_happy:
  - text: Great, carry on!
  utter_iamabot:
  - text: I am a bot, powered by Rasa.
  utter_no_repair_required:
  - text: Happy that i could help you :)
  - text: That's such a great news CHEERS
  - text: Perfect
```

domain.yml

```yaml
actions:
- utter_please_rephrase
- utter_did_that_help
- utter_goodbye
- utter_happy
- utter_no_repair_required
- utter_out_of_scope
- utter_thankyou
- action_specificq
- action_greet
- utter_generalQ/mooring-query
- utter_generalQ/teak-surfing-query
- utter_generalQ/min-age-for-ski-craft-query
- utter_generalQ/no-of-people-towed-query
- utter_generalQ/observers-query
```

3. The domain includes the definitions for responses and forms

4. Action.py

actions > actions.py > Action_SpecificQ > Answer_finder

```python

from typing import Any, Text, Dict, List

from rasa_sdk import Action, Tracker
from rasa_sdk.executor import CollectingDispatcher
from rasa_sdk.events import SlotSet
import pandas as pd


from sentence_transformers import SentenceTransformer, util

from spellcheck import correction
from sef import entity_finder,slot_setter
class ActionGreet(Action):

    def name(self) -> Text:
        return "action_greet"

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

            dispatcher.utter_message(text="Hi how can i help you with your boat")

            return []

    def loader(self):
        model = SentenceTransformer('distilbert-base-nli-stsb-mean-tokens')
        df=pd.read_csv('/home/bavalpreet/Downloads/boatbox/faq-rasa.csv')
        sentences=df['Questions'].str.replace("\n", "", case = False).tolist()
        solutions=df['Answers'].str.replace("\n", "", case = False).tolist()

        embeddings = model.encode(sentences)

        return [model,embeddings,solutions]



class Action_SpecificQ(ActionGreet):

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

Actions

After each user message, the Rasa will predict an action that the assistant should perform next.

Responses

A response is a message the assistant will send back to the user. This is the action you will use most often, when you want the assistant to send text, images, buttons or similar to the user.

Custom Actions

A custom action can run any code you want, including API calls, database queries etc. They can turn on the lights, add an event to a calendar, check a user's bank balance, or anything else you can imagine.

For Boat-box we have Implemented some **Custom Actions**
Some Custom Actions Were :-

1. `ActionGreet`
2. `Action_SpecificQ`

ActionGreet

Is the action that responds to Greetings sent by user i.e
When user says :-

1. Hi
2. Hello
3. Heya
4. whats-up
5. Or Anything like that

Boatbox will reposnd to user and at the same time it will create embeddings using `SBERT` these embeddings will be later used by Another action for finding Answer to User Query.

And these embeddings are for questions that we will use to find most suitable Answer for user query.

Action_SpecificQ

This Custom Action uses all the embeddings that we have created for Questions in our Database.

STEPS here:-

1. First we will extract entities from user input.
   a. Then we will store all entities in a list and will correct there spellings using our spellcheck code.
   b. After we get correct spellings we will look for only those answers which have these specific entities and will select only those Question-Answer pairs.
   c. After selecting Question-Answer pairs (say n) are selected we will now check cosine similarity for user input and all the selected questions.
   d. The question which is most similar to user input that corresponding answer will be pushed to chatbot.

<u>After creating the Custom actions in actions.py, don't forget to link this with bot in domain.yml file. Add “action_specificq” action in the list of the action. This should look like this</u>

```yaml
# ! domain.yml
actions:
- action_specificq
```

<u>Now in the stories, add this custom action as your flow.</u>

```yaml
# data > ! stories.yml
- story: interactive_story_1
  steps:
  - intent: greet
  - action: action_greet
  - intent: specificq
  - action: action_specificq
  - action: utter_did_that_help
  - intent: thankyou
  - action: utter_thankyou
```

<u>Tell rasa to use Custom Action Server in “endpoints.yml”.</u>

```yaml
endpoints.yml
action_endpoint:
  url: "http://localhost:5055/webhook"
```

then retrain the model because we have made changes in stories and nlu .yml files.

Slots

Slots are your bot's memory. They act as a key-value store which can be used to store information the user provided. Slots are defined in the slots section of your domain with their name, type and if and how they should influence the assistant's behavior. The following example defines a slot with name "slot_name" and type `text`.

```yaml
slots:
  slot_name:
    type: text
```

Slots and Conversation Behavior

You can specify whether or not a slot influences the conversation with the `influence_conversation` property.

If you want to store information in a slot without it influencing the conversation, set `influence_conversation: false` when defining your slot.

The following example defines a slot `boat_length` which will store information about the boat's length, but which will *not* influence the flow of the conversation. This means that the assistant will ignore the value of the slot each time it predicts the next action.

```yaml
slots:
  boat_length:
    type: list
    initial_value: []
    influence_conversation: false
```

When defining a slot, if you leave out `influence_conversation` or set it to `true`, that slot will influence the next action prediction, unless it has slot type `any`. The way the slot influences the conversation will depend on its slot type. There are 6 type of slots available namely: `Text Slot, Boolean Slot, Categorical Slot, Float Slot, List Slot and Any Slot`. Among them we are using the List Slot.

Handling FAQ's using Retrieval Intents

To handle FAQs and chitchat you'll need a rule-based dialogue management policy (the RulePolicy) and an easy way to return the appropriate response for a question (the ResponseSelector).

1. Updating the configuration

For FAQs and chitchat, you always want the assistant to respond the same way every time the same type of question is asked. Rules allow you to do exactly that. To use rules, the you need to add the RulePolicy to your policies in your configuration file:

photo

Next, include the ResponseSelector in your NLU pipeline in your configuration file. The ResponseSelector requires a featurizer and intent classifier to work, so it should come after these components in your pipeline

photo

By default, the ResponseSelector will build a single retrieval model for all retrieval intents. To retrieve responses for FAQs and chitchat separately, we used multiple ResponseSelector components and specify the `retrieval_intent` key:

2. Defining Retrieval Intents and the ResponseSelector

Consider an example where you have 20 different FAQs. Although each question is represented as an individual intent, all FAQ intents are handled the same way in the dialogue. For each FAQ intent, the assistant **retrieves** the proper response depending on which question has been asked.

Instead of writing 20 rules, you can use a single action, e.g. `utter_generalQ` to handle all FAQs with a single rule by grouping them together under a single retrieval intent called e.g. `generalQ`.

3. Creating rules

You need to write only one rule for each retrieval intent. All intents grouped under that retrieval intent will then be handled the same way. The action name starts with `utter_` and ends with the retrieval intent's name. Write rules for responding to FAQs and chitchat:

rules.yml

```
- rule: respond to generalQs
  steps:
  - intent: generalQ
  - action: utter_generalQ

- rule: respond to chitchat
  steps:
  - intent: chitchat
  - action: utter_chitchat
```

The actions `utter_generalQ` and `utter_chitchat` will use the ResponseSelector's prediction to return the actual response message.

4. Updating the NLU Training Data

NLU training examples for the ResponseSelector look the same as regular training examples, except that their names must refer to the retrieval intent they are grouped under:

nlu.yml

```yaml
- intent: generalQ/mooring-query
  examples: |
    - How is the maximum vessel length determined for my mooring?
    - how to calculate maximum length of vessel
    - how to do mooring?
    - what is mooring?
    - what is meaning of mooring?

- intent: generalQ/teak-surfing-query
  examples: |
    - what is teak surfing?
    - how to perform teak surfing?
    - how teak surfing is done?
    - can you explain what exactly teak surfing is?
    - please elaborate about teak surfing?
```

Be sure to update your domain file to include the added `chitchat` intent:

domain.yml

```yaml
intents:
# other intents
- chitchat
```

5. Defining the responses

Responses for the ResponseSelector follow the same naming convention as retrieval intents. Besides this, they can have all the characteristics of normal bot response. For the chitchat intents listed above, our responses could look like:

domain.yml

```yaml
responses:
utter_generalQ/mooring-query:
- text: Maximum Vessel Length (MVL) is determined by a formula that includes body
    swing radius, pickup rope length and maximum swing room. The formula ensures
    safety for vessel occupants in the event of an emergency.
```

Train a model

The main command is:

```
rasa train
```

This command trains a Rasa model that combines a Rasa NLU and a Rasa Core model. If you only want to train an NLU or a Core model, you can run `rasa train nlu` or `rasa train core`. However, Rasa will automatically skip training Core or NLU if the training data and config haven't changed.

`rasa train` will store the trained model in the directory defined by `--out`. The name of the model is per default `<timestamp>.tar.gz`. If you want to name your model differently, you can specify the name using `--fixed-model-name`.

Talk to your Assistant

To start a chat session with your assistant on the command line, run:

```
rasa shell
```

Start an Action Server

To run your action server run

```
rasa run actions
```

Json response for front end

go in `credentials.yml` file and add this code snippet:

```
rasa:
  url: "http://localhost:5002/api"
```

command to run rasa chatbot as a API:

rasa run --enable-api

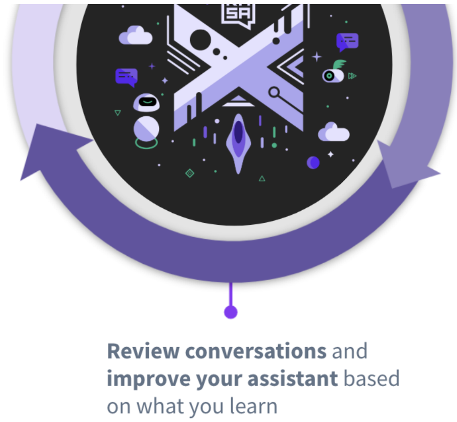
Start Rasa X

Rasa X is a toolset that helps you leverage conversations to improve your assistant. You can find more information about it here.

You can start Rasa X locally by executing

```
rasa x
```

To be able to start Rasa X you need to have Rasa X local mode installed and you need to be in a Rasa project.



HOW TO ADD SPACY NER MODEL TO RASA PIPELINE

By default RASA uses DIET classifier for entity and Intent recognition. But if we want to use SPACY NER model for the same we can.

First of all we'll have to install spacy if not installed:

```
pip install -U spacy
```

then spacy model as a package in the same virtualenv as of RASA and for that the steps are:-

1. python3 -m spacy package [input_dir] [output_dir] --force
2. cd /output/<en_model-0.0.0> here we have to go into the specific folder where our model is lying in the rasa environment.
3. python3 setup.py sdist
4. cd dist
5. python3 -m pip install <en_core_web……tar.gz>

NOTE - [input_dir] is where our ner model resides and [output_dir] is were we want to kept it in rasa enviroment.

Once the package is successfully installed

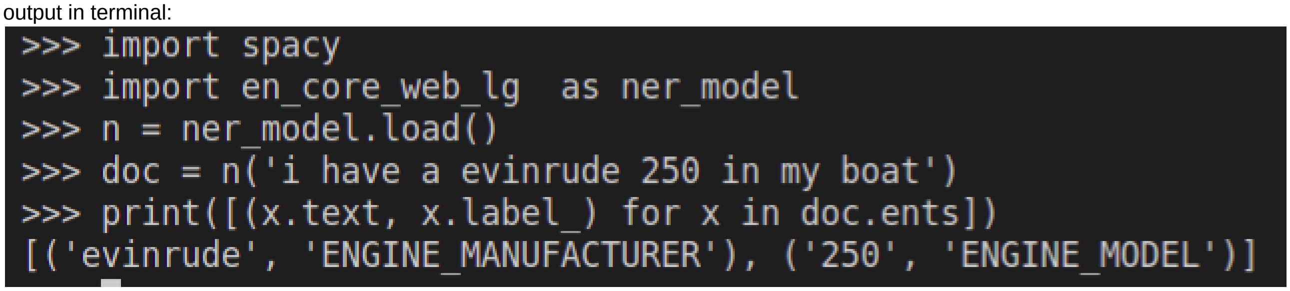
now when we are importing our model named as en_core_web_lg-2.3.1 , what we have to do is:

```
import spacy

import en_core_web_lg as ner_package
```

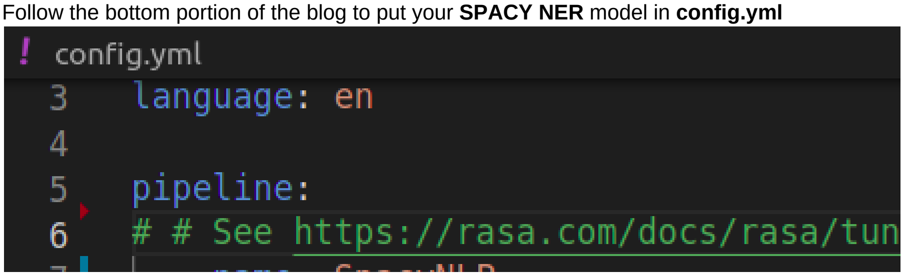
rather than writing the complete name <en_core_web_lg-2.3.1 > while importing.

output in terminal:



```
>>> import spacy
>>> import en_core_web_lg  as ner_model
>>> n = ner_model.load()
>>> doc = n('i have a evinrude 250 in my boat')
>>> print([(x.text, x.label_) for x in doc.ents])
[('evinrude', 'ENGINE_MANUFACTURER'), ('250', 'ENGINE_MODEL')]
```

Follow the bottom portion of the blog to put your **SPACY NER** model in **config.yml**



```
  - name: SpacyNLP
    model: "en_core_web_lg"
  - name: SpacyTokenizer
  - name: SpacyFeaturizer
    pooling: mean
  - name: RegexFeaturizer
  - name: SpacyEntityExtractor
  - name: LexicalSyntacticFeaturizer
  - name: CountVectorsFeaturizer
  - name: CountVectorsFeaturizer
    analyzer: char_wb
    min_ngram: 1
    max_ngram: 4
  - name: DIETClassifier
```

Let's note a few things here;

1. The first step in the pipeline tells us that we're going to use the `en_core_web_lg` model in spaCy. This is equivalent to calling `spacy.load("en_core_web_lg")` which means that you need to make sure that it is downloaded beforehand via `python -m spacy download en_core_web_lg`.
2. Because we're using the spaCy model we now also have to use the tokenizer from spaCy. We do this in the second pipeline step.
3. In the third step we're telling spaCy to also generate word embeddings. We take the mean of all these embeddings such that a single array is passed to the later steps.
4. In the fourth step we're telling spaCy to detect entities on our behalf.
5. In the next steps we generate some features using the `CountVectorsFeaturizer` that will be passed to the `DIETClassifier`.

We can train this pipeline and talk to it to see what the effect is

now firstly we give input with incorrect spellings of intent and what we observed was that only DIET was able to detect that entity

```
Your input ->  need some hulll reapir
There are two scenarios where a hull can be damaged – one is when the hull is damaged above the waterline, the second when it is damaged below
 the waterline. For first scenario take it out and dry it thoroughly.For hull repair, a basic fibreglass repair kit is used, using which the d
amaged section is removed in a circular cut. The part can be then patched using either fibreglass and the proper adhesives or the putties avai
lable
Can i help you some other way.
```

```
['hulll']
New total info is  ['hull']
```

But when we provide the correct spellings both DIET and Spacy were able to extract/detect the entity.

```
Your input ->  need somr HULL repair
There are two scenarios where a hull can be damaged – one is when the hull is damaged above the waterline, the second when it is damaged below
 the waterline. For first scenario take it out and dry it thoroughly.For hull repair, a basic fibreglass repair kit is used, using which the d
amaged section is removed in a circular cut. The part can be then patched using either fibreglass and the proper adhesives or the putties avai
lable
Thanks can I help you some other way.
```

```
['hull', 'hull']
New total info is  ['hull', 'hull']
```

How to handle large file storage on git using git lfs:

INSTALL curl required package to install git lfs

`curl -s https://packagecloud.io/install/repositories/github/git-lfs/script.deb.sh | sudo bash`

GIT LARGE FILE STORAGE

1. Download and install the Git command line extension. Once downloaded and installed, set up Git LFS for your user account by running:

`git lfs install`
You only need to run this once per user account.

2. In each Git repository where you want to use Git LFS, select the file types you'd like Git LFS to manage (or directly edit your .gitattributes). You can configure additional file extensions at anytime.

   `git lfs track "*.tar.gz"` < `.tar.gz`> is extension of our model.

3. Now make sure .gitattributes is tracked:

   `git add .gitattributes`

4. Just commit and push to GitHub as you normally would; for instance, if your current branch is named `main`:

   `git add file.tar.gz git commit -m "Add design file" git push origin main`

~~NOTE~~ ~~WE HAVE MADE AN PROCESS ENTTY IN RASA FOR DIET NEED TO BE REMOVED/CHANGED WITH SOME OTHER ENTITY.~~