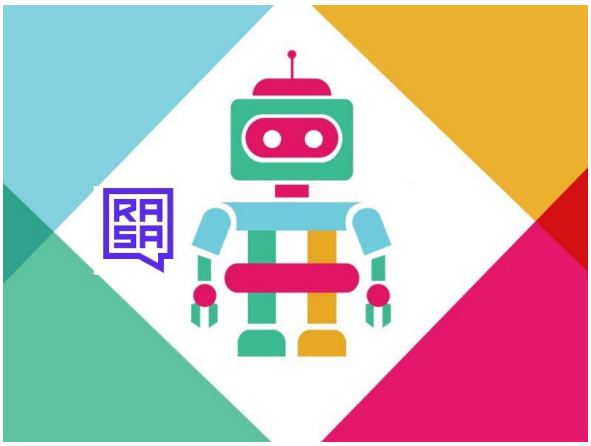
Building Your Own Chatbot: The Hard Way Made Easy



In this workshop, you will learn how to build your own conversational AI assistant using machine learning and real conversational data. The goal of this workshop is to walk you through the process of building an ML-powered assistant from scratch and build an actual assistant which you can improve later.

There are no additional requirements to run this notebook. But if you encounter any issues or have more questions about the content included here, feel free send a message to following email address at gamagebimsara@gmail.com (mailto:gamagebimsara@gmail.com)

Introduction

During the course of this 3-hour workshop, you will go through each stage of the chatbot development and build an assistant capable of providing real-time weather information of a given location. Below is an example conversation your assistant will be able to handle:

U: Hello
A: Hey, how can I help?
U: What's the weather outside?
A: Where are you based?
U: Tell me the current weather in Colombo.
A: Sure, it is currently rainy in Colombo.
U: Goodbye.

A: Goodbye :(

The workshop consists of the following stages:

---

0. Intro:

0.1 Setup and installation

---

1. Stage 1: Natural language understanding:

1.1. Designing the happy path
1.2. Generating the NLU training examples
1.3. Designing the training pipeline

2. Stage 2: Dialogue management model:

2.1. Designing training stories
2.2. Creating a custom action
2.3. Defining the domain

---

3. Stage 3: Training and testing the model:

3.1. Training the bot
3.2. Testing the bot in the terminal
3.3. Model evaluation

---

4. Stage 4: Closing the feedback loop:

4.1. Improving the assistant using the interactive learning

---

5. Stage 5: Integrating the chatbot with Slack:

5.1. Setup the endpoint and credentials
5.2. Ngrok for local testing
5.3. Start the rasa core server

---

0. Intro

In this section, we will install all the necessary dependencies needed to successfully run this exercise.

0.1. Setup and installation

The best way to insall the necessary modules is to use the requirements.txt file. After creating a virtual environment, run:

pip install -r requirements.txt

Throughout this workshop, we will use only open source tools. The code block below checks if Rasa NLU and Rasa Core have been installed suffessfully.

In [ ]:
```
requirements = """
rasa[spacy]
git+https://github.com/apixu/apixu-python.git
ngrok
"""
%store requirements > requirements.txt
```

In [ ]:
```
!pip install -r requirements.txt
```

In [ ]:
```
import rasa.nlu
import rasa.core
import warnings
warnings.filterwarnings('ignore')


print("rasa_nlu: {} rasa_core: {}".format(rasa.nlu.__version__, rasa.core.__ve
```

You should also download **Spacy's English language model**.

In [ ]:
```
!python -m spacy download en_core_web_md
```

In [ ]:
```
!python -m spacy link en_core_web_md en
```

Change the working directory to a newly created one where the necessary project files should be created.

In [ ]:
```
!mkdir bot
```

In [ ]:
```
%cd bot
```

In [ ]:
```
!pwd
```

The first step is to create **a new Rasa project**. To do this, run:

rasa init --no-prompt

The **rasa init** command creates all the files that a Rasa project needs and trains a simple bot on some sample data. If you leave out the **--no-prompt** flag you will be asked some questions about how you want your project to be set up.

In [ ]:
```
!rasa init --no-prompt
```

1. Natural Language Understanding

In this section, you will enable your assistant to understand the user inputs by building a Rasa NLU model. This model will take unstructured user inputs and extract structured data in a form of intents and entities:

- *intent* - a label which represents the overall intention of the user 's input
- *entity* - important detail which an assistant should extract and use to steer the conversation

1.1. Designing a happy path

A good starting point is to define a happy path first. A happy path is a conversation flow where the user provides all the required information and allows the assistant to lead the conversation.

1.2. Designing the NLU training data

To train the NLU model you will need some labeled training data. Rasa NLU training data samples consist of the following components:

- intent label which starts with a prefix *
- examples of text inputs which correspond to that label
- entities which follow the format *[entity_value] (entity_label)*

We will start by generating some training data examples by hand. For a completed data file check out the *helper_files/nlu_data.md* in the repository of this exercise.

In [ ]:
```
%%writefile data/nlu.md
## intent:greet
- Hello
- hey
- hello
- heya
- howdy
- hello there
- hi
- hello there
- good morning
- good evening
- moin
- hey there
- let's go
- hey dude
- goodmorning
- goodevening
- good afternoon

## intent:goodbye
- cu
- good by
- cee you later
- good night
- good afternoon
- bye
- goodbye
- have a nice day
- see you around
- bye bye
- see ya
- see you later

## intent:inform
- What's the weather today?
- What's the weather in [London](location) today?
- Show me what's the weather in [Paris](location)
- I wonder what is the weather in [Vilnius](location) right now?
- what is the weather?
- Tell me the weather
- Is the weather nice in [Barcelona](location) today?
- I am going to [London](location) today and I wonder what is the weather out
- I am planning my trip to [Amsterdam](location). What is the weather out ther
- Show me the weather in [Dublin](location), please
- in [London](location)
- [Lithuania](location)
- Oh, sorry, in [Italy](location)
- Tell me the weather in [Vilnius](location)
- The weather condition in [Italy](location)
```

1.3 Designing the training pipeline

Once the training data is ready, we can define the NLU model. We can do that by constructing the processing pipeline which defines how structured data will be extracted from unstructured user inputs: how the sentences will be tokenized, what intent classifier will be used, what entity extraction model will be used, etc. Each component in a training pipeline is trained one after another and can take inputs from the previously defined component as well as pass some information to subsequent ones.

In [ ]:
```
%%writefile config.yml
# Configuration for Rasa NLU.
# https://rasa.com/docs/rasa/nlu/components/
language: "en"
#pipeline: "pretrained_embeddings_spacy"
pipeline:
- name: "SpacyNLP"                    # loads the spacy language model
- name: "SpacyTokenizer"              # splits the sentence into tokens
- name: "SpacyFeaturizer"             # transform the sentence into a vector r
- name: "RegexFeaturizer"
- name: "CRFEntityExtractor"
- name: "EntitySynonymMapper"         # trains the synonyms
- name: "SklearnIntentClassifier"     # uses the vector representation to clas

# Configuration for Rasa Core.
# https://rasa.com/docs/rasa/core/policies/
policies:
  - name: MemoizationPolicy
  - name: KerasPolicy
  - name: MappingPolicy
```

2. Dialogue Management

In this section of this workshop you will build a machine learning-based dialogue model which will enable your assistant to decide on how to respond to user inputs based on the state of the conversation.

2.1 Designing the training stories

Let's start with generating the training data. Rasa Core models learn by observing real conversational data between the user and the assistant. The only important thing is that this data has to be converted into the Rasa Core format: user inputs have to be expressed as corresponding intents (and entities where necessary) while the responses of the assistant are expressed as action names. Each training story follows the format:

- the story starts with a story name which has a prefix ##
- intents, corresponding to user inputs, start with *
- if NLU model extracts entities which should influence the predictions of the dialogue model, they have to be included in the stories using the following format: * intent{'entity_name':"entity_value"}
- the responses of the bot start with -
- the story ends with an empty line which marks the end of the story

In the next step of this tutorial, we will generate some training stories to cover the happy path. To see a complete training data example, check out the **data/stories.md** file of this repository.

In [ ]:

```
%%writefile data/stories.md
## Generated Story 3320800183399695936
* greet
    - utter_greet
* inform
    - utter_ask_location
* inform{"location": "italy"}
    - slot{"location": "italy"}
    - action_weather
    - slot{"location": "italy"}
* goodbye
    - utter_goodbye
    - export
## Generated Story -3351152636827275381
* greet
    - utter_greet
* inform{"location": "London"}
    - slot{"location": "London"}
    - action_weather
* goodbye
    - utter_goodbye
    - export
## Generated Story 8921121480760034253
* greet
    - utter_greet
* inform
    - utter_ask_location
* inform{"location":"London"}
    - slot{"location": "London"}
    - action_weather
* goodbye
    - utter_goodbye
    - export
## Generated Story -5208991511085841103
    - slot{"location": "London"}
    - action_weather
* goodbye
    - utter_goodbye
    - export
## Generated Story -5208991511085841103
    - slot{"location": "London"}
    - action_weather
* goodbye
    - utter_goodbye
    - export
## story_001
* greet
   - utter_greet
* inform
   - utter_ask_location
* inform{"location":"London"}
   - slot{"location": "London"}
   - action_weather
* goodbye
   - utter_goodbye
## story_002
* greet
   - utter_greet
* inform{"location":"Paris"}
   - slot{"location": "Paris"}
   - action_weather
* goodbye
```

```
   - utter_goodbye
## story_003
* greet
   - utter_greet
* inform
   - utter_ask_location
* inform{"location":"Vilnius"}
   - slot{"location": "Vilnius"}
   - action_weather
* goodbye
   - utter_goodbye
## story_004
* greet
   - utter_greet
* inform{"location":"Italy"}
   - slot{"location": "Italy"}
   - action_weather
* goodbye
   - utter_goodbye
## story_005
* greet
   - utter_greet
* inform
   - utter_ask_location
* inform{"location":"Lithuania"}
   - slot{"location": "Lithuania"}
   - action_weather
* goodbye
   - utter_goodbye
```

2.2 Creating custom action

We are going to use the backend integration to enable our assistant to fetch the relevant data based on user's queries. For that, we will create custom actions which, when predicted, will collect necessary data and use it to steer the conversation further:

In [ ]:
```
%%writefile actions.py
from __future__ import absolute_import
from __future__ import division
from __future__ import unicode_literals

from rasa_sdk import Action
from rasa_sdk.events import SlotSet

class ActionWeather(Action):
    def name(self):
        return 'action_weather'

    def run(self, dispatcher, tracker, domain):
        from apixu.client import ApixuClient
        api_key = '' #your apixu key
        client = ApixuClient(api_key)

        loc = tracker.get_slot('location')
        current = client.getcurrent(q=loc)

        country = current['location']['country']
        city = current['location']['name']
        condition = current['current']['condition']['text']
        temperature_c = current['current']['temp_c']
        humidity = current['current']['humidity']
        wind_mph = current['current']['wind_mph']

        response = """It is currently {} in {} at the moment. The temperature

        dispatcher.utter_message(response)
        return [SlotSet('location',loc)]
```

2.3 Defining the domain

Once we have the training data in place, we can define the domain of our assistant. A domain defines the environment in which the assistant operates - what user inputs it should expect to see, what actions it should be able to predict, what information the assistant should store throughout the conversation.

In [ ]:
```
%%writefile domain.yml
slots:
  location:
    type: text


intents:
 - greet
 - goodbye
 - inform


entities:
 - location

templates:
  utter_greet:
    - text: 'Hello! How can I help?'
  utter_goodbye:
    - text: 'Talk to you later.'
    - text: 'Bye bye :('
  utter_ask_location:
    - text: 'In what location?'


actions:
 - utter_greet
 - utter_goodbye
 - utter_ask_location
 - action_weather
```

3. Training and testing the model

We now have all the components necessary to train our first model.

3.1. Training the bot

The code cell below will train the model using the defined pipeline and policies and store the model in a specified location for us to test later.

In [ ]:
```
!rasa train
```

3.2. Testing the bot in the terminal

Before testing the bot, first we need to start our **action server** in **a new terminal**. Following command will start the action server.

rasa run actions

Following script will loads your trained model and lets you talk to your assistant on the command line **(You may need to run this in a new terminal)**.

rasa shell

3.3. Model evaluation

Another great way to see how good our model is, is to test it using evaluation script:

In [ ]:
```
!rasa test
```

You can visualize your training stories using the following script:

In [ ]:
```
!rasa visualize
```

4. Closing the feedback loop

Developing an assistant is just one part of the process. Another very important part which defines a successful assistant is enabling your assistant to learn from real user feedback. In the last part of this workshop, we will cover two ways to improve your bots using real user feedback - using interactive learning and using the history of the conversations. We will also, connect our assistant to a custom webpage to see how it works in action! We will complete this part using the command line.

4.1. Improving the assistant using the interactive learning

Interactive learning is a great way to improve your assistant and generate more training example by simply talking to your bot and providing feedback for all predictions it made. That is the main idea behind it - instead of responding right away, an assistant will tell you what it thinks it should do next and ask you for feedback. To start the interactive learning session, we will use a command line and use the following command:

rasa interactive

5. Integrating the chatbot with Slack

In the very last step of this workshop, you will learn how to connect your assistant to slack messaging platform. Rasa already supports many other messaging platforms and also allows you to implement your custom channel.

5.1. Setup the endpoint and credentials

You need to change **endpoint.yml** and **credentials.yml** files as bellow.

endpoints.yml

In [ ]:
```
%%writefile endpoints.yml
action_endpoint:
  url: "http://localhost:5055/webhook"
```

credentials.yml

In [ ]:
```
%%writefile credentials.yml
# This file contains the credentials for the voice & chat platforms
# which your bot is using.
# https://rasa.com/docs/rasa/user-guide/messaging-and-voice-channels/

rest:
#  # you don't need to provide anything here - this channel doesn't
#  # require any credentials


#facebook:
#  verify: "<verify>"
#  secret: "<your secret>"
#  page-access-token: "<your page access token>"

slack:
 slack_token: "paste your slack token here (xoxb something something)"
#  slack_channel: "<the slack channel>"

#socketio:
#  user_message_evt: <event name for user message>
#  bot_message_evt: <event name for but messages>
#  session_persistence: <true/false>

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

5.2. Ngrok for local testing

Ngrok is a multi-platform tunnelling, reverse proxy software that establishes secure tunnels from a public endpoint such as the internet to a locally running network service. In simple words it means, it opens access to your local app from the internet.

You need to start a ngrok in **a new terminal**. Start it by telling it which port we want to expose to the public internet.

ngrok http 5005

5.3. Start the rasa core server

After setting up the backend, we will start our assistant on a server and connect to the UI using:

rasa run --endpoints endpoints.yml