EURECOM
Campus SophiaTech
CS 50193
06904 Sophia Antipolis cedex
FRANCE

Developing a Classical Music Chatbot With RASA

June 25th, 2019

Daniel HOMS ALFONSÍN
Supervisors:
Raphaël Troncy
Pasquale Lisena
Thibault Ehrhart

Tel : (+33) 4 93 00 81 00
Fax : (+33) 4 93 00 82 00
Email: homs@eurecom.fr

Introduction

Description and goals of the project

This project consists of the development of a musical chatbot using RASA Stack which is an open source framework used in the development of AI assistants as chatbots, this together with DOREMUS, which is a classical music knowledge graph developed at EURECOM, the information of this knowledge graph can be accessed using a SPARQL endpoint.

The goal of this project is to familiarize with these technologies used in the development of the chatbot, to understand how these technologies and different libraries dedicated to entity recognition allows us to be able to comprehend users conversations and how to retrieve the information related to classical music from the conversations and from DOREMUS endpoint.

Importance of chatbots

The usage of AI assistants is increasing in the last years, mainly because using these virtual assistants is an easier way for people to engage in some service, through a natural conversation users can retrieve information they want to know or do some tasks, as it would be to ask for directions or schedule an appointment. Most of the time this tasks can be done without the help of an assistant, but the idea of doing it more naturally is appealing to people, as it is a faster approach. One example would be telling a virtual assistant as *Alexa* to add to your cart a product you want to buy on *Amazon*, rather than going to their website, searching for the product and adding it to your cart. Now, there are different ways of providing this experience, one of this would be through voice assistants, as *Alexa* or *Google Assistant*, another way of providing this assistance to users is by using chatbots, through messaging apps, as *Facebook Messenger* or *Slack*, to deliver the final user some customized help.

There are a lot of companies that are developing bots in order to provide their customers with a better experience when using their services or buying their products, this is another reason why the development of this kind of services is growing, instead of only offering your customers a normal *Frequently Asked Questions* site you can have, as an experience enhancer, a chatbot, where the user can go and ask some questions about what the company offers and request any information related to it.

In the case of this project, the idea is of developing a chatbot that can provide users interested in classical music an easier way to access to this information through a conversation.

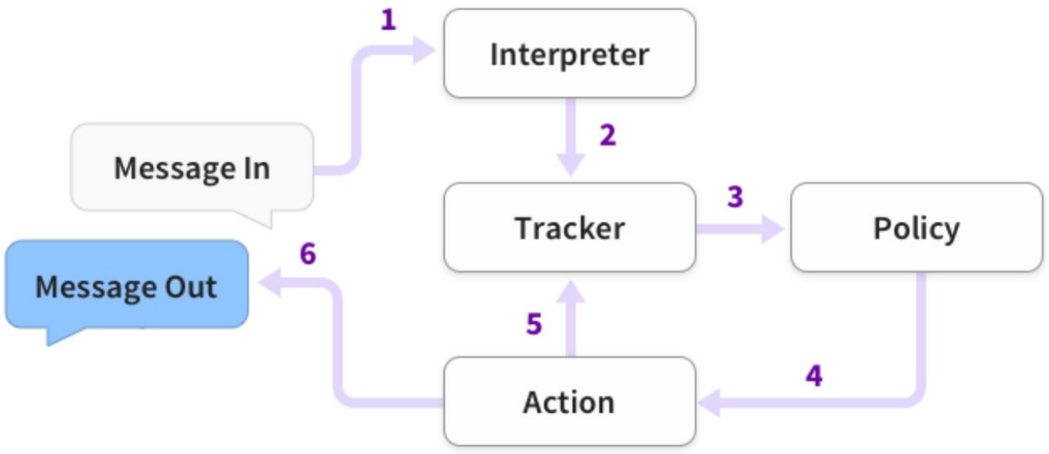
Technologies

RASA Stack

The RASA Stack is an open source machine learning framework used to build conversational AI as assistants and chatbots. This framework is divided in two main components: RASA Core and RASA NLU.

The Core is the machine learning framework focused on managing the dialogue, the NLU is a library for natural language understanding with intent classification and entity extraction. These components are independent of each other.

The following diagram shows the general overview of how the RASA Stack manages a message:



Source: http://rasa.com/docs/rasa/user-guide/architecture/

The message is passed to the Interpreter, which is in charge of classifying the intents and extracting the entities, this is part of the NLU. The tracker keeps the conversation state, and receives the information when a new message is received. After that, the policy receives the current state of the tracker and chooses which next action to take. Then the action taken is logged by the tracker and the response is sent to the user.

RASA NLU

NLU is Natural Language Understanding. The RASA NLU takes a sentence as an input from the user and understands the message based on previous training data. It recognizes and classifies intents and extracts the entities from the sentence.

Here we have a sentence as example:

```
"I am looking for a Mexican restaurant in the center of town"
```

This sentence will return a structured data like:

```
{
  "intent": "search_restaurant",
  "entities": {
    "cuisine" : "Mexican",
    "location" : "center"
  }
}
```

Source: https://rasa.com/docs/rasa/nlu/about/

The NLU has defined a pipeline which contains different components in order to process the messages sequentially and classify them into intents and extract the entities. As it is possible to see in the example, the sentence was classified as a *search_restaurant* intent, extracting the entity *Mexican* and a location *center*.

Intents

The concept of intents is used to describe how the message sent by the user should be categorized. This message can be classified into one or multiple intents.

There are two components in here from where to choose:

Pretrained Embeddings (Intent Classifier Sklearn):

This classifier uses the spaCy library that contains pretrained language models used to represent each word in the user message as word embedding. This word embeddings are vector representations of words, so each one of these is converted into a numeric vector, capturing semantic and synaptic aspects of words.

RASA NLU takes the average of all the words embeddings and performs a grid search to find the best parameters for the support vector classifier which classifies the average embeddings into different intents.

Supervised Embeddings (Intent Classifier TensorFlow Embedding):

This intent classifier was developed by RASA and instead of using pre-trained embeddings it trains word embeddings from scratch. The tensorflow embedding classifier supports messages with multiple intents, in contrast to the previous one.

Because this classifier trains words embeddings from scratch the amount of training data needed for the classifier has to be bigger so that the classifier can generalize well.

Entity extraction

When a user introduces a message it is needed to obtain the relevant information out of this, such as addresses or dates. The process of extracting the different pieces of information is called entity recognition. The entities are this pieces of information needed to understand what the user is talking about.

Depending on which entity type is needed to be extracted there are different components that the RASA NLU can use in order to achieve this. The entity extractors used for this project are going to be explained deeper in the Libraries section of this report, but here are some examples of the different entity extractors available for RASA:

- SpaCy: Entity recognition with language models
- Duckling: Rule based entity recognition
- CRF: Extractor for custom entities (using conditional random field)
- MITIE: Extractor for custom entities (structured SVM)

RASA Core

The RASA Core is a dialogue engine using a machine learning model trained on example conversations, this example conversations are called *stories*. The user's input is represented by intents and entities, and the chatbot responses are expressed by actions.

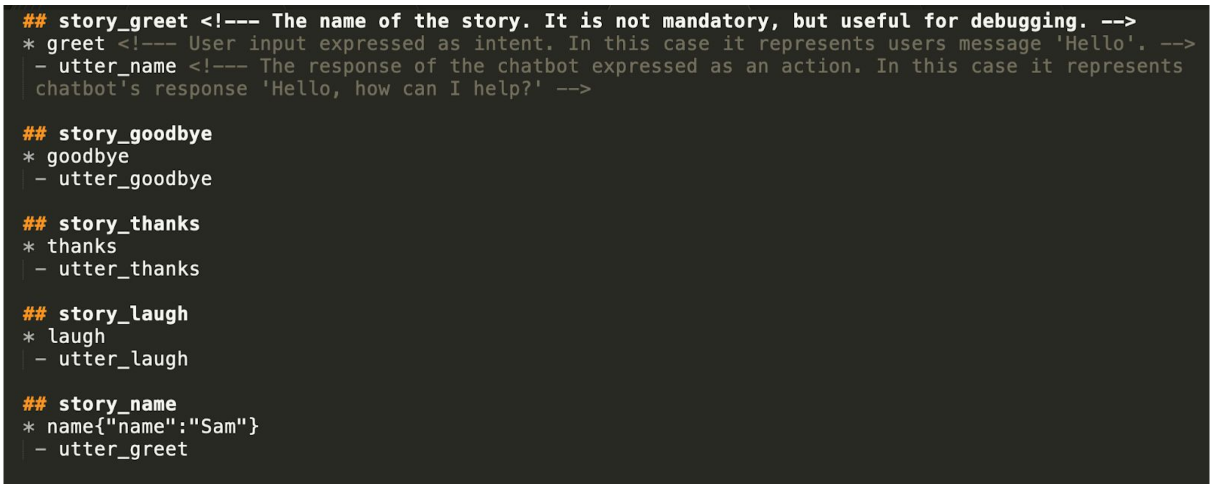
Stories

Stories represent a guide where each intent, and it's entities, have a correspondent action as response. Each story has:

- A name, preceded by two hashes ("##").
- The message sent by the user, represented with the name of the intent and the entities to be extracted from it, with the format *intent{"entity1": "value", "entity2": "value"}*.

- Actions to be executed by the bot given a specific intent.
- Events returned by an action are on lines immediately after that action.

Here is an example of different stories:

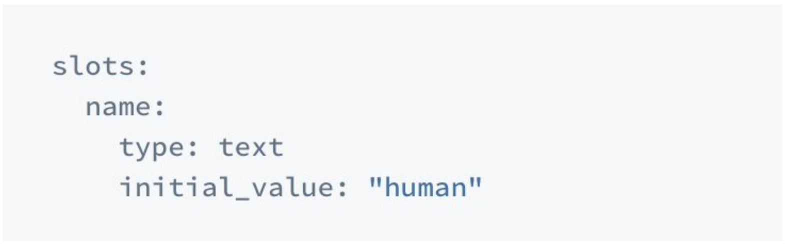


```
## story_greet <!--- The name of the story. It is not mandatory, but useful for debugging. -->
* greet <!--- User input expressed as intent. In this case it represents users message 'Hello'. -->
 - utter_name <!--- The response of the chatbot expressed as an action. In this case it represents
 chatbot's response 'Hello, how can I help?' -->

## story_goodbye
* goodbye
 - utter_goodbye

## story_thanks
* thanks
 - utter_thanks

## story_laugh
* laugh
 - utter_laugh

## story_name
* name{"name":"Sam"}
 - utter_greet
```

Slots

Slots represent the memory of the bot, they hold the information that is relevant for the bot in order to keep track of the conversation. They act as a key-value store which can be used to store information the user provided (e.g their home city) as well as information gathered about the outside world (e.g. the result of a database query).

There are different slot types, which depends of the expected value, this types could be text, floats, lists, categorical, boolean or unfeaturized. The slots are set as:



```
slots:
  name:
    type: text
    initial_value: "human"
```

Source: http://rasa.com/docs/rasa/core/slots/

Actions

The actions are the bot response to user input. There are 3 different types of actions in RASA:

- **Default actions**: This actions that are already defined in RASA as *action_listen, action_restart, action_default_fallback*.

- **Utterance actions:** This are messages that are sent to the user after a given intent, they start with *utter_*.
- **Custom actions:** These actions are defined by the developer and can run an arbitrary code. The Core calls an endpoint specified by the developer when this action gets predicted.

Domain

The domain defines the universe of the chatbot, it determines what the bot should understand, what it can do and what information is necessary for the chatbot context. It has declared the intents, entities, slots, actions and templates that it needs to know about and it is defined in yaml format. The templates are defined messages for utterance actions.

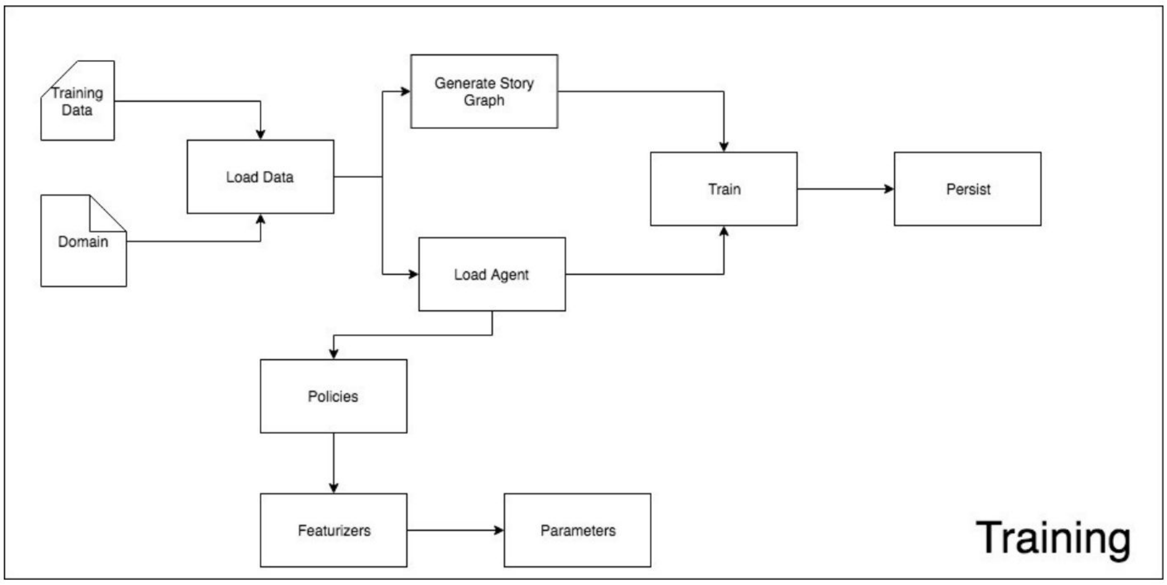
An example of a domain file for a bot:

```
intents:
  - greet
  - goodbye
  - affirm
  - deny
  - mood_great
  - mood_unhappy

actions:
- utter_greet
- utter_cheer_up
- utter_did_that_help
- utter_happy
- utter_goodbye

templates:
  utter_greet:
  - text: "Hey! How are you?"

  utter_cheer_up:
  - text: "Here is something to cheer you up:"
    image: "https://i.imgur.com/nGF1K8f.jpg"

  utter_did_that_help:
  - text: "Did that help you?"

  utter_happy:
  - text: "Great carry on!"

  utter_goodbye:
  - text: "Bye"
```

Source: http://rasa.com/docs/rasa/core/domains/

Training

The following image represents all the steps needed to train the RASA Core, this is followed by the explanation of each process:



Training Steps

Source: https://medium.com/strai/contextual-conversational-engine-the-rasa-core-approach-part-1-1acf95b3d237

First the Core loads the data, this being the information contained in the training data, this are, as mentioned previously, the stories defining the different conversation paths that a user can have with the assistant, and the domain, this representing all the information that the bot needs to know and to keep track of.

The Agent is the class that manages the training, handling of messages, loading a dialogue model, getting the next action, and handling a channel. So the agent is loaded with some parameters that determine how the training data is going to be converted in features for the training agent, one of this parameters is the policy. The Policy class is in charge of deciding which next action should be executed at the next step of the conversation. The default policies used are:

- **Memoization Policy:** Memorizes the stories and uses them to predict the next action. In here the prediction is binary, so if the conversation matches a story the confidence of the prediction will be 1, if not it will be 0.
- **Keras Policy:** This policy uses a LSTM neural network to predict the next possible action.

In the featurization step the conversation gets represented as a vector, so each state of the conversation will contain features with the intents, entities, slots and previous actions. There is one very important hyperparameter to take into account before providing all the conversation features to feed the policies, this is the Max History parameter, this determines how much in the past of the conversation does the policy needs to look back to predict the next step of the conversation.

For the training step, the policies defined previously are fed to the training process, more than one policy can be given to the process, each policy will be used to train the model separately and they will be used together to predict the next action the bot needs to take for the next step. And once the training is completed the model is saved in a file in order for it to be persistent.

Libraries

SPARQLWrapper

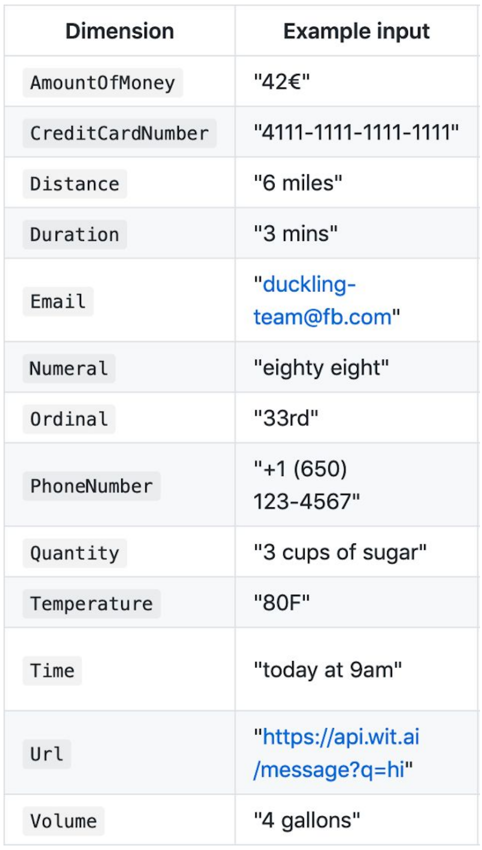
This library is a Python wrapper around a SPARQL service to remotely execute queries. It helps in creating the query URI and, possibly, convert the result into a more manageable format, for example JSON or XML.

SpaCy

SpaCy is a free, open-source library for advanced Natural Language Processing (NLP) in Python, it offers pretrained entity extractors and it is good for extracting places, dates, people and organizations.

Duckling

Duckling is a rule-based entity extraction library developed by wit.ai that parses text into structured data. RASA provides a docker image *rasa/duckling* in order to be able to use duckling for entity extraction using the REST interface of duckling, this component has to be declared into the NLU pipeline as *ner_duckling_http*. It has supports for different languages and different dimensions:



Dimension	Example input
`AmountOfMoney`	"42€"
`CreditCardNumber`	"4111-1111-1111-1111"
`Distance`	"6 miles"
`Duration`	"3 mins"
`Email`	"duckling-team@fb.com"
`Numeral`	"eighty eight"
`Ordinal`	"33rd"
`PhoneNumber`	"+1 (650) 123-4567"
`Quantity`	"3 cups of sugar"
`Temperature`	"80F"
`Time`	"today at 9am"
`Url`	"https://api.wit.ai/message?q=hi"
`Volume`	"4 gallons"

Source: https://github.com/facebook/duckling

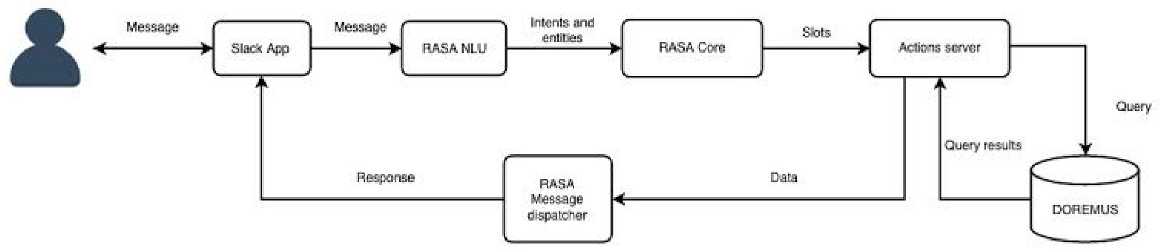
Pendulum

Pendulum is a Python library that helps with easy datetime manipulation.

Music Chatbot with DOREMUS

Description

Architecture of the chatbot



The Chatbot is using both RASA components, the RASA NLU and the RASA Core. The user inputs the message into the Slack app connected to the chatbot, this will send the message to the chatbot server using a webhook, the message is processed using the NLU in order to classify it into an intent and extract the defined entities of the message and save them into slots, after identifying the intent, depending on to what story this belongs to, the Core will determine which action to do, the actions are in charge of getting the entities from the slots and realize a query to the DOREMUS SPARQL endpoint, the endpoint will answer with the requested data and this is managed by using the Message dispatcher from RASA in order to deliver this as a message through the slack app to the user.

Intents, Entity extraction and Stories of the Music Chatbot

Intents

The defined intents relevant to the project are the following:

- **discover_artist**

Retrieves the information about a specific artist, requested by name of the artist by the user, giving the birth/death place and date and a little biography of the artist.

- **works_by**

This intent receives a request to retrieve a set of works for a given artist, but this will trigger the question to add or not more filters to the request.

- **works_by_no**

After requesting a set of works by an artist, if the user does not want to add more filters, the query to retrieve the set of works is sent with the initial given filters.

- **works_by_yes**

After requesting a set of works by an artist, if the user decides to add more filters, the user is asked to provide this extra filters.

- **works_by_filter**

If the user decides to add more filters, this intent is responsible of receiving the filters and retrieving the set of works.

- **find_performance**

Search for a performance (future or past) in a given city and in a given date, the user will receive the details about the available performances with the given filters.

- **find_artist**

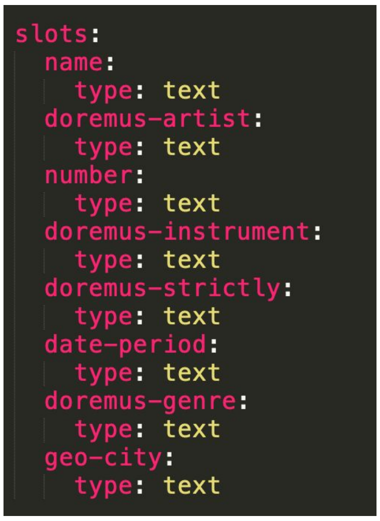
Finds a set of artists according to the given filters, these filters could be works for an instrument, a specific genre or the birthplace of the artists.

Entities

The entities declared and that are needed to be extracted are:

- doremus-artist
- number
- geo-city
- date-period
- doremus-genre
- doremus-strictly
- doremus-instrument

In this case, one slot was declared per each entity to store each one of the entities to extract:

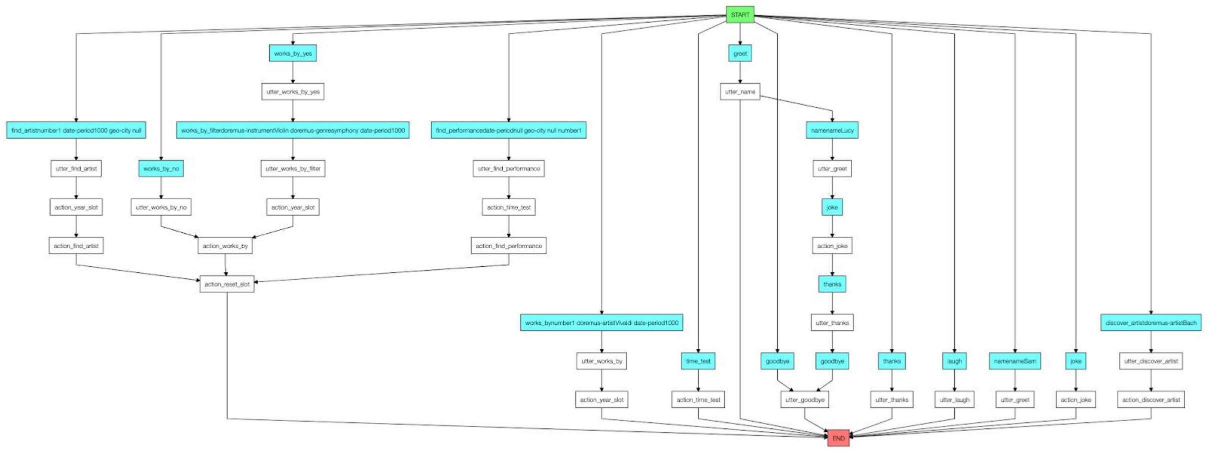


```
slots:
  name:
    type: text
  doremus-artist:
    type: text
  number:
    type: text
  doremus-instrument:
    type: text
  doremus-strictly:
    type: text
  date-period:
    type: text
  doremus-genre:
    type: text
  geo-city:
    type: text
```

For the extraction of the entities doremus-artist, number, geo-city, doremus-genre, doremus-strictly and doremus-instrument the default entity extractors used in the project are spaCy and NER CRF. In the case of the date-period entity, the extraction is realized using the Duckling extractor provided in a docker container by RASA. After the extraction this entities are saved in their respective slots in order to be able to retrieve them when executing the actions and realizing the queries.

Stories

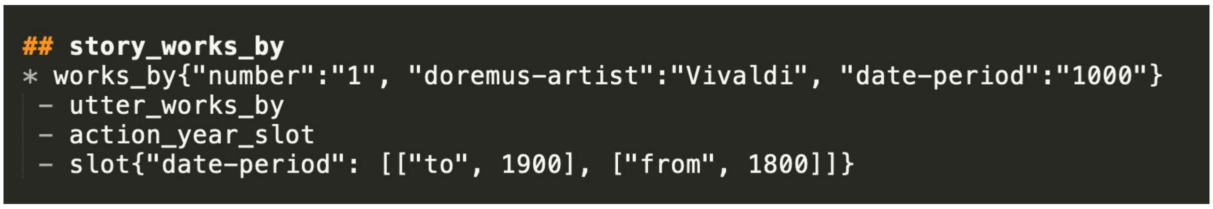
The following image is a graph of all the stories declared in the bot:



In the chatbot there are declared several stories regarding to the different conversations paths the users can take depending on the information they want to retrieve. The stories relevant to be able to do a request to the DOREMUS endpoint are defined as:

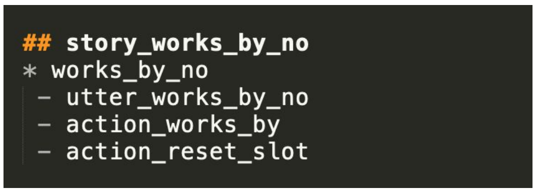
Works by

In here there are 3 stories to be able to identify what the user is requesting. For the first story the user may ask for a set of works composed by an artist, in here the user may specify filters as the number of compositions he wants to retrieve, the name of the artist and a date, in the case of the date it could be a range of years, a specific year or before/after a specific year. In this story the dates are processed as custom slots, this part is going to be explained in the Implementation section. Here is how this story is defined in RASA:



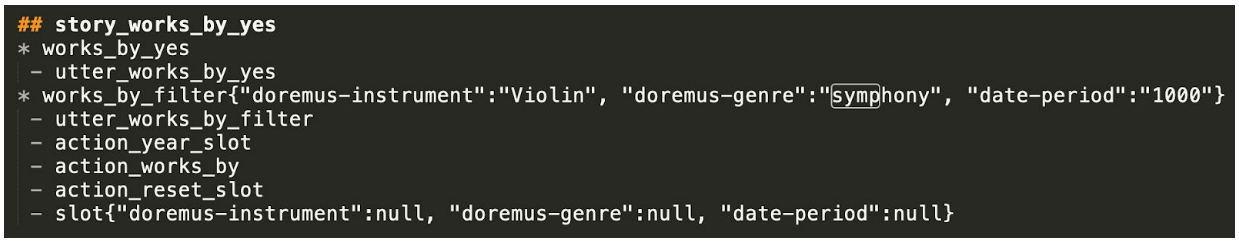
```
## story_works_by
* works_by{"number":"1", "doremus-artist":"Vivaldi", "date-period":"1000"}
  - utter_works_by
  - action_year_slot
  - slot{"date-period": [["to", 1900], ["from", 1800]]}
```

After receiving the request from the user, a utterance message would be sent to the user asking if he wants or needs to add some more filters, in the case it does not want to add any other filter the request is going to be done would the first filters that were specified at the start of the request. This story is defined as:



```
## story_works_by_no
* works_by_no
  - utter_works_by_no
  - action_works_by
  - action_reset_slot
```

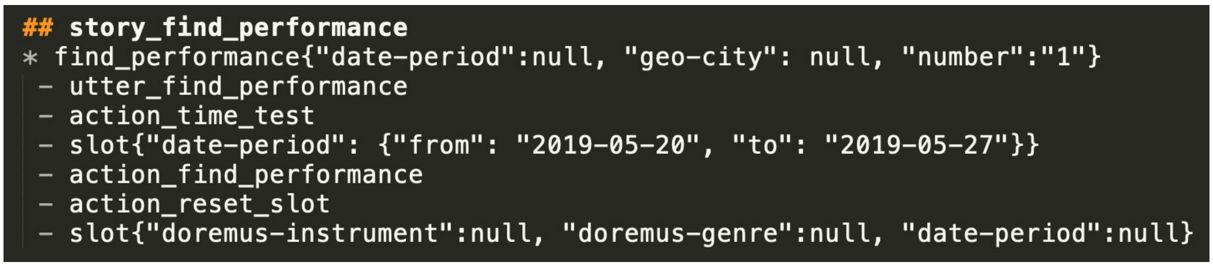
In the case the user wants to add more filters, a new utterance message is going to ask to write the new filters, this could be a genre, an instrument or a date (in case it was not previously mentioned). This story represents this path chosen by the user during the conversation:



```
## story_works_by_yes
* works_by_yes
  - utter_works_by_yes
* works_by_filter{"doremus-instrument":"Violin", "doremus-genre":"symphony", "date-period":"1000"}
  - utter_works_by_filter
  - action_year_slot
  - action_works_by
  - action_reset_slot
  - slot{"doremus-instrument":null, "doremus-genre":null, "date-period":null}
```

Finding a performance

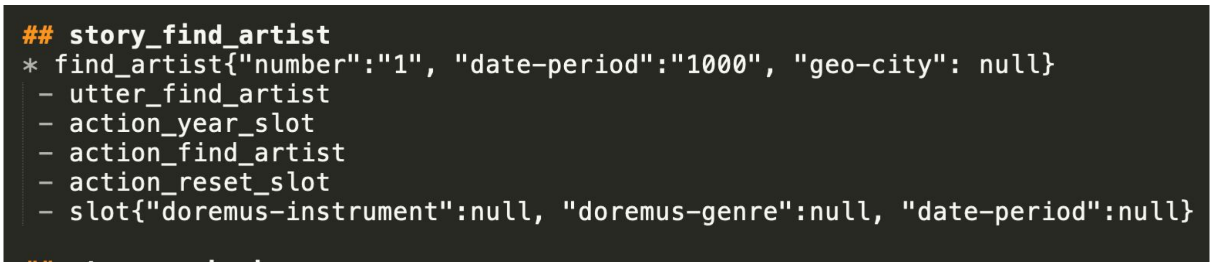
Another story is for when the user wants to search for a performance, this will be classified as a find_performance intent, the user can specify the number of performances it wants to look at, the date range in which he wants the performance to be, this date has a custom slot, and the city where the performance is going to have place. In this case it works for the city of Paris as it is the only city with registered performances at the DOREMUS database, but there should be no problem when trying to retrieve any other city when available in the database. This story is defined as:



```
## story_find_performance
* find_performance{"date-period":null, "geo-city": null, "number":"1"}
  - utter_find_performance
  - action_time_test
  - slot{"date-period": {"from": "2019-05-20", "to": "2019-05-27"}}
  - action_find_performance
  - action_reset_slot
  - slot{"doremus-instrument":null, "doremus-genre":null, "date-period":null}
```

Finding an artist

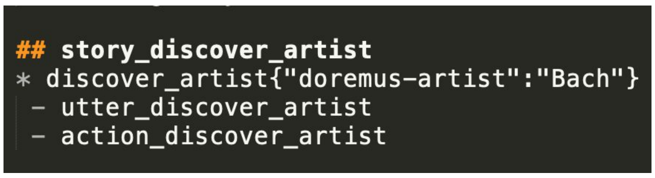
When the user wants to find an artist or a set of artists with some specific constraints, as the period in which they were born and the city where they were born, the genre of their compositions or the instruments used to play those compositions, the NLU will classify this as a find_artist intent, the returned set of artist are ordered by the descending count of works composed in their work life, this story represent this request with the following definition:



```
## story_find_artist
* find_artist{"number":"1", "date-period":"1000", "geo-city": null}
 - utter_find_artist
 - action_year_slot
 - action_find_artist
 - action_reset_slot
 - slot{"doremus-instrument":null, "doremus-genre":null, "date-period":null}
```

Discovering an artist

This story is the simplest of them all, as in this conversation the user asks about a specific artist, the returned response will be the artist data, as their birth and death place and date, and small biography. This story is defined as:



```
## story_discover_artist
* discover_artist{"doremus-artist":"Bach"}
 - utter_discover_artist
 - action_discover_artist
```

Implementation and how to run it

Description

The chatbot code is organized the following way:

- **Makefile** contains the commands needed to train the NLU and the Core, to run the action server, the chatbot in the command line or the chatbot as a server using the port 5005.

- **nlu_config.yml** is the file with the pipeline of the RASA NLU.

- **credentials.yml** has declared the token used for the Slack app, in here all other tokens and credentials should be declared for other messaging apps.

- **endpoints.yml** is the file containing the webhooks for the action server and the slack app.

- **domain.yml** is a file with the defined intents, entities, slots, actions and the templates for the utterance actions.

- **policies.yml** contains the policies for the RASA Core, it has defined the Keras Policy, Fallback Policy, Memoization Policy and a Form Policy.

- **requirements.txt** contains all the libraries needed to be installed in order to be able to run the chatbot.

- **data/** is a directory containing the training data:
    - the stories, stored in the **stories.md** file
    - the **nlu_data.md** containing the intents and the training sentences for the classification of the NLU.

- **actions.py** is the python file where all the actions of the chatbot are declared, also the time slots are processed in this code and the functions to detect misspellings. All the actions go through the SPARQLWrapper library to be able to connect to the DOREMUS SPARQL endpoint.

Actions

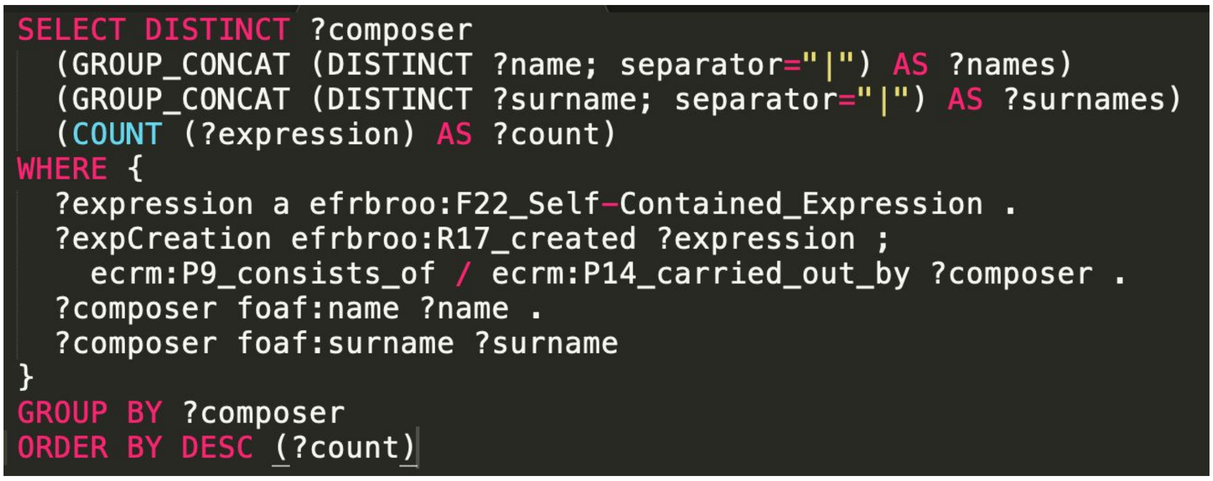
Before proceeding to explain each of the actions the bot realizes and how they are implemented it is important to clarify the processes that some of the entities go through previous to the query execution, this procedures involve the artists, genres, instruments and dates entities.

Artists, genres and instruments

After extracting the entities doremus-artist, doremus-genre and doremus-instrument the obtained value would be the input provided by the user, in this case it could be the name and/or last name of the artist, the name of a genre or an instrument, but this values could be contained differently in the DOREMUS database, obtaining bad results after executing the query, this is why for this entities the process for the actions is the following:

- doremus-artist

This entity is used to train and recognize the artists when a user request a set of works by an artist or when asking for more information about an artist. In the DOREMUS knowledge base the artists have designated an id key, and each artist has associated multiple ways of how this artist name can be written, depending on the language. In order to retrieve this key values the next query was used:

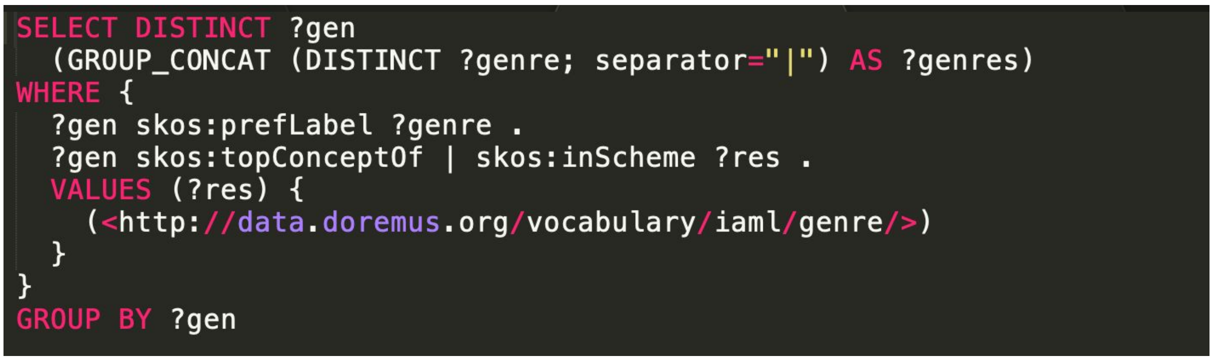


```
SELECT DISTINCT ?composer
  (GROUP_CONCAT (DISTINCT ?name; separator="|") AS ?names)
  (GROUP_CONCAT (DISTINCT ?surname; separator="|") AS ?surnames)
  (COUNT (?expression) AS ?count)
WHERE {
  ?expression a efrbroo:F22_Self-Contained_Expression .
  ?expCreation efrbroo:R17_created ?expression ;
    ecrm:P9_consists_of / ecrm:P14_carried_out_by ?composer .
  ?composer foaf:name ?name .
  ?composer foaf:surname ?surname
}
GROUP BY ?composer
ORDER BY DESC (?count)
```

After retrieving the keys and names for each artist the file is saved in a json format named *artists.json*, in here each entry of the file contains the uri value for each artist and the multiple ways that their names could be written separated by the symbol "|". When executing a query in the actions, the artist name is searched inside this file sequentially, if it is contained, the value for the given artist is retrieved and used, if it is not contained, the returned value would be returned by the most similar name to the one provided by the user.

- doremus-genre

For the genres entity the case is similar to the artists one, doremus-genre is used for the recognition of genres, after extracting the genre inputted by the user it is needed to obtain the key id in the DOREMUS knowledge graph. This is the query used for obtaining the genres registered in DOREMUS:

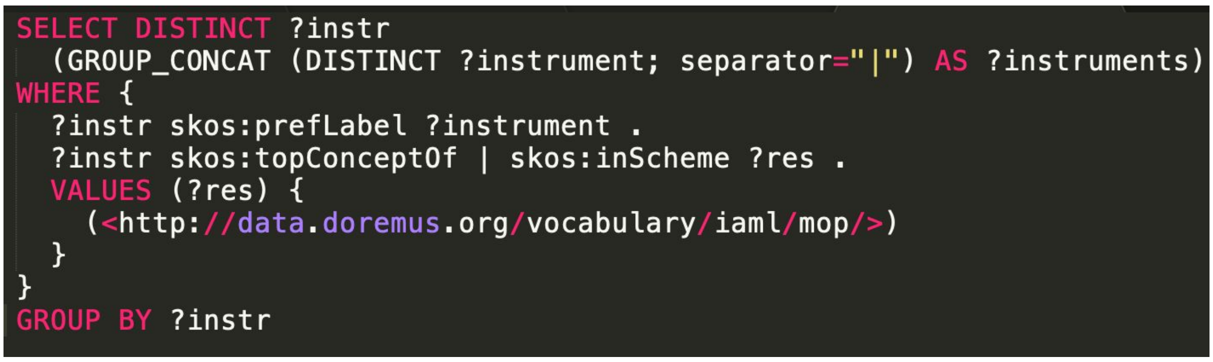


```
SELECT DISTINCT ?gen
  (GROUP_CONCAT (DISTINCT ?genre; separator="|") AS ?genres)
WHERE {
  ?gen skos:prefLabel ?genre .
  ?gen skos:topConceptOf | skos:inScheme ?res .
  VALUES (?res) {
    (<http://data.doremus.org/vocabulary/iaml/genre/>)
  }
}
GROUP BY ?gen
```

Similarly as the artists, the result is saved in a json format file called *genres.json*, this is also searched sequentially and the rest of the procedures goes as explained before.

- doremus-instrument

For instruments we have the same case as the two previous ones.
doremus-instruments is the entity used for the identifying the instruments inputted by the user. The query used to obtain the instruments from DOREMUS is the following:



```
SELECT DISTINCT ?instr
  (GROUP_CONCAT (DISTINCT ?instrument; separator="|") AS ?instruments)
WHERE {
  ?instr skos:prefLabel ?instrument .
  ?instr skos:topConceptOf | skos:inScheme ?res .
  VALUES (?res) {
    (<http://data.doremus.org/vocabulary/iaml/mop/>)
  }
}
GROUP BY ?instr
```

This results are also saved in a file, named *instruments.json*, and are also searched sequentially as the two previous ones.

Custom Slots for dates

In the case of dates extraction the best extractor for this task is Duckling, as it is a rule based entity extractor so the confidence of it will always be either 1 or 0. When extracting dates, Duckling can return a string value of the date extracted, if the user is looking for information in a specific year or date, or a dictionary containing two dates, this if the user is looking for information in a range of dates.

For the custom slots for dates there are two cases, the first one being when the requested date is to search for something historical, being this works written in a specific year or finding an artist who wrote more compositions during some period, the second case is when the user is searching for a performance in the “next week” or “next month”.

Historical dates

When the user inputs a sentence requesting for some works written in a specific year Duckling returns a string containing the date the user inputted, for a period of time, or before or after a year the returning value is a dictionary containing the values “*to*” and “*from*”, in the case two years were introduced by the user both keys will contain dates, if the user introduced a request as after or before one of those keys will have a value set to “*None*”, knowing that this request goes in a specific direction from the given date.

Now, when receiving the extracted entities from Duckling it is necessary to verify if the dates belong to an interval or not, this verification is done by the *class ActionYearExtractionSlot()*, this class is declared in the actions file mentioned previously. If the value corresponds to a string a dictionary is created with the key "*not interval*", and the value set to the given date. If the value returned by Duckling is a dictionary, a dictionary in python is going to be created with keys named "*to*" and "*from*", the keys returned from Duckling are verified to see if one of those contains a *None* value and the other one is set to the year requested, in the case neither of those contains a *None* value, the minimum value is set to *from* and the other one to *to*. Finally the dictionary created during this process is returned as the date-period Slot for the action implementation.

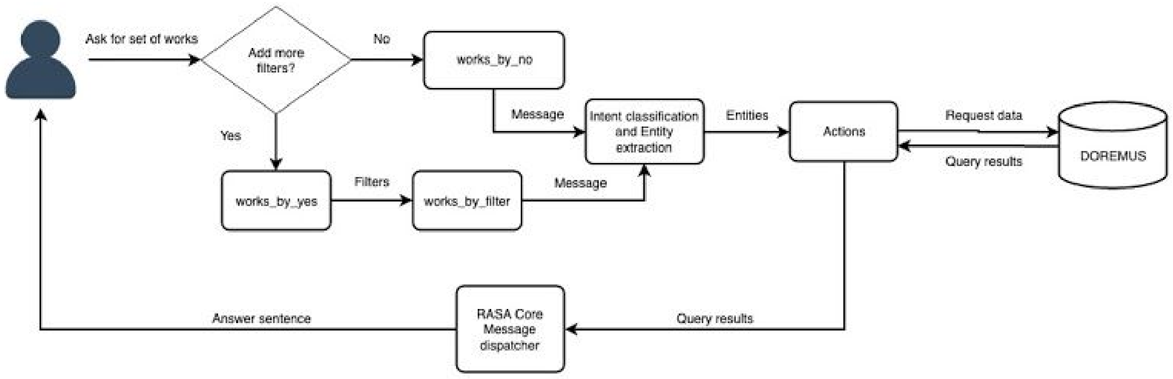
Performances dates

Here the extraction from Duckling works in a similar way, but now the input from the user could be a sentence as "*this week*" or "*next month*". In this case Duckling returns just one date, but the needed values for completing the query should be intervals, as the user requests is ranging over a week, month or year and there is only the initial day of this range. In order to be able to complete the query the Pendulum library is used to compute the final date of this requests. For this case the *class ActionTimeTest()* was created.

During the extraction, Duckling returns some additional information, this is the "*grain*" of the time entity, this time grain can specify if the given sentence is referring to minutes, hours, weeks, months or years, so this value is used to know when the ending date should be. As for the search of the performances the grain values smaller than weeks would not matter this are going to be ignored, so the computation of the ending values only take into account grain values for week, month and years. Because this only happens to a set of phrases as "*this week*" the ending date must be 1 week, month or year after the extracted date. After computing the end of the interval the values are saved in a dictionary with the keys *from* and *to*, then this dictionary is returned as value to the date-period Slot for further use.

Retrieving a set of works

As explained on the previous Stories section, in order to be able to retrieve a set of works there are 3 different stories to take into account before realizing the query to the DOREMUS endpoint. The first one represents the actual request for the set of works, the second one is if the user does not need to add any other fitler, and the third one is if the user wants to add more filters to the search. The workflow for retrieving a set of works is the following:



The user can ask for any number of works as long as the knowledge base contains this amount of works, if the user asks for a bigger set of works the returned quantity of works would be all the available works found.

There are several filters to take into account for this action:

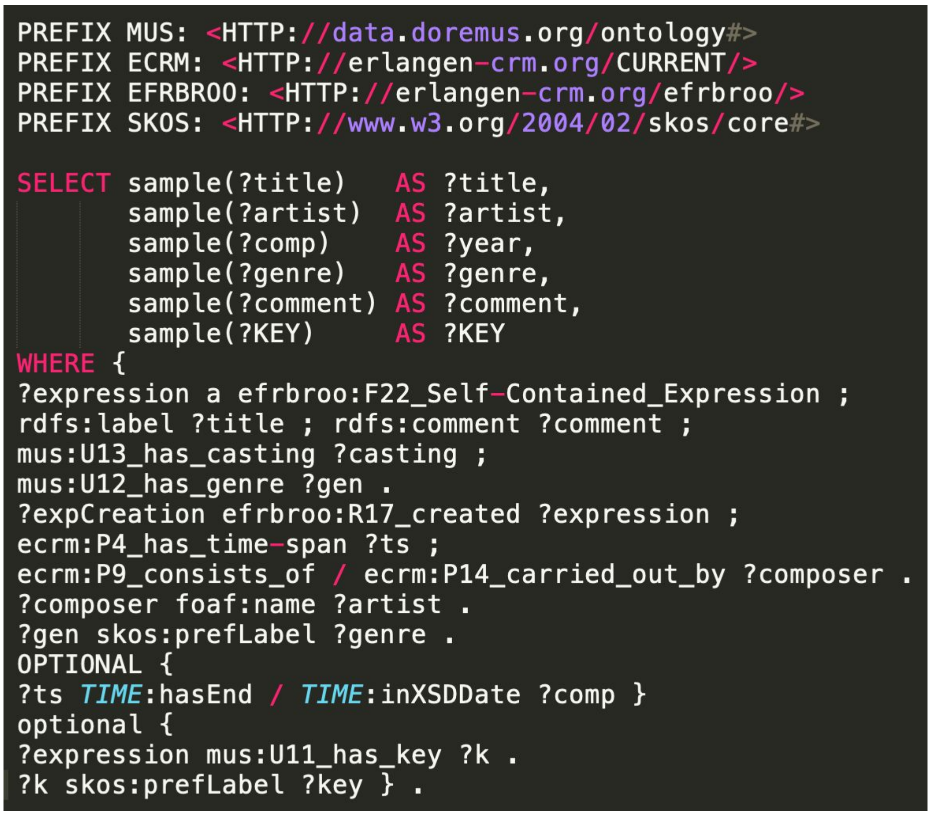
- **Artist:** this filter is just for the artist name, this can be either the full name or part of it.
- **Instruments:** the instruments can be inputted in different languages (the ones available in the knowledge graph)
- **Genres:** the music genre (as a symphony or concerto), this also can be inputted in the different languages found in DOREMUS.
- **Period:** this represents the date period when the compositions were written, it has 4 cases, it can be a specific year of composition, a range of dates, before a specific year or after a specific year.

These filters can be applied in different ways, the user can input the filters since the beginning of the request or after the bot asks if adding more filters is needed, in both cases the filters would be applied to the query. An example of how a conversation of this case could go is:

User: *Tell me 2 works by Vivaldi*
Bot: *Sure! 2 of Vivaldi Do you want to add some filters? Like the instruments, genre or composition period.*
User: *Yes*
Bot: *Ok, tell me what*
User: *With piano*

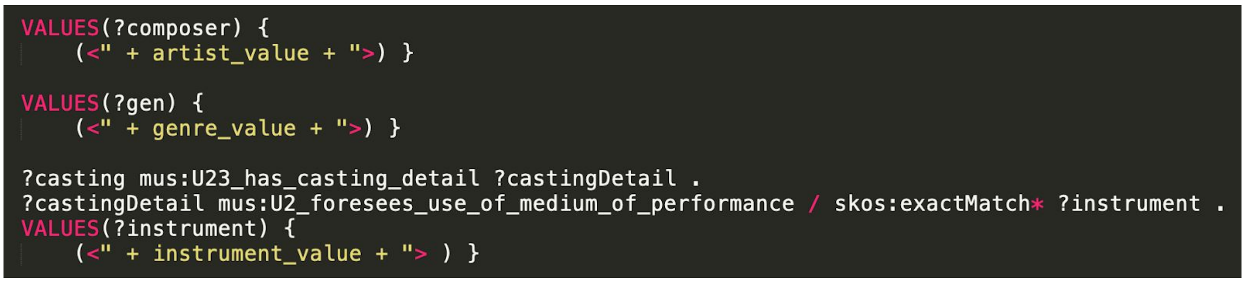
This is just one path that could be taken by the user, the user could also answer *no* to adding more filters and the request would go for just 2 works composed by Vivaldi, also a date period as *between 1720 and 1740* could be applied in the input sentence and this would return only works written in that specific period of time by Vivaldi.

The static section of the query used in the works by action is the following:



```
PREFIX MUS: <HTTP://data.doremus.org/ontology#>
PREFIX ECRM: <HTTP://erlangen-crm.org/CURRENT/>
PREFIX EFRBROO: <HTTP://erlangen-crm.org/efrbroo/>
PREFIX SKOS: <HTTP://www.w3.org/2004/02/skos/core#>

SELECT sample(?title)   AS ?title,
       sample(?artist)  AS ?artist,
       sample(?comp)    AS ?year,
       sample(?genre)   AS ?genre,
       sample(?comment) AS ?comment,
       sample(?KEY)     AS ?KEY
WHERE {
?expression a efrbroo:F22_Self-Contained_Expression ;
rdfs:label ?title ; rdfs:comment ?comment ;
mus:U13_has_casting ?casting ;
mus:U12_has_genre ?gen .
?expCreation efrbroo:R17_created ?expression ;
ecrm:P4_has_time-span ?ts ;
ecrm:P9_consists_of / ecrm:P14_carried_out_by ?composer .
?composer foaf:name ?artist .
?gen skos:prefLabel ?genre .
OPTIONAL {
?ts TIME:hasEnd / TIME:inXSDDate ?comp }
optional {
?expression mus:U11_has_key ?k .
?k skos:prefLabel ?key } .
```

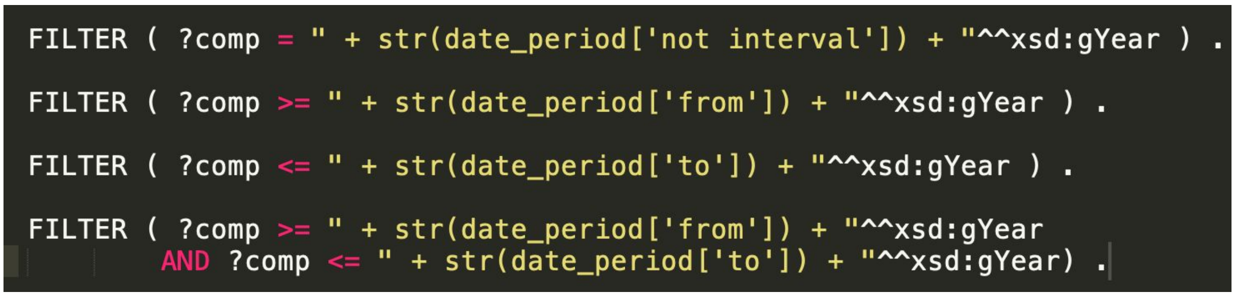
This next part is for a dynamic section, depending if the user introduced this values of the artist, genre and/or instruments:



```
VALUES(?composer) {
    (<" + artist_value + ">) }

VALUES(?gen) {
    (<" + genre_value + ">) }

?casting mus:U23_has_casting_detail ?castingDetail .
?castingDetail mus:U2_foresees_use_of_medium_of_performance / skos:exactMatch* ?instrument .
VALUES(?instrument) {
    (<" + instrument_value + "> ) }
```

The values used in this section of the query are the ones retrieved from the respective json files for each of the entities (artists.json, genres.json and instruments.json).

For the date period section (also part of the dynamic section of the query), there are the 4 cases previously mentioned:

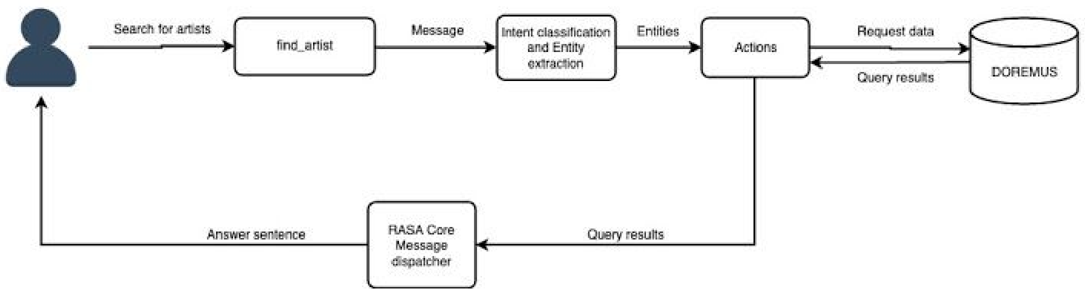


```
FILTER ( ?comp = " + str(date_period['not interval']) + "^^xsd:gYear ) .

FILTER ( ?comp >= " + str(date_period['from']) + "^^xsd:gYear ) .

FILTER ( ?comp <= " + str(date_period['to']) + "^^xsd:gYear ) .

FILTER ( ?comp >= " + str(date_period['from']) + "^^xsd:gYear
        AND ?comp <= " + str(date_period['to']) + "^^xsd:gYear) .
```

For all of these filters, the date values are extracted from a date_period python dictionary saved in the slot date-period. Previous to executing this action, the date-period entity goes through the action *action_year_slot*, this is just to obtain the years as they would be expected in the query, without any other information.

Finding artists

Another action in the chatbot is searching for an artist or a set of artists given some filters. The idea is to be able to find an artist given some information about its compositions and/or its birth place and/or date. The retrieved list of artists is sorted in a descending order, using the number of total works composed by the artists with the specified filters. The flow of this action is the following:



This action is simpler than the previous one in respect to how the conversation should go. The user will simply input a sentence with all the requested constraints that wants for the search. The entities received in this action are the genre of the compositions, the instruments used to play these works, the date period of the birth date of the artists and the city in where they were born.

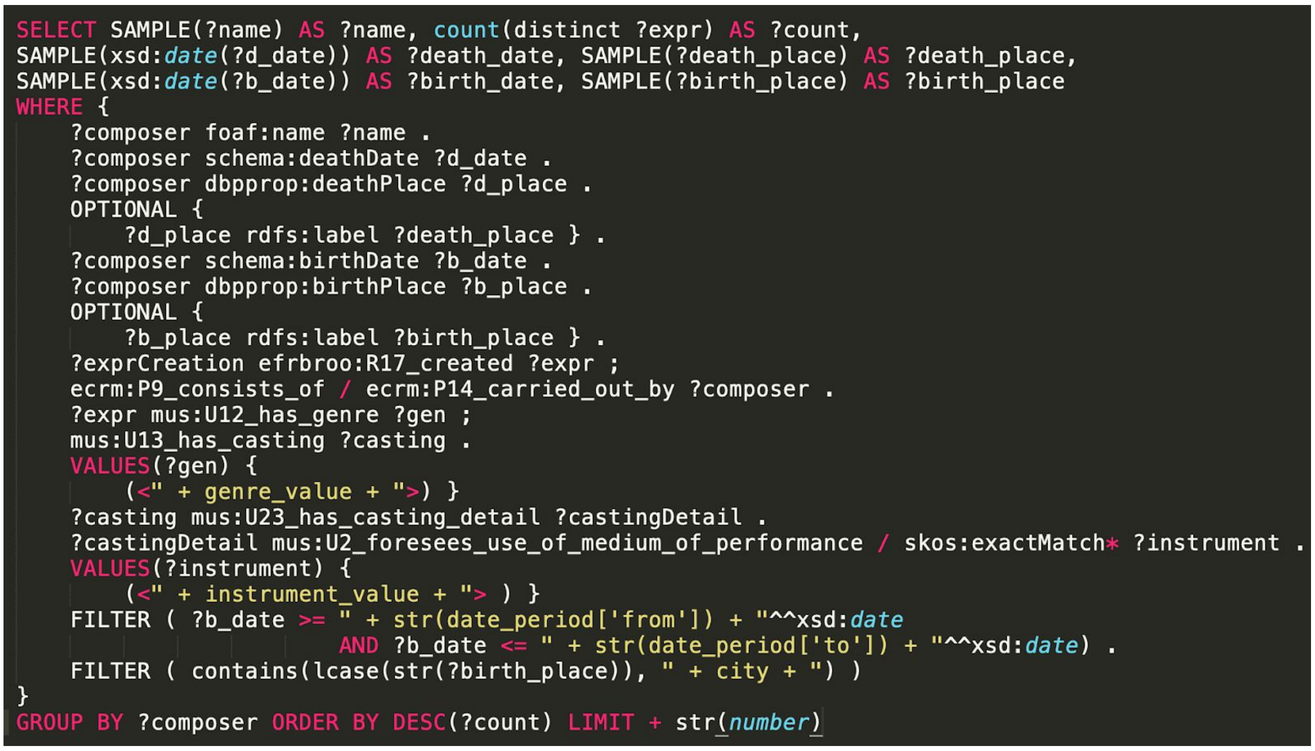
An example of a request for this action would be:

User: *Find me 2 artists born in Vienna between 1700 and 1800*

or it could also be something like:

User: *Give me 2 artists who wrote more works for piano*

The query used for this action is:

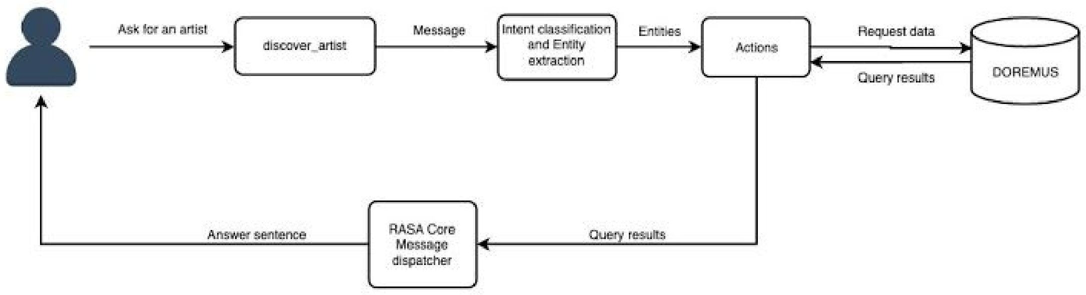


```
SELECT SAMPLE(?name) AS ?name, count(distinct ?expr) AS ?count,
SAMPLE(xsd:date(?d_date)) AS ?death_date, SAMPLE(?death_place) AS ?death_place,
SAMPLE(xsd:date(?b_date)) AS ?birth_date, SAMPLE(?birth_place) AS ?birth_place
WHERE {
    ?composer foaf:name ?name .
    ?composer schema:deathDate ?d_date .
    ?composer dbpprop:deathPlace ?d_place .
    OPTIONAL {
        ?d_place rdfs:label ?death_place } .
    ?composer schema:birthDate ?b_date .
    ?composer dbpprop:birthPlace ?b_place .
    OPTIONAL {
        ?b_place rdfs:label ?birth_place } .
    ?exprCreation efrbroo:R17_created ?expr ;
    ecrm:P9_consists_of / ecrm:P14_carried_out_by ?composer .
    ?expr mus:U12_has_genre ?gen ;
    mus:U13_has_casting ?casting .
    VALUES(?gen) {
        (<" + genre_value + ">) }
    ?casting mus:U23_has_casting_detail ?castingDetail .
    ?castingDetail mus:U2_foresees_use_of_medium_of_performance / skos:exactMatch* ?instrument .
    VALUES(?instrument) {
        (<" + instrument_value + "> ) }
    FILTER ( ?b_date >= " + str(date_period['from']) + "^^xsd:date
                       AND ?b_date <= " + str(date_period['to']) + "^^xsd:date) .
    FILTER ( contains(lcase(str(?birth_place)), " + city + ") )
}
GROUP BY ?composer ORDER BY DESC(?count) LIMIT + str(number)
```

In this query we have the same dynamic cases for the filters, genre, instrument and the birth place would be included in the query just if the user specified it. And the dates have the same 4 cases as the works by action, in the image just one of those cases is included for simplification. For finding an artist the date-period entity also goes through the action *action_year_slot* to only extract the years wanted for the request.

Discovering artists

This action is the simplest of all the actions as the user only introduces as filter the artists that he wants information about, but also this is the action that provides most information of them all, as it provides a small biography, and all the information related to the born and death places and dates. The flow of the action goes as:



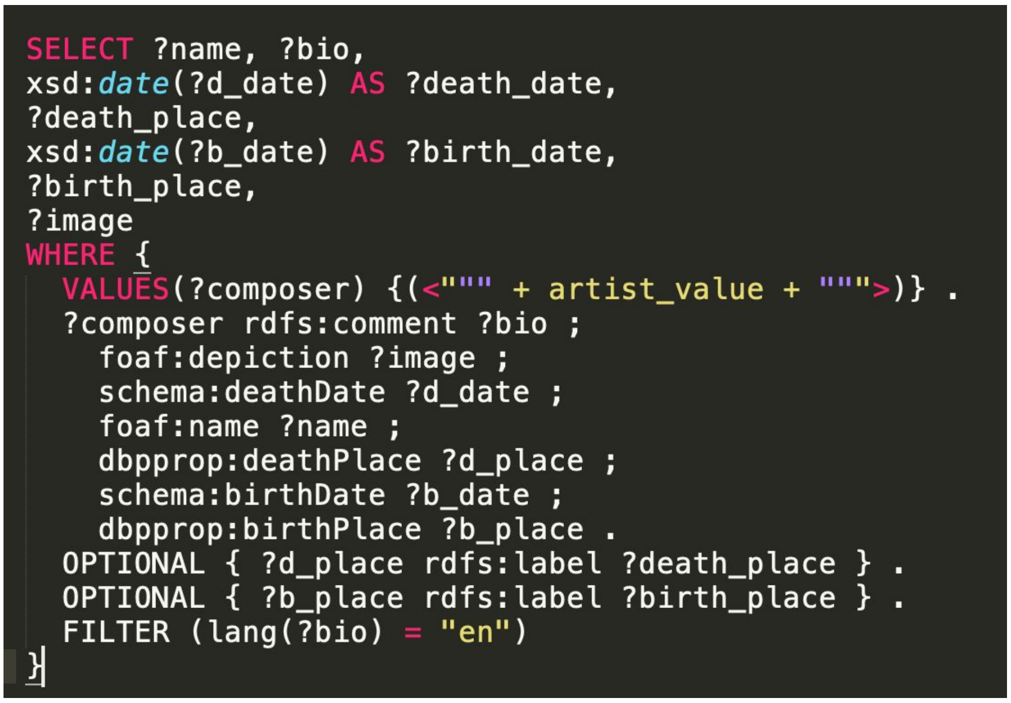
An example of how the user can request information for an artist would be:

User: *Tell me about Mozart*

there are different ways of asking for this information, another example would be:

User: *Who is Liszt?*

The query to request for an artist is presented in this image:

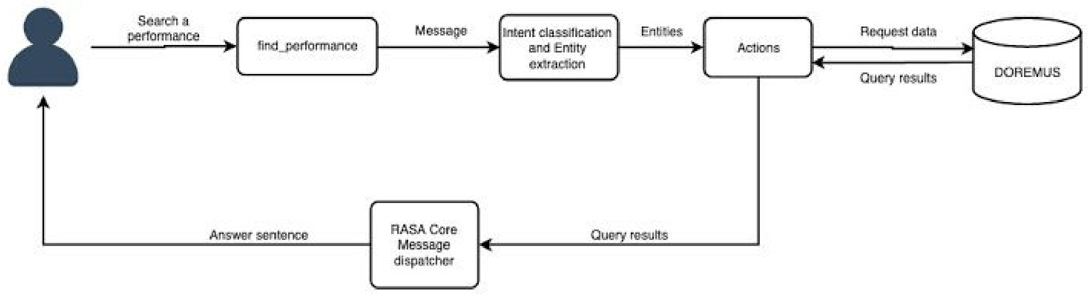


```
SELECT ?name, ?bio,
xsd:date(?d_date) AS ?death_date,
?death_place,
xsd:date(?b_date) AS ?birth_date,
?birth_place,
?image
WHERE {
  VALUES(?composer) {(<""" + artist_value + """>)} .
  ?composer rdfs:comment ?bio ;
    foaf:depiction ?image ;
    schema:deathDate ?d_date ;
    foaf:name ?name ;
    dbpprop:deathPlace ?d_place ;
    schema:birthDate ?b_date ;
    dbpprop:birthPlace ?b_place .
  OPTIONAL { ?d_place rdfs:label ?death_place } .
  OPTIONAL { ?b_place rdfs:label ?birth_place } .
  FILTER (lang(?bio) = "en")
}
```

Finding performances

Finding musical performances can support two filters, one for the city in which the performance is going to take place, and one for the date when it is happening. In the case of the city, it can process any request for any city, but as right now the DOREMUS knowledge graph only contains performances occurring in Paris, so any other request for other cities will not return performances, but when available there it should be able to retrieving them. For the date here we have a different situation as the previous action because in the past ones the date needed to find an artist or a set of works had to be more specific and the user had to provide the year of that period he is searching for, but now we have that the user can express the period of time more implicitly compared to the historical dates, so after retrieving the extracted time entity this goes through *action_time_test*.

The flow of this action is the following:



An example of a sentence for this action would be:

User: *Tell me 2 events in paris this week*

in this sentence the user can provide a range of periods of time when the performance could be.

The query to retrieve the performances is:

```
SELECT SAMPLE(?title) AS ?title, SAMPLE(?subtitle) AS ?subtitle,
SAMPLE(?actorsName) AS ?actorsName, SAMPLE(?placeName) AS ?placeName,
SAMPLE(?date) AS ?date
WHERE {
?performance a mus:M26_Foreseen_Performance ;
  ecrm:P102_has_title ?title ;
  ecrm:P69_has_association_with / mus:U6_foresees_actor ?actors ;
  mus:U67_has_subtitle ?subtitle ;
  mus:U7_foresees_place_at / ecrm:P89_falls_within* ?place ;
  mus:U8_foresees_time_span ?ts .
?place rdfs:label ?placeName .
?actors rdfs:label ?actorsName .
?ts time:hasBeginning / time:inXSDDate ?time ;
   rdfs:label ?date .
FILTER ( ?time >= '""" + start_day + """'^^xsd:date
AND ?time <= '""" + end_day + """'^^xsd:date) .
FILTER ( contains(lcase(str(?placeName)), \"""" + city + """\") )
}
GROUP BY ?performance
ORDER BY rand()
LIMIT """ + str(number))
```

Issues during the implementation

Artists, genres and instruments names

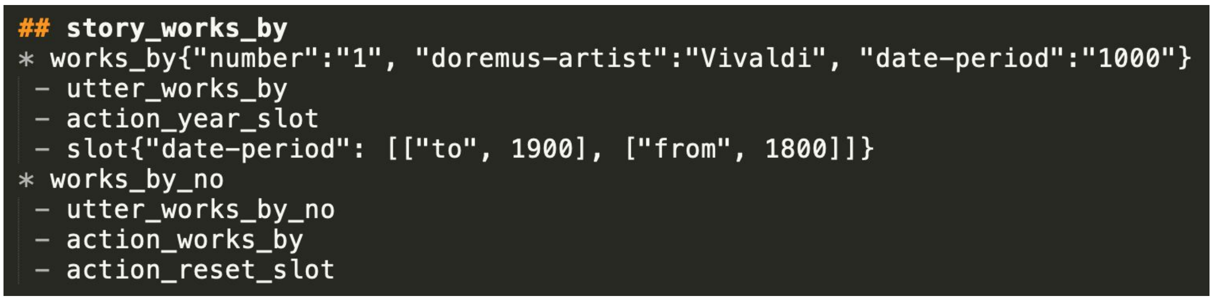
One of the issues presented when implementing the chatbot was how to manage the different ways in which a user can write the name of an artist, a genre or an instrument. When trying to use the values given by an user to complete the query, as literal as it was written, the assistant could retrieve sometimes some of the requests, but it could not retrieve everything, as inside the knowledge base the names are also written differently.

A possible solution was to create a dictionary of synonyms, which represented the multiple ways this names could be written. For example, this implied collecting all the names of the artists, and generating multiple permutations of their names, all of these being related to a specific name declared as a synonym inside RASA. But this solution would take a lot of time to generate a file containing all the permutations for the each of the names of artists, genres and instruments.

Because of this problem is that the solution used for the actual implementation was to use the values assigned to each one of these entities, as explained in the actions section. Also, in order to be able to manage misspellings or different ways of writing names 3 extra functions were created. These functions (declared in the actions file) are *similarArtist*, *similarGenre* and *similarInstrument*, the functions work in the same way, they receive the name that was not found in the respective json file and compute the similarity ratio of the string using the *SequenceMatcher* from the python library *difflib*. The functions will get the closest match from the concatenated names of each one of them and use the sequence matcher to evaluate how similar are the strings, if a ratio of 0.8 or more is achieved, that would be the selected name and value to use for the query, if not it will return the name with the highest ratio.

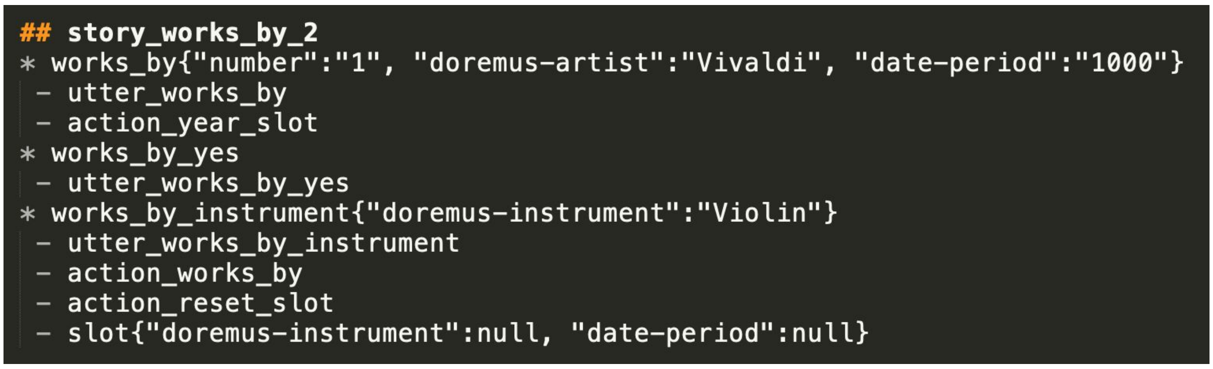
Story collision

Another issue that appeared during the development was having story collisions. One of the story collisions presented was with the works by flow, when doing the request the chatbot would follow either the *no* path or the *yes* path randomly. This issue was due to the way the stories were declared, in the first implementation of this stories these were declared as illustrated in the following images:

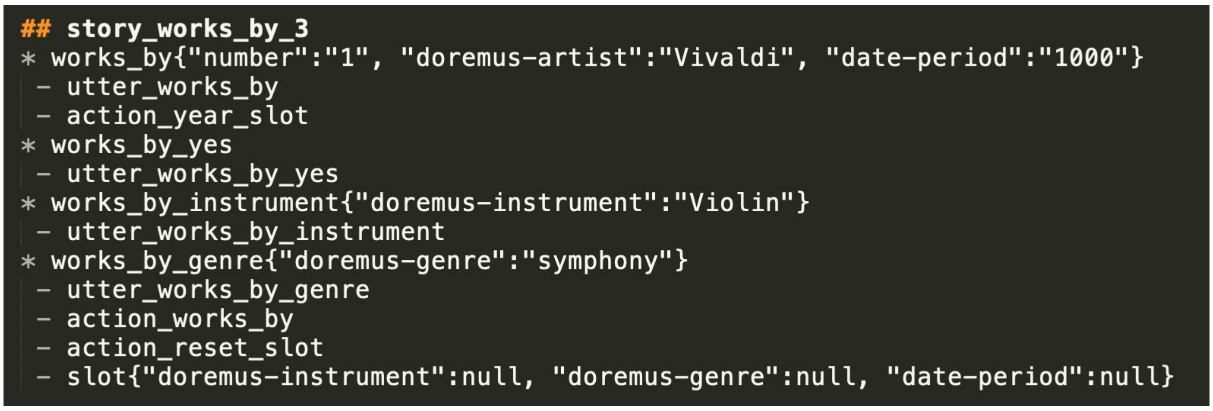


```
## story_works_by
* works_by{"number":"1", "doremus-artist":"Vivaldi", "date-period":"1000"}
 - utter_works_by
 - action_year_slot
 - slot{"date-period": [["to", 1900], ["from", 1800]]}
* works_by_no
 - utter_works_by_no
 - action_works_by
 - action_reset_slot
```

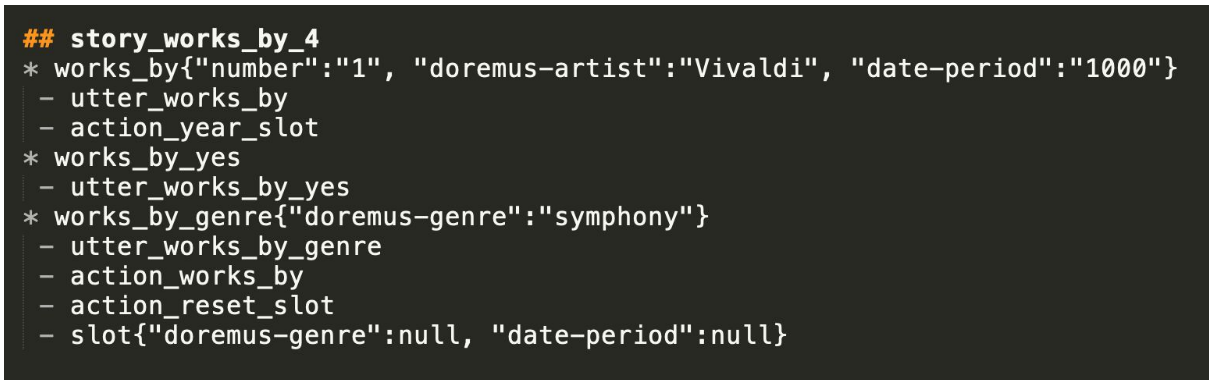
In the first image, there is represented the path for when the user does not want to add any more filters. For the following images, all are paths were the user would add more filters, but each filter had an intent:



```
## story_works_by_2
* works_by{"number":"1", "doremus-artist":"Vivaldi", "date-period":"1000"}
 - utter_works_by
 - action_year_slot
* works_by_yes
 - utter_works_by_yes
* works_by_instrument{"doremus-instrument":"Violin"}
 - utter_works_by_instrument
 - action_works_by
 - action_reset_slot
 - slot{"doremus-instrument":null, "date-period":null}
```



```
## story_works_by_3
* works_by{"number":"1", "doremus-artist":"Vivaldi", "date-period":"1000"}
 - utter_works_by
 - action_year_slot
* works_by_yes
 - utter_works_by_yes
* works_by_instrument{"doremus-instrument":"Violin"}
 - utter_works_by_instrument
* works_by_genre{"doremus-genre":"symphony"}
 - utter_works_by_genre
 - action_works_by
 - action_reset_slot
 - slot{"doremus-instrument":null, "doremus-genre":null, "date-period":null}
```



```
## story_works_by_4
* works_by{"number":"1", "doremus-artist":"Vivaldi", "date-period":"1000"}
 - utter_works_by
 - action_year_slot
* works_by_yes
 - utter_works_by_yes
* works_by_genre{"doremus-genre":"symphony"}
 - utter_works_by_genre
 - action_works_by
 - action_reset_slot
 - slot{"doremus-genre":null, "date-period":null}
```

The problem with having this approach to the works by action was that there were too many intents to take into account, as it is possible to see, there was an intent for adding genre filters, another one for instruments, and the intents for identifying the requests, so there were too many ways for a conversation to go on, having a lot of possibilities and generalizing the conversation for all these intents was complicated.

The solution for this problem was to create an intent that represents when the user wants the addition of more filters, in general, not an intent per entity, as it was defined before. So, the story now is more general and there are less intents to take into account, and the conversation is better represented.

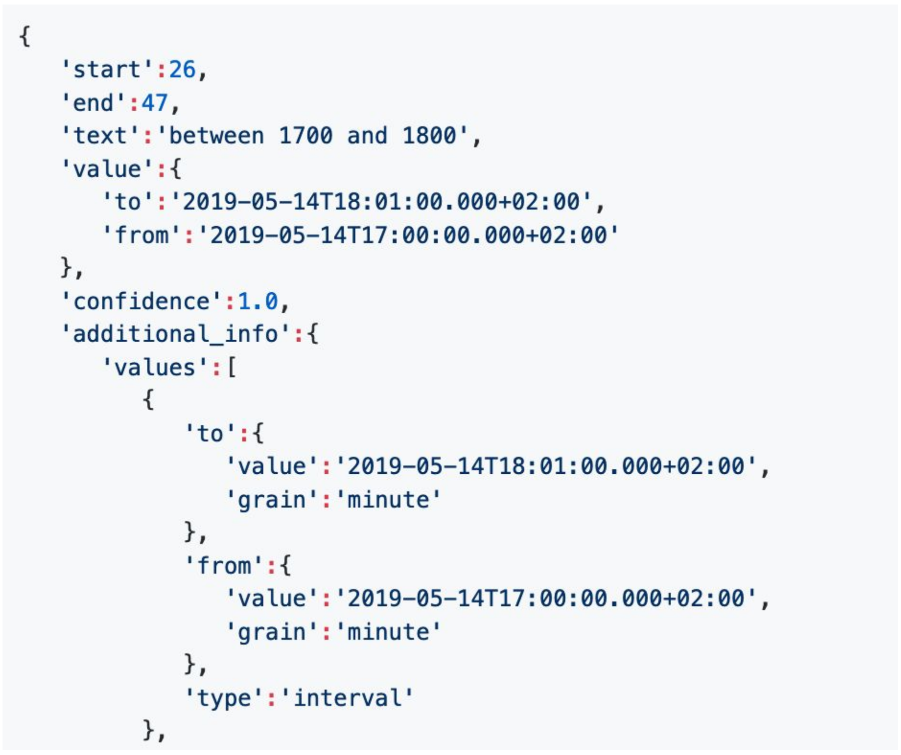
Time extraction

Time extraction is a complicated topic, it may be easier to do this for historical dates, as the expected values is usually a specific year or date, but when receiving a phrase referring to a period of time instead of a specific value it gets more difficult. This is why the selected library for time extraction was Duckling, it uses rule based extraction, so usually the extraction is accurate, getting very good results when used for dates, also this was the most recommended library for this task during the research on how to process this values.

For time entity extraction there were some issues, to be more specific there were two type of issues:

Historical dates

The first problem was during the extraction of entities from historical dates, for most values Duckling works fine, and when using the extractor in their website https://duckling.wit.ai/ it always returns the expected values, but while using the duckling container provided by RASA (the image of this docker container is rasa/duckling) there was a problem. For years having the format XX00 to XX50, as 1700 and 1750, the extracted value returned by the container had the time grain of minute. The value returned by Duckling in this case was the current day with the year taken as a time of the day, so it was taking the format XX00 to XX50 as hours and minutes. Here is an example of this:



```
{
   'start':26,
   'end':47,
   'text':'between 1700 and 1800',
   'value':{
      'to':'2019-05-14T18:01:00.000+02:00',
      'from':'2019-05-14T17:00:00.000+02:00'
   },
   'confidence':1.0,
   'additional_info':{
      'values':[
         {
            'to':{
               'value':'2019-05-14T18:01:00.000+02:00',
               'grain':'minute'
            },
            'from':{
               'value':'2019-05-14T17:00:00.000+02:00',
               'grain':'minute'
            },
            'type':'interval'
         },
```

The request showed in the image is for the sentence *between 1700 and 1800*. The returned value was a range of hours from the current day the request was done. This issue was posted in Ducklings github site, in the issues section, the link to this post is https://github.com/facebook/duckling/issues/371 but no answer has been received from them.

In order to retrieve the correct values a work around had to be done, but in this extraction there is still some valuable information, this is if the sentence represents a interval or not, so the extraction is used to find out if the user is referring to which one. After retrieving to what the user is referring to with this period of time, the years are extracted from the sentence manually, verifying to what part of this extraction corresponds to digits.

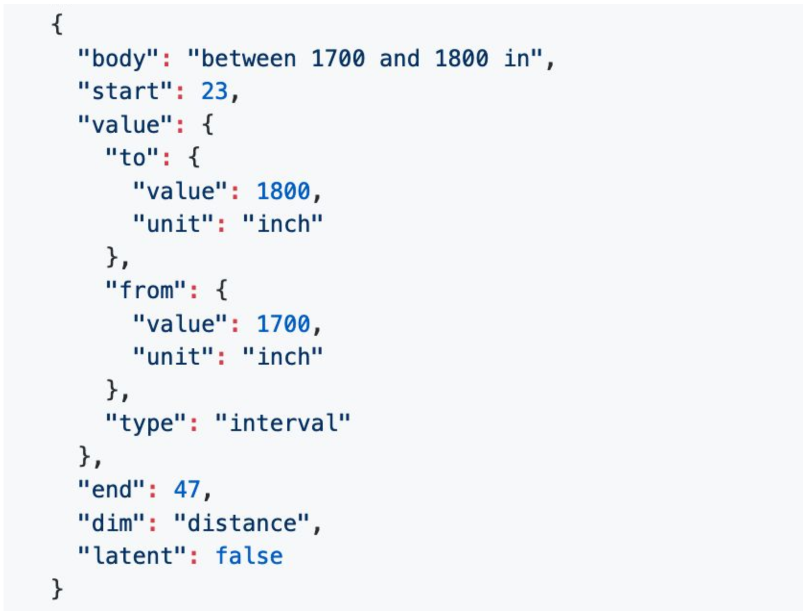
Another problem related to this type of dates was presented when changing the structure of the sentence. Here is an example of this change:

Find me 2 artists born in Vienna between 1700 and 1800

when changing the previous sentence to this structure:

Find me 2 artists born between 1700 and 1800 in Vienna

In this case, again, when using the extractor found in their website, the extraction went without any problems, but when using the Duckling docker container the value returned had a different dimension, in this case Duckling recognized this as a distance rather than a period of time. Here is an example of the returned value:



```
{
  "body": "between 1700 and 1800 in",
  "start": 23,
  "value": {
    "to": {
      "value": 1800,
      "unit": "inch"
    },
    "from": {
      "value": 1700,
      "unit": "inch"
    },
    "type": "interval"
  },
  "end": 47,
  "dim": "distance",
  "latent": false
}
```

From this extraction it is possible to see that Duckling is taking the preposition *in* as if it was referring to inches. In order to be able to extract dates from this case, the entity extractor used as an alternative is spaCy, as it manages to identify this value as a date. This issue was posted initially in the RASA github, thinking it was an issue when going through the NLU pipeline, this issue is posted in https://github.com/RasaHQ/rasa/issues/3643 but this was not an issue of RASA, so another post was created in Ducklings github site, this is the link to it https://github.com/facebook/duckling/issues/378 .

Current dates

This issue with the dates is presented also during the usage of the Duckling extractor, but this is a simpler case than with historical dates. The problem was present when extracting time from some phrases as *this week*, *this month* or *this year*. In here the problem is that when receiving this kind of sentences when searching for performances means the user expects to obtain results in the range of either a week, a month or a year, but for this 3 particular sentences the returned value was one value, this is the starting day of the period requested, so when trying to do the request the resulting value would be since the starting day of this period with no ending date, so the user could get performances outside it.

To be able to close the period requested by the user the Pendulum library is used to compute the ending dates of these periods, given that Duckling provides the time grain and that Pendulum provides tools to easily compute ranges of times each of this expected ending dates is computed inside the class action *ActionTimeTest*. With respect to other requests as *next week*, *next two weeks* or *next year* there were no issues as the values always returned the expected start and ending days.

Issues not solved

Almost all the issues presented during the development of the chatbot were managed, the remaining issues are the ones related to the date extraction when using Duckling, even though they are partially covered with some manual fixes that were needed in order to be able to do query requests including the date constraints.

The links to the issues posts for the problems presented with the usage of Duckling are the following:

- Problem with date extraction: https://github.com/facebook/duckling/issues/371
- Incorrect extraction when changing sentence order: https://github.com/facebook/duckling/issues/378

How to run it

There is a make file declared in the project and it contains the following make actions:

- To train the Rasa NLU model by running *make train-nlu*. This will train the Rasa NLU model and store it inside the */models/current/nlu* folder of your project directory.

- Train the Rasa Core model by running *make train-core*. This will train the Rasa Core model and store it inside the */models/current/dialogue* folder of your project directory.

- In a new terminal start the server for the custom action by running *make action-server*. This will start the server for emulating the custom action.

- To test the assistant by running *make cmdline*. This will load the assistant in your terminal for you to chat.

- To start the chatbot as a server *make chatbot-server*. This will load the assistant to use the port 5005 in order to integrate it with a messaging app, as for now I just have done this with slack, but it should work with others (credentials for the messaging app should be added in the **credentials.yml** file).

- To be able to use the chatbot it is necessary to have running the action server in order to be able to do the requests to the DOREMUS SPARQL endpoint and have running the docker container for Duckling (for time extraction) using *docker run -p 8000:8000 rasa/duckling*.

To install all the libraries needed to run the chatbot it is necessary to use the command *pip install -r requirements.txt*, this file contains a list and versions of the libraries used to develop the chatbot.

Evaluation

What works

The development of this chatbot has been successful as it is able to recognize different entities needed to resolve the multiple actions that are needed in order to answer users requests, also the chatbot has showed being able to classify all the intents presented previously, the only limitations being the available training data as describing all the ways a user can ask is a complicated task, but this could be solve with the addition of more training data or using the interactive learning mode (explained in the Future work section).

The bot is able to manage the addition of multiple filters by the users in the different actions were this information is relevant for retrieval of information.

Here is a list of samples by intent of example requests that were tested during the implementation of the chatbot and that were successfully answered:

- **works_by**

  Give me 2 works by Bach
    - *Added filters: with violin and sonata genre*

  Give me 2 works with piano
  Can you tell me 2 works by Mozart?
    - *Added filters: with violin after 1780*

- **find_performances**

  Are there any events in Paris this summer?
  Find me 2 events in Paris this week
  Find me 2 events in Paris next week
  Propose me 3 events in Paris from July to September
  Give me 2 events in Paris in the next 2 months

- **find_artist**

  Find me 2 artists born in Vienna between 1700 and 1800
  Give me 2 artists born in vienna who wrote more works for piano
  Do you know 3 artists born in Venice who wrote works for sonata?
  Tell me 2 artists born between 1700 and 1800 in Paris

- **discover_artist**

  Tell me about Mozart
  Talk me about Liszt
  Who is Haydn?
  What do you know about Vivaldi?
  Do you know something about Ludwig van Beethoven?

For all these intents the bot is able to classify them and the extraction of all the defined entities has not shown any other issue different to the previously mentioned in the Issues section. There is some room for improvement in the dialogue management of the chatbot, but it is able to answer to different requests from the user. Also it can manage some misspellings from the user and is able to resolve for the action requested.

Future work

There are several things that could be improved from this implementation. One important improvement would be the addition of support for additional languages, the only constraint with increasing languages support would be the limitations of some of the libraries used for this project, one of this would be the Duckling library, that contains support for several languages but not all languages are supported for all the dimensions available there.

Another improvement would be the training data. Increasing the number of examples for intent classification would improve the accuracy and the range of natural language understanding this chatbot when having a conversation. One way to improve the chatbot would be the usage of interactive learning. Interactive learning is a training mode provided by RASA Stack were feedback is provided during the conversation with the chatbot, so by using this RASA feature it is possible to teach the bot to do some actions. By doing interactive learning the bot could develop better conversation flows and could learn to follow more complex conversations and do it more fluid.

Also integration with other messaging platforms as Facebook Messenger, Telegram or some voice assistants could be a good addition to the current build of this chatbot as for now it is only integrated with Slack.

Keeping track of the solutions provided to the issues presented with the usage of Ducklings extractor would be important in order to have a better and more accurate extraction of the time entities, as for now the solutions developed work, but having the correct extraction from Duckling would allow for a better dialogue management.

The addition of more intents and actions to the chatbot could also enrich the user experience when using this assistant for retrieval of classical music information. Some analysis on which actions does the user needs that are not already covered in this implementation needs to be done, as an example of this possible actions could be the implementation of retrieving information about a specific composition.

Conclusions

During the implementation of this project we learned and understood RASA Stack, the way its components, NLU and Core, interact in order to understand a conversation with an user and how the dialogue is managed. For the NLU we saw how the intents are defined is important, as they have to generalize enough what the user intention is with his sentence, and also how there are different libraries that perform better for the extraction of some type of entities. For the Core we understood how the stories interact during the training process and the importance of having well structured general stories in order to avoid collisions between stories that could overlap due to the way the user can elaborate the sentences.

In the implementation there were some issues, mainly in entity extraction, this because the input data received by the bot could be very different and the retrieval of the queries with this input data did not give the expected responses. There were also issues related with one of the libraries used in the development, Duckling, this issues were addressed manually inside the build of the bot and a post informing of this problems was done in the respective website.

Overall the development of the project was successful as the chatbot it is able to answer to the different request the users could ask, this does not mean this are all the possible requests, but some of the most important ones, and now an user can obtain the information more easily through the use of this chatbot.

The content of this project is on Github and can be found in the following link: https://github.com/danielhomsa/Music-chatbot-DOREMUS-RASA

References

- Legacy-docs.rasa.com. (n.d.). *Step 1: Understand the Rasa Stack*. [online] Available at: http://legacy-docs.rasa.com/docs/get_started_step1/
- Rasa Blog. (2019). *Rasa NLU in Depth: Intent Classification*. [online] Available at: https://blog.rasa.com/rasa-nlu-in-depth-part-1-intent-classification/
- Rasa Blog. (2019). *Rasa NLU in Depth: Entity Recognition*. [online] Available at: https://blog.rasa.com/rasa-nlu-in-depth-part-2-entity-recognition/
- Medium. (2018). *Contextual Conversational Engine— The Rasa Core Approach — Part 1*. [online] Available at: https://medium.com/strai/contextual-conversational-engine-the-rasa-core-approach-part-1-1acf95b3d237
- Rasa.com. (2019). *Rasa Docs*. [online] Available at: http://rasa.com/docs/
- GitHub. (2019). *facebook/duckling*. [online] Available at: https://github.com/facebook/duckling
- Towards Data Science. (2018). *A Chatbot from Future: Building an end-to-end Conversational Assistant with Rasa*. [online] Available at: https://towardsdatascience.com/a-chatbot-from-future-building-an-end-to-end-conversational-assistant-with-rasa-ai-51a1c93dabf2
- Duckling.wit.ai. (n.d.). *Duckling*. [online] Available at: https://duckling.wit.ai/
- Medium. (2019). *1. Build a Conversational Chatbot with Rasa Stack and Python— Rasa NLU*. [online] Available at: https://medium.com/@itsromiljain/build-a-conversational-chatbot-with-rasa-stack-and-python-rasa-nlu-b79dfbe59491
- Spacy.io. (n.d.). [online] Available at: https://spacy.io/usage/spacy-101
- Pendulum. (n.d.). [online] Available at: https://pendulum.eustace.io/
- Towards Data Science. (2019). *Building a Conversational Chatbot for Slack using Rasa and Python -Part 2*. [online] Available at: https://towardsdatascience.com/building-a-conversational-chatbot-for-slack-using-rasa-and-python-part-2-ce7233f2e9e7