Developing a virtual assistant for answering music related questions

Claudio SCALZO
(scalzo@eurecom.fr)

Luca LOMBARDO
(lombardo@eurecom.fr)

Professor: Raphaël Troncy
Supervisors: Pasquale Lisena — Thibault Ehrhart

Contents

Chapter 1

Introduction

1.1 Why a virtual assistant?

In these years we're assisting to a huge development of virtual assistants, mainly in two different shapes: vocal assistant and chat bots. Usually, each of the tasks that the user asks to the virtual assistant are tasks that can be done in other ways; we can think about a user asking for directions to *Siri* or a user who tells his *Google Assistant* to schedule an appointment in his calendar. Each of these are tasks that can be done without the help of an assistant! So, why they are becoming so popular? The reason is of course related to the ease of use. There are tasks which are, even if easy, boring or annoying to do, and we can take as example the scheduling of an appointment: we can unlock the phone, open an app, write the name of the appointment, select the start date and the end date, select the reminder scheduling, select the option to keep it private or make it public, and select other options depending on which calendar app we're using. Easy of course, but way more complicated than saying: *"Hey Google, set me an appointment with Luca for tomorrow afternoon at 5pm, and please remind me 2 hours before"*.

Having said that, the reason behind the development of our virtual assistant should be clear: simplify the user's life when he wants to access some informations about classical music.

1.2 Scope of the project

Our project puts down its root into *DOREMUS*[1], a classical music knowledge base, mainly composed by an ontology of 64 classes and 250 properties, and a set of 17 vocabularies. The informations contained inside *DOREMUS* can be accessed thanks to a SPARQL endpoint, that lets users write queries to retrieve the set of

data they want to achieve.

Our project starts from this difficulty: how many people know SPARQL? And inside the set of people who know it, how many problems can rise using the SPARQL language? Using the right properties, checking the exact domain and ranges for each property and other annoying problems just to obtain simple results. Our bot faces this issue, making simple to obtain the most common informations the users want.

1.3 Aim and expected results

The real aim of the virtual assistant so, is to make simpler the use of the *DOREMUS* knowledge base and the retrieval of its data. But which will be our, concrete, final results? First of all, the shape of the virtual assistant is originally a chat bot. The queries are textually written and the responses are visually seen on the display of each device running a client with the bot installed. This, of course, is not the only shape the bot can have. It can be a vocal assistant, and the query can be done by voice. In each of the cases, the bot will extract the informations from the *DOREMUS* knowledge base using SPARQL queries, answering to the user's requests and without letting them know a single detail about the SPARQL query that is using or other technical details. It will also help the user to make a more detailed and precise query (guiding him with some questions), and will correct the users sentences when he does some typos or when he's wrong in writing some precise words (like artist names, instruments or musical genres).

Chapter 2

Natural Language Understanding

2.1 What is NLU?

Natural language understanding *(NLU)* is an artificial intelligence technique that deals with the automatic treatment of informations provided in a given natural language. It puts its roots into the first works by Daniel Bobrow[2] in 1964 and the *ELIZA*[3] project in 1965, by Joseph Weizenbaum. The interest in *NLU* has become in the years bigger and bigger, and after some interesting works like the one of William Woods[4], who introduced the *ATN (Augmented Transition Network)* in the natural language processing field, the research focused (in the 70s and 80s) in using machine learning techniques to fulfill natural language processing tasks.

2.2 NLP, NLU and NLG

Natural Language Processing (NLP), *Natural Language Understanding (NLU)* and *Natural Languange Generation (NLG)* are similar terms that doesn't share the same meaning.

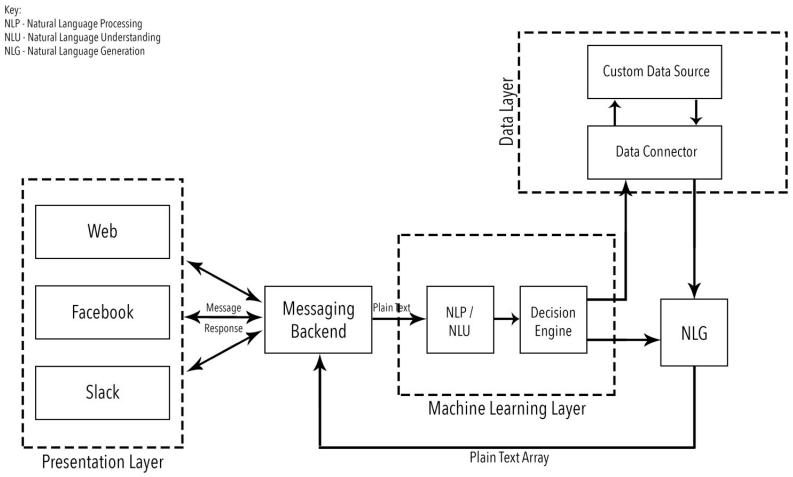

Figure 2.1: NLP, NLU, NLG and how Chatbots work.[5]

- *NLP* is the biggest set between the three, because it's a term which indicates the capability of a software to ingest an input sentence, split it in pieces (entities), understand the entities and their relationship, and give the answer to the user.
- *NLU* is a subset of *NLP*. It deals with understanding a natural language input (totally or partially unstructured) and convert it into a structured form that the machine can process and understand. Is a small yet critical task to achieve.
- *NLG* can be seen as the dual phase of *NLU*: turning structured data into natural language, to give the user the answer he wants.

2.3 How machine learning can help NLU?

One interesting work that can help in understanding the strong relationship between natural language understanding and machine learning, can be the one by Bing Liu and Ian Lane[6]. Intent detection and slot-filling are indeed crucial parts of NLU: intent detection can be seen as a classification problem were, given a set of words (the sentence), the algorithm is capable to classify that sentence and give it a label which will correspond to one of the intents that the machine is able to recognize. Slot-filling can be just seen as a sequence labeling task. For what regards the intent detection phase, the classification problem can be faced by popular machine learning algorithms like *Neural Networks (NNs)* or *Support Vector Machines (SVMs)*; instead, for the slot-filling sequence labeling task, useful tools

can be *Markov models (MEMMs)* and *Conditional Random Fields (CRFs).*

Is fundamental in this case, to understand the concept of alignment:

- Alignment is present when the output of the encoder is the same input given to the decoder. Is the case of slot-filling, where the entire set of entities is aligned (when processed with the encoder, can be directly given to the decoder).
- Alignment is not present when the output of the encoder can't be directly given in input to the decoder. An additional processing phase is needed before feeding the decoder with the informations it needs as input. is the case of the intent detection, where the decoder needs the information of all the words to detect the intent.

In both intent detection and slot-filling tasks, *RNNs (Recurrent Neural Networks)* can be (and have been) applied:

In the case of slot-filling, the input-output sequence is explicitly aligned, with a "slot" to fill for each single information extracted from the sentence. The models used in the years, in this case, have the aim to maximize the likelihood:

$$\arg\max_{\theta} \prod_{t=1}^{T} P(y_t | {y_1}^{t-1}, x; \theta) \tag{2.1}$$

Another structure exploiting RNNs can be instead the RNN Encoder-Decoder framework: in this case the input-output sequence is not aligned, and this structure is of course better for an intent detection task. In this case, the encoder and the decoder are separated entities: the encoder reads a sequence of inputs into a context vector c, which is used from the decoder as source of informations for fulfilling its task. The probability of the output sequence of the decoder is:

$$P(y) = \prod_{t=1}^{T} P(y_t | {y_1}^{t-1}, c) \tag{2.2}$$

The work conducted by Liu and Lane aims to propose two different alternatives to be able to work well in joint intent detection and slot-filling tasks.

2.3.1 Attention-Based RNN model for joint intent detection and slot-filling

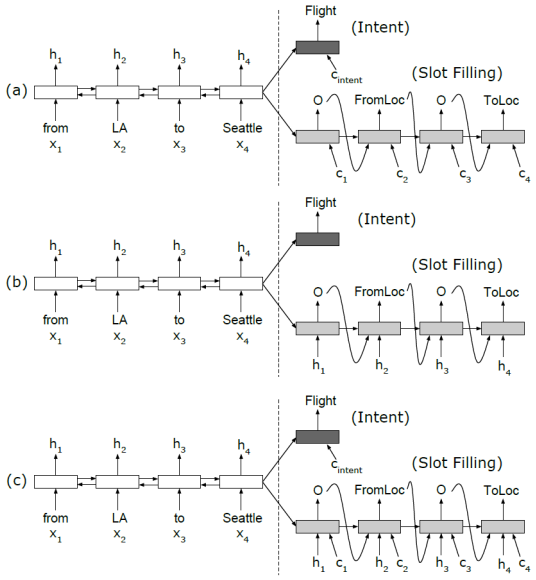

Figure 2.2: The three alternatives using the Encoder/Decoder model.

In this case, as the figure shows, the encoder and the decoder phase are separated. In the encoder phase we can find the *bidirectional RNNs*: each of them produces an hidden "forward" state fh_i and an hidden "backward" state bh_i. The output of the last *RNN* is used for the intent detection and to initialize the decoder *RNN* state.

In the subfigure *(a)*, we can find an example of non-aligned inputs. Indeed, the input given to the decoder phase is the context vector c, produced by a weighted sum of all components of the h vector produced by the encoder phase.
In the subfigure *(b)*, the inputs are aligned. The input vector of the decoder phase is just the h array.
The last subfigure *(c)* shows a situation with aligned inputs and also the addition of attention. This is useful for the joint intent classification and slot-filling task: both the h and the c vectors are used in the decoder phase, that shows also an intent detection taking as input the output of the last *RNN* and also the c context vector.

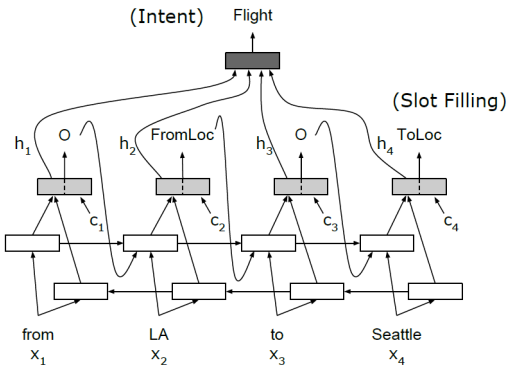

Figure 2.3: Attention-based RNN model for joint intent detection and slot filling.

The attention-based model proposed by Liu and Lane shows a clear structure: each bidirectional *RNN* produces a forward and a backward hidden state, useful for predicting the slot label (the core of the slot-filling task). At the same time, these outputs h_i are merged with the context vector c to predict the final intent label.

2.4 State of art: famous and most used tools

- **Wit.ai**[7]
  Facebook owned, is one of the most complete solutions for natural language processing. It supports 50 languages, it's free.

  Pros:

  - Supports machine learning to learn alternative phrases.
  - UI to work with intent and entities.
  - Developer view with conversation flows, context variables, branching logic.
  - Supports "roles" in entities (e.g. `fromLocation(Nice)` to `toLocation(Turin)`).

  Cons:

  - No "required field" option (slot-filling)

- **Snips**[8]
  An open source alternative to the most popular NLU systems.

  Pros:

  - Fully runs on device.
  - Heavily exploits machine learning (especially CRF).
  - Privacy by design.
  - Precise performance metrics .
  - Open source (*https://github.com/snipsco*).
  - Integrated Automatic Speech Recognition.

  Cons:

  - Doesn't support a big set of languages.

- **IBM Watson**[9]
  "The first choice as a bot-building platform for 61% of businesses".
  It supports Node, Java, Python, iOS and Unity SDK. Free and Premium plans. It's heavy but powerful.

- **Amazon Lex**[10]
  It powers *Alexa*. SDKs for popular platforms.
  You pay only for what you use. No demo, no open-source.

- **Microsoft LUIS**[11]
  Supports and heavily uses machine learning. Composite entities (e.g. *"2-adults first class flight ticket"*). Rich metrics of performances evaluation of the assistant.

- **Recast.ai**[12]
  SAP-owned. *"Free"* and *"Company"* accounts. Funny interactive demo on the website which shows SDK code for a lot of platforms.

Chapter 3

Dialogflow

3.1 What is Dialogflow?

Dialogflow is a powerful NLP developed by Google, born from the *API.ai* project, which aims to provide a rich conversational experience and a strong ease in using and developing the bot.

One of the strength points of *Dialogflow* is, first of all, its machine learning power: indeed, it's able to recognize complex entities inside sentences, and react also to big changes in the sentences with respect to the training ones. Another important point to notice, is the simplicity with which it can be trained: it provides a pleasant UI that lets adding intents, entities, training sentences, managing contexts and, in general, avoiding low-level details. Each of the terms we're used so far, are fundamental elements of the Dialogflow job: now we're going to see them in details.

3.2 How does it work?

First of all, the *Dialogflow* natural language processing analyzes the input that the user sends thanks to a client, which can be a web platform, a messaging application, or even a device with a microphone which takes as input the user's voice.

Dialogflow, except for base sentences, has to retrieve some data to fulfill the user's request: this can be done thanks to a data source. It can be, of course, any possible data store, like a database, a knowledge base, etc.

Dialogflow, retrieved the information, has to provide an answer: it can be done with a series of methods: textual, voice, graphics. It depends on the device that

the user is using to talk with the bot.

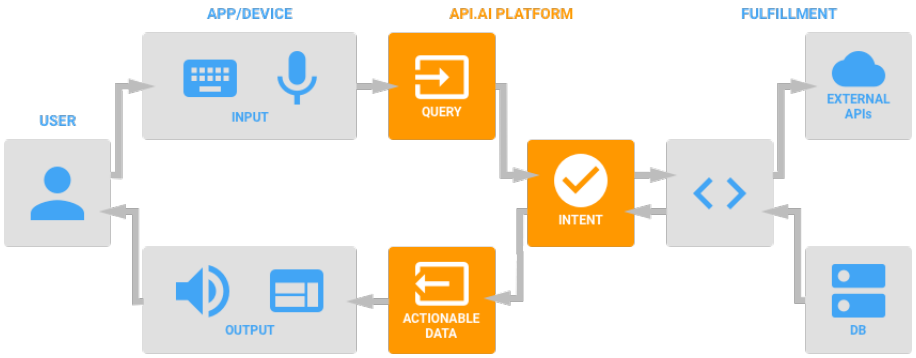

Figure 3.1: Architecture of a generic system using *Dialogflow* [13]

3.3 Fundamental concepts

Dialogflow works thanks to some fundamental pillars:

- **Intents**
  They are the mapping between the user query and the desired action. Each intent, indeed, corresponds to "what the user wants to do" in a certain fraction of the conversation.
- **Entities**
  They are fragments of natural language inputs, corresponding to a specific type. They can be cities, places, numbers, names, etc. There are a lot of default entities, but custom entities can be created by the user.
- **Actions**
  Actions represent what the bot needs to do after the intent has been resolved.
- **Parameters**
  They are all the values that are needed to perform an action.
- **Contexts**
  They are "containers" in which all the entities and parameters of the conversation are stored. They are fundamental to keep the informations through the conversational flow and make possible the "slot-filling".

One of the most powerful aspects of *Dialogflow* is the support for slot-filling. It is a technique that lets users to specify details (necessary or not) in different phases of the conversation: this means that the bot can go ahead and keep asking more details to the user before answering the query. Let's make an example:

Please give me 2 works - **User**
You told me few filters. Do you want to add something? - **Bot**
Yes! - **User**
Ok, tell me what - **Bot**
The artist - **User**
Ok! Just tell me his name - **Bot**
Mozart - **User**
Perfect! Do you want to add something else? - **Bot**
...

This result is obtained thanks to the "required" entity: in the *Dialogflow* UI (and of course, in the JSON file of the intent) it's possibile to specify some fields that are required for the query to work well. The bot will provide the final answer of the query if and only if all the required slot have been filled! This means that the conversation can go over and over until these informations are provided.

The power of *Dialogflow* so, is heavily based on slot-filling. It is able to keep the informations through all the flow of the conversation

Chapter 4

The bot

4.1 The architecture

The architecture of the bot is divided in four categories:

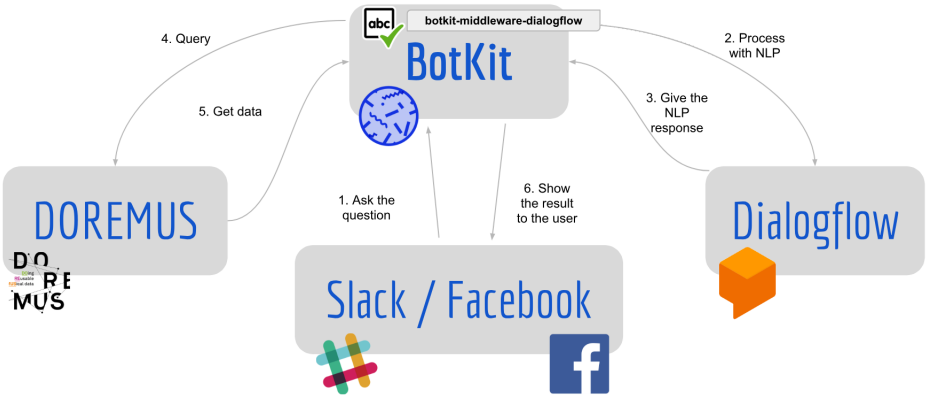

Figure 4.1: The high-level architecture of DOREMUS Bot.

First of all we can find the client, that in the case of our development, testing and validation process, has been *Slack*. Of course, it can be any of the clients which support the installation of the bot on it (*Telegram*, *Facebook Messenger*, etc.). The use of Slack let us to exploit the beautiful *Slack Cards*, to make the answers of our bot (works, artists, performances) prettier and easier to understand in a glance. A set of examples will be provided in the last chapter.

The second part of the architecture is represented by the NLP. In this case, as told in the previous chapter, we used *Dialogflow*, to exploit its advanced slot-filling techniques and its NLU power.

However, we didn't use *Dialogflow* on its own, exploiting the direct integration with *Slack*, but we used something that we placed in the middle of the two: *Botkit*. *Botkit* is a bot-making toolkit that aims to ease the building process of a bot, potentially exploiting different NLPs and/or different clients. In our case, *Botkit* has been deployed on a web server, and thanks to the *NodeJS* code we were able to come up with a series of features (like the spell checker) that would have been impossible to reach with a simple (direct) integration between *Slack* and *Dialogflow*. We'll talk more about that in the following paragraphs. The communication with the *Dialogflow* NLP is done thanks to a modified version of the `botkit-middleware-dialogflow`[14]: there will be a paragraph dedicated to this middleware in the following pages.

The last part of the architecture is of course represented by the data source: *DOREMUS*. We talked about that in the previous chapters, but from the architecture is important to notice how the knowledge base is queried: each query is dynamic, in the sense that according to the intent, the number of filters and the desired results wanted by the user, the query will have a different shape and a different content. The code in the web server is able to add different pieces of queries according to what *Dialogflow* is able to understand and to provide as output values of the API.

The bot code is organized in this way:

```
bot.js
spell-checker-middleware.js

config/
  .env

doremus/
  bot_functions.js

slack/
  slack_io.js
  slack_cards.js

facebook/
  facebook_io.js
  facebook_cards.js

dialogflow/
  webhook_server.js
```

where:

- `bot.js` is the core file of the bot. It contains the code to declare the fundamental libraries, to start the RTM, and to load the hears methods.
- `spell-checker-middleware.js` is the custom module to perform the spell-checking before sending the received sentences to the NLP.
- `.env` is the secret file (to place into the `config` directory) containing all the tokens for *Slack*, *Facebook* and *Botkit Studio*.
- `doremus/` contains the files related to the access to the informations of the *DOREMUS* knowledge base: in this case, contains a single file (`bot_functions.js`) that contains all the queries that the bot can do to *DOREMUS*.
- `slack/` is a directory containing the files related to *Slack*:
  - `slack_io.js` contains the methods to receive the sentences sent through *Slack* and processed by the NLU.
  - `slack_cards.js` contains the code to build the *Slack* cards to make the answers prettier.
- `facebook/` is a directory containing the files related to *Facebook*:
  - `facebook_io.js` contains the methods to receive the sentences sent through *Facebook* and processed by the NLU.
  - `facebook_cards.js` contains the code to build the *Facebook* cards to make the answers prettier.
- `dialogflow/` is a directory containing the webhook useful for the *Dialogflow* fulfillment phase. This flow is totally detached from our original architecture (which uses *Botkit*), and it's useful only for the lightweight version of the bot which can be used with *Google Home*.

4.2 Entities

The sentences that the bot is required to recognize are of course full of informations related to the *DOREMUS* knowledge base. This means that the NLP has to be able to understand some "entities" (informations, words, piece of sentences) that are not in the standard language, but are related to the *DOREMUS* or, in general, musical world. Our bot is equipped with three different entities:

- `doremus-artist`
  Contains all the artists in the *DOREMUS* knowledge base. It's organized in a *key-value* pair where the *key* is the unique id of the artist inside the knowledge base, and the *value* is the set of full names and surnames of the artist, in all the available language of the knowledge base. The query with which the artist were retrieved is the following:

```
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)
```

  The artists are ordered by descending count of works, in order to have the most famous artists at the top of the entity dictionary, and let *Dialogflow* find the most famous artists in case just a surname is given. The result of the query has been taken as a `.csv` file and then processed a little bit (delete the count, join the names and surnames columns, duplicate the composer id, eliminate the pipes and some special characters) in order to fit with the *Dialogflow*'s constraints.

- `doremus-instrument`
  Contains all the instruments of the `iaml/mop/` dictionary of the *DOREMUS* knowledge base. Also in this case the *key-value* pair dictionary has as keys the id of the instrument, and as values all the synonyms (names in different languages). The query thanks to which we retrieved this entity is:

```
SELECT DISTINCT ?instrument
  (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
```

- `doremus-genre`
  Contains all the genres of the `iaml/genre/` dictionary of the *DOREMUS* knowledge base. The query thanks to which we retrieved this entity is:

```
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
```

4.3 Intents

The intents are grouped in a simple and clear way, according to what the user wants to retrieve from the *DOREMUS* knowledge base:

- `works-by`
  Retrieves a set of works according to different filters (artists who composed the works, instruments used, music genre and/or year of composition).

- `find-artist`
  Finds a set of artists according to some filters (number of composed works, number of works of a given genre, etc.).

- `find-performance`
  Propose to the user a future performance (that can be filtered by city and/or date period), or shows to the user the details of a past performance.

- `discover-artist`
  Shows a card with a summary of an artist, with its birth/death place and date, a picture and a little bio. After the card visualization, a set of works of the artist (connection with the `works-by` intent) can be asked.

Now we're going to go deeper in the intent descriptions.

4.3.1 Retrieving a set of works

The `works-by` intent is the most complex one in the entire bot's intents set. It can retrieve a certain number of works from 1 to L, where L is the number specified by the user if it's smaller than the number of available works. Otherwise, if it's greater, all the avilable works are returned. Its default value (if not specified by the user) is 5.

The filters can be various:

- **Artist:** the artist name (full or surname).
  "Give me 3 works composed by Bach"
- **Instruments:** the instrument(s) (in `and`/`or` relation).
  "Give me 2 works for violin, clarinet and piano"
  "Tell us 4 works for violin or piano"
- **Genre:** the music genre.
  "List me 10 works of genre concerto"
- **Composition period:** the period in which the work has been written.
  "Tell me one work composed during 1811"
  "Give us 3 works written between 1782 and 1821"

The filters can be specified in every way: this means that the user can specify all the available filters, some of them and even none. If the number of filters in the first query is smaller than two, the bot asks the user if he wants to apply other filters. The user can answer positively or negatively, and then decide which kind of filter (and the value) to apply. It's important to notice that the kind of filter and the value can be specified together or not; let's see an example to make it more clear.

First of all, we are in the context in which the bot asks the users if he wants to apply some filters:

> *Please give me 3 works by Beethoven!* - **User**
> *You told me few filters. Do you want to add something?* - **Bot**
> *Yes!* - **User**
> *Ok, tell me what* - **Bot**

In this case, two scenarios can happen:

> *The composition year* - **User**
> *Of course! Tell me the time period.* - **Bot**
> *Between 1787 and 1812* - **User**

or directly...

Only works composed between 1787 and 1812 - **User**

The **dynamic** query used for the works by artist intent is the following:

```
SELECT SAMPLE(?title) AS ?title, SAMPLE(?artist) AS ?artist,
       SAMPLE(?year) AS ?year, SAMPLE(?genre) AS ?genre,
       SAMPLE(?comment) AS ?comment, SAMPLE(?key) AS ?key
WHERE {

  ------------------------------------------------------------ STATIC SECTION
  ?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 .
    BIND (year(?comp) AS ?year) .
    ?expression mus:U11_has_key ?k .
    ?k skos:prefLabel ?key
  } .

  ------------------------------------------------------------ DYNAMIC SECTION

  ------------- Instrument
  ?casting mus:U23_has_casting_detail ?castingDetail .
  ?castingDetail mus:U2_foresees_use_of_medium_of_performance
                 / skos:exactMatch* ?instrument .
  VALUES(?instrument) {
    (<http://data.doremus.org/vocabulary/iaml/mop/' <instrument> '>)
  }

  ------------- Genre
  VALUES(?gen) {
    (<http://data.doremus.org/vocabulary/iaml/genre/' <genre> '>)
  }

  ------------- Artist
  VALUES(?composer) {
    (<http://data.doremus.org/artist/' <artist> '>)
  }
```

```
    ------------- Composition Year
    FILTER (?comp >= "' <yearstart> '"^^xsd:gYear AND
            ?comp <= "' <yearend> '"^^xsd:gYear)
  }
  GROUP BY ?expression
  ORDER BY rand()
  LIMIT ' <num> '
```

Some particularities: first of all, the property that let us obtain the instrument: `mus:U2_foresees_use_of_medium_of_performance / skos:exactMatch*`. The "`mus:U2_foresees_use_of_medium_of_performance`" is a property of the *DOREMUS* ontology that allows to connect a generic *castingDetail* to a specific *instrument*. The "`skos:exactMatch*`" is instead a way to allow the query to search not only in the actual vocabulary used by us (`iaml/mop/`) but also in other vocabularies that contains an *equivalent* of that instrument. The "`*`" symbol means *"one or more"*. It's also important to notice the particularity of the instrument information because, as previously said, it can be a list of instrument connected by an `and/or` logic operator. In the query implemented inside the `works-by` intent, this doesn't mean only to create dynamically the query (like for the other informations) but also to differentiate the code between a single-instrument filter, an "`and`" multi-instrument filter and an "`or`" multi-instrument filter. The differences are shown here:

```
---- AND CASE
if (strictly === "and") {
  for (var i = 0; i < instr.length; i++) {
    query += '?casting mus:U23_has_casting_detail ?castingDetail' + i + ' .
            ?castingDetail' + i + '
              mus:U2_foresees_use_of_medium_of_performance
              / skos:exactMatch* ?instr' + i + ' .
            VALUES(?instr' + i + ') {
              (<http://data.doremus.org/vocabulary/iaml/mop/' + instr[i] + '>)
            }'
  }
}

---- OR CASE
else {
  query += '?casting mus:U23_has_casting_detail ?castingDetail .
            ?castingDetail
              mus:U2_foresees_use_of_medium_of_performance
              / skos:exactMatch* ?instr .
            VALUES(?instr) {'

  for (var i = 0; i < instr.length; i++) {
```

```
      query += '(<http://data.doremus.org/vocabulary/iaml/mop/' + instr[i] + '>)'
    }
  }
```

In the `and` case we add (with a `for` operator that loops through the `instr` array) a `castingDetail` for each instrument, forcing its value to that specific instrument. In the `or` case, instead, the `castingDetail` is just one and can assume a set of values filled with the `for` operator that loops through the `instr` array.

4.3.2 Finding some artists

One powerful intent of the bot is `find-artist`, that can retrieve a set of artists given some filters. The filters can be for example the birth date or the birth city but the interesting functionality comes thanks to the sorting of the result: having a descending filter, the artists with more works comes first; so, we made the assistant answer questions like:

> *Find the five artists who composed more concerto works* - **User**

or,

> *Find the 3 artists, born between 1752 and 1772, who composed more works* - **User**

or,

> *Find one artist, born between 1752 and 1772, who wrote more works for clarinet* - **User**

and so on.

The query of the intent is the following:

```
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 .

  ------------- Genre
  VALUES(?gen) {
    (<http://data.doremus.org/vocabulary/iaml/genre/' <genre> '>)
  } .

  ------------- Instrument
  ?casting mus:U23_has_casting_detail ?castingDetail .
  ?castingDetail mus:U2_foresees_use_of_medium_of_performance
                / skos:exactMatch* ?instrument .
  VALUES(?instrument) {
    (<http://data.doremus.org/vocabulary/iaml/mop/' <instrument> '>)
  } .

  ------------- Birth date
  FILTER ( ?b_date >= "' <startdate> '"^^xsd:date AND
           ?b_date <= "' <enddate> '"^^xsd:date ) .

  ------------- Birth city
  FILTER ( contains(lcase(str(?birth_place)), "' <city> '") ) .

}
GROUP BY ?composer
ORDER BY DESC(?count)
LIMIT ' <num> '
```

4.3.3 Finding some future and past performances

The bot is also capable of giving the user some future and past performances and events. The place and the time period can be, of course, specified in the query. The query is the following:

```
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 >= "' <startdate> '"^^xsd:date AND
           ?time <= "' <enddate> '"^^xsd:date ) .
  FILTER ( contains(lcase(str(?placeName)), "' <city> '") )
}
GROUP BY ?performance
ORDER BY rand()
LIMIT ' <number> '
```

It's important to notice the `ecrm:P89_falls_within*` property. It selects not only the place where the event is foreseen, but also the places where that place is contained (so, for example, a city). The "`*`" means, as usual, *"one or more"*. And here there is an example:

> *Tell me one event in Paris!* - **User**
> *In which period?* - **Bot**
> *Next month* - **User**

4.3.4 Know more about an artist

The last intent of the bot is `discover-artist`. This is the richest intent in terms of informations provided: it returns the artist summary in a *Slack card*, with a picture, a short bio, the birth/date places and dates. The query is the following:

```
SELECT ?name, ?bio, ?image,
       xsd:date(?d_date) AS ?death_date, ?death_place,
       xsd:date(?b_date) AS ?birth_date, ?birth_place
WHERE {
  VALUES(?composer) {(<http://data.doremus.org/artist/' <artist> '>)} .
  ?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")
}
```

And here there is an example:

Tell me something about Mozart - **User**

or...

What do you know about Beethoven? - **User**

The bot is also capable of mixing the `discover-artist` and `works-by` intents: after asking the bot of the details of an artist, the user can retrieve its works, applying the usual filters. Let's make an example:

Tell me something about Mozart - **User**
[Result with bio, picture, birth/death date/place]
Now give me 5 of his works, written for clarinet - **User**
[Result with the 5 works of that artist]

In this case, indeed, the `works-by` intent is triggered after the `discover-artist` intent: the artist name, is obtained from the context generated with the `discover-artist` intent, which is of course still active.

4.4 The spell checker middleware

One of the most important aspect of the bot is the misspelling and language detection mechanism, which make it smarter and more reliable. Everything is implemented in an additional middleware located upstream, before the Dialogflow middleware actually send the request to the NLP engine. In this way we intercept every message typed by the user, fixing any typos which otherwise would affect the Dialogflow capabilities to detect intent and entities. In addition we can set the language of the agent, making the bot capable of changing the spoken language at each message. Of course the language detection is a fundamental step in order to use the proper dictionaries during misspelling check.

The core implementation of the middleware could be splitted in 3 parts:

1. Language detection
2. Misspelling check
3. Setting of the variables used from the next middleware

In the first step we use the *Google Translate API*. For this reason we initially prepare few parameters that we are going to attach to the HTTP request. Among the many, the most important is `sl` that stands for "source language": setting it to "auto" we let the *Google Translate API* detecting it. This information is in the response JSON.

```
// LANGUAGE CHECK

// prepare arguments for the request to Google Translate API
var url = "https://translate.googleapis.com/translate_a/single"
var parameters = {
  q: message.text,
  dt: 't',
  tl: 'it',
  sl: 'auto',
  client: 'gtx',
  hl: 'it'
};

request({url:url, qs:parameters}, function(err, response, body) {

  if (err) {
    console.log("Error during language detection!");
    next(err);
  }

  // get language from json
  var res = JSON.parse(body);
  var lang = res[2];

  // update accordingly the speller and the global var
  if (lang == "fr") {
    speller = spellFR;
    currentLang = "fr";
  }
  else if (lang == "en") {
    speller = spellEN;
    currentLang = "en";
  }
  //otherwise don't change anything
```

In the second step, after setting up the speller accordingly to the language detected, we perform the misspelling check: for each word (splitting by space) we check if is it in the dictionary (in other words well-spelled) otherwise we replace it with the most similar corrected word, if it exists:

```
// SPELL CHECKING

// perform the misspelling with the (potentially) updated speller
var cleanMessage = performMisspellingCheck(message)
message.text = cleanMessage;
```

```
// fill the language field in order to send it to dialogflow api
message.language = currentLang;

next()
```

The third step is the simplest one: we simply change the text of the message, the object that will handle by the next middleware, with the new, misspelling-free sentence; in addition we set an additional field inside it: the language one, that will be used by `botkit-middleware-dialogflow` to set the language of the agent before sending the request. This code, differently from the previous one, has to be inserted directly inside the `botkit-middleware-dialogflow`: after opening a pull-request to the original author, this code was merged into the author's repository.

```
if (message.lang) {
  app.lang = message.lang;
}
else {
  app.lang = 'en';
}
```

One minor aspect is behind the variable `showNewSentence`. We want to give the user a feedback when a correction occurred. When it is set to true, the first next answer will include a warning for the user, indicating the actual sentence that has been processed.

4.5 Supported platforms

4.5.1 Textual approach: Slack and Facebook

The two supported platforms to communicate with the bot in a textual way are *Slack* and *Facebook Messenger*. Both platforms have been integrated in the *Botkit* environment. The mechanism through which the sentences are heard is simple to understand: the hears methods are instantiated, one for each intent: this methods represent the actions that the bot has to do when an intent is triggered. Usually, without a bot-building environment like *Botkit*, these actions would have been done directly by the "fulfillment" phase in *Dialogflow*. In our case, instead, *Dialogflow* is just an NLP! So, for each platform we use (in this case *Slack* and *Facebook*) we need to instantiate the methods that needs to be triggered when an intent is detected.

The prototype of a generic hears functions (in our code) is the following:

```
// HELLO INTENT
module.exports.hello = botVars.slackController.hears(['hello'], 'direct_message
  , direct_mention, mention', botVars.dialogflowMiddleware.hears, function(bot,
   message) {

  bot.reply(message, message['fulfillment']['speech']);
});
```

The `hello` variable (put in the module.exports object to be available from the code in other files) is instantiated as an `hears` (in this case of the `slackController` object, but it could have been `fbController`). The code, in the case of the `hello` intent is really simple, because it just needs to reply with the same answer provided by the *Dialogflow* NLP. For the other intents, the code is obviously more complicated because it needs to get the informations from the *DOREMUS* knowledge base, as explained in the previous chapters.

All the textual results that the bot provides are made "prettier" thanks to cards. Both in the *Facebook* and in the *Slack* chat, the results are indeed organized in "key-value" pairs following the JSON syntax. In this way, for example, *Mozart* is presented as the value of the key *Composer*, or *05/12/1791* is the value of the *Death date* key. An example of the code of a card creation is the following:

```
module.exports.getPerformanceCard = function getPerformanceCard(title, subtitle
  , placeName, actorsName, date) {
  var performanceAttachment = {
    "type": "template",
    "payload": {
      "template_type": "list",
      "top_element_style": "compact",
      "elements": [
      {
        "title": title,
        "subtitle": subtitle
      },
      {
        "title": "Where",
        "subtitle": placeName
      },
      {
        "title": "When",
```

```
          "subtitle": date
        },
        {
          "title": "Actors",
          "subtitle": actorsName
        }
        ]
      }
    }
    return performanceAttachment;
```

4.5.2 Vocal approach: Google Home

Chapter 5

Evaluation

Chapter 6

Conclusions

References

[1] DOREMUS: DOing REusable MUSical data. — *http://www.doremus.org/*

[2] Natural Language Input for a Computer Problem Solving System. 1964. Bobrow, Daniel. — *http://dspace.mit.edu/handle/1721.1/5922*

[3] ELIZA: A computer program for the study of natural language communication between man and machine. 1966. Weizenbaum, Joseph. — *https://dl.acm.org/citation.cfm?id=365168*

[4] Augmented Transition Networks for Natural Language Analysis. 1969. Woods, William. — *https://eric.ed.gov/?id=ED037733*

[5] NLP, NLU, NLG and how Chatbots work. — *https://chatbotslife.com/nlp-nlu-nlg-and-how-chatbots-work-dd7861dfc9df*

[6] Attention-Based Recurrent Neural Network Models for Joint Intent Detection and Slot Filling. 2016. Liu, Bing and Lane, Ian. — *https://arxiv.org/abs/1609.01454*

[7] Wit.ai: Turn what your users say into actions. — *https://wit.ai/*

[8] Snips: Using Voice to Make Technology Disappear. — *https://snips.ai/*

[9] IBM Watson: Does your business think? — *https://www.ibm.com/watson/*

[10] Amazon Lex: Build conversation bots. — *https://aws.amazon.com/lex/*

[11] Microsoft LUIS: A machine learning-based service to build natural language into apps, bots, and IoT devices. — *https://www.luis.ai/*

[12] SAP Recast.ai: A collaborative bot platform. — *https://recast.ai/*

[13] Chatbots made easy with Dialogflow. An overview on the architecture. — *https://blog.huhtanen.eu/2017/10/15/chatbots-made-easy-dialog-flow.html*

[14] The GitHub repository of the Botkit middleware for Dialogflow. — *https://github.com/jschnurr/botkit-middleware-dialogflow*