UCLouvain

epl

École polytechnique de Louvain

JAQ: A Chatbot for Foreign Students

Authors: **Arnaud Gellens, Simon Gustin**
Supervisor: **Yves Deville**
Readers: **Yves Deville, Cédrick Fairon, Christine Jacqmot**
Academic year 2018–2019
Master [120] in Computer Science and Engineering

Abstract

The primary objective of this master thesis is to design a chatbot intended to assist international students at UCLouvain. The produced chatbot, named *JAQ*, is aimed to be a Proof of Concept of an additional way for those students to obtain general information about the university. Furthermore, we characterize the modern approach to chatbot conception, identify key challenges to overcome and analyze popular chatbot frameworks employed in the industry.

To create an efficient chatbot, employing a framework was the best option, both in terms of conception speed and performances. We investigated both *IBM Watson Assistant* and *Google Dialogflow* and determined the latter one was more suited to the context. The scope of topics relevant to this chatbot being very extensive, we implemented a complete system on top of this framework to aid with data management and maintenance. The implementation of this system, along with the conception steps of the chatbot, are thoroughly described.

Several experiments were performed with the final version of *JAQ* in order to assess its quality. Namely, we measured its comprehension performance and quantified the perceived quality of the user experience. From a performance viewpoint, the chatbot correctly understands the user message almost three times out of four. Regarding the perceived quality, experiments showed the user experience to be satisfying but definitely improvable.

We can thus conclude *JAQ* successfully fulfilled its goals, even though numerous aspects could be enhanced. Therefore, we propose several potential improvements to both the produced chatbot and the employed framework. We also discuss the limitations of the additional system we built on top of this framework.

Acknowledgements

First of all, we would like to thank our supervisor, Pr. Yves Deville and our technical advisor, Dr. Christine Jacqmot for their advice and support throughout the year.

Then, we would like to thank Pr. Cédrick Fairon to accept becoming one of our readers.

We also want to express our gratitude to all our friends and relatives who accepted to test various versions of our chatbot. Their feedback was a great help. We want to specifically thank the following people who performed the last usability experiment, which we draw parts of our conclusions on: Anne Draelants, Morgane Druez, Julien Gomez, Lucie Gustin, Michel Gustin, Françoise Meulenberghs, Robin Hormaux, Romain Hubert, Thibaut Piquard, Carolina Eugenia Unriza Salamanca, Kevin Stoffler, Trong-Vu Tran and Vincent Vandervilt.

Finally, we would like to express our deepest thankfulness to Charles-Edwin de Brouwer for proofreading parts of this master thesis, and Daniel Gellens, Julien Gomez and Trong-Vu Tran for proofreading it in its entirety.

Contents

List of Figures

Chapter 1

Objectives

This chapter first outlines the problems this master thesis aims to address and the overall objectives it tries to fulfill. Finally, it states the research questions that will be answered.

1.1 Problem statement

As the title of this master thesis suggests, the purpose of this collaboration is the design of a chatbot meant to assist future international students at UCLouvain. As we will explain, the point is to give those students an additional way of acquiring information about the organization of the university.

To a broader extent, in any large entity, it is often required for its members to be able to find relevant information about the organization of the corporation and tasks within it. Undeniably, having incorrect information can lead to misunderstandings, organizational problems and time losses. Moreover, information querying is something that new members of a corporation often need to do. Therefore, having one or several ways to acquire correct and relevant information should be regarded as quite important, and that even for basic information. It should also be taken into account that the members of the corporation could speak different languages or be located in different places.

UCLouvain being such an entity, it is important for its members (professors, assistants, secretaries, students, etc.) to be able to gather information regarding the organization of the university life, specific procedures and required documents. At the time of writing this document, members of the university have merely three main sources of information:

- Searching for information on UCLouvain's official website[2], notably in the Frequently Asked Questions sections;
- Searching for information using internal or external web search engines;
- Communicating with members in charge of the organization, e.g. central and departmental secretaries or teaching assistants, by phone, email or in person.

Expectedly, each technique to retrieve information has its qualities and limitations. Namely, searching for information on UCLouvain's website yields correct information but can be slow and

cumbersome to accomplish. Moreover, some parts of the website are only available in French. Search engines can return more information faster, by finding older versions of the official website or other websites, but this information might be outdated or incorrect.

Contacting an employee with the relevant knowledge is of course the fastest and most reliable way to request correct and relevant information. However, it can be hard to obtain the contact details to reach the right person (if any). Plus, there are different downsides depending on the communication channel: for phone and in person conversations, both people need to fix an appointment; to reach someone in person, both interlocutors need to be nearby; contacting the person by email can be slower. Each solution thus requires a high availability of the staff.

Furthermore, it is not always possible for the university to employ enough people to reply to all the questions and requests other members could have, given the large number of people in need of specific information. Moreover, a large amount of such requests can be a burden to the people in charge of answering them. Lastly, we should note that those employees are not necessarily aware of all the information and the specific cases which might be applicable, given how large and diverse the scope is, making their task even harder.

The point of this master thesis is thus to provide an additional way to acquire information. Our attention is focused onto the members of UCLouvain for whom the process of information gathering is hardest, that is, future students from abroad. As can be imagined, retrieving relevant pieces of information is more difficult for them because they are located far from the sites of the university, usually speak a different language than the prominent one in UCLouvain and generally have different cultures, habits and legislations.

The amount of information relevant to those students being extremely large, we limited ourselves to a smaller scope, notably related to information about accommodations, registration to the university, courses and exams, presentation of Belgium, etc. The scope however stays large, which constitute a challenge both during conception of the system and experiments.

As an example, we aim the produced chatbot to be able to have the following kind of conversation with a future student:

> *Student:* Can you tell me if I need a visa to come to Belgium?
> *Chatbot:* First, I need to know if you are from the European Union?
> *Student:* I am.
> *Chatbot:* Then, you do not need a visa. You need to have an ID or a passport, though.

1.2 Research questions

This master thesis tries to answer two different research questions, although closely related, one of them being more theoretical. Those research questions are as follows:

- *How do modern chatbots usually work internally and what are the typical approaches adopted to design modern chatbots? Additionally, what are the typical tools used in chatbot creation?*
- *Can we design a chatbot intended to assist international students of UCLouvain? Would*

such a chatbot be usable in real life situations?

For the first question, we start by defining chatbots in chapter 2 and what challenges are to be overcome in chapter 3. Then, we describe in chapter 4 the specificities of modern chatbots, as well as the common design tools and technologies related to them. This thesis is mostly concerned with experienced industrial technologies rather than the latest development from research laboratories.

For the latter question, we aim to design a chatbot meant to assist international students who would like to study at UCLouvain or are in the process of registering at UCLouvain. We describe the design process in chapter 5 and the tools we implemented to facilitate that process in chapter 6. We should note that we are interested here in the design of a chatbot system for the use case we defined, and not in the creation of a complete framework to build them. The produced chatbot is then assessed and the results are discussed respectively in chapters 7 and 8.

As the research question states, the produced chatbot can serve as a Proof Of Concept demonstrating that modern chatbots are suited in the context we described earlier. It can be tested at `http://tfe-gustin-gellens.info.ucl.ac.be` inside the network of UCLouvain. Moreover, the source code of the scripts developed for this master thesis, along with the formatted data, is available on the following *GitHub* repository: `https://github.com/gellens/Master_thesis_JAQ_code`.

Chapter 2

What is a chatbot?

This chapter contains definitions of concepts that will be used throughout the dissertation. We notably define what chatbots are, as well as their usages, qualities and limitations. Ultimately, we describe the internal functioning of modern chatbots.

2.1 Useful definitions

Intuitively, a chatbot can be assimilated to an automated interlocutor. It is a computer program that one or several users can have a conversation with, using a voice-based or text-based interface. Chatbots are usually meant to either assist users to perform a certain task or provide them with requested pieces of information. Some other chatbots are simply intended to mimic human behavior.

Before defining more formally what chatbots are and are not, there are a number of concepts that need to be explained.

Natural Language Intuitively, a *natural language*, or *ordinary language*, is a human language that can be employed in everyday life, e.g. English or French. The term "natural language" is used to distinguish them from machine languages and programming languages in computer science, from formal languages in logic, and from other means of communication in linguistics [3].

Giving a more formal and precise definition has proven to be very hard, as there is some controversy on whether some languages can be considered as natural languages or not [4].

Conversation Called *dialog* or *dialogue* when occurring between only two people, *conversation* has the same meaning here than traditionally: an exchange of information between two or more people, with the communication being made in a back-and-forth manner, i.e. with several turns [5]. Conversations usually refer to exchanges of complex information, which implies it cannot be done through any communication channel. Most of the time, a conversation is voice-based or text-based.

Utterance An *utterance* is one or several sentences that are spoken by a speaker in a conversation and that can be considered the smallest unit of speech in a dialog. In text-based conversations, an utterance is often synonymous to a message.

Conversation scope A conversation is usually limited to a few topics, which are called the *conversation scope* (or *dialog scope*). Utterances that are off-topic will thus be deemed out-of-scope.

2.2 Chatbot definition and usages

2.2.1 Chatbot definition

A *chatbot* (also called *chatterbot*, *conversational agent* or sometimes *spoken dialogue system*) is a computer program able to converse about certain predefined topics in natural language with one or several human users, and this without the intervention of another human being. We will restrain ourselves in this dissertation to the study and design of the ones supposed to interact with one person at a time, meaning their conversations are always dialogs with one interlocutor.

The broad operations performed by a chatbot are illustrated in figure 2.1.

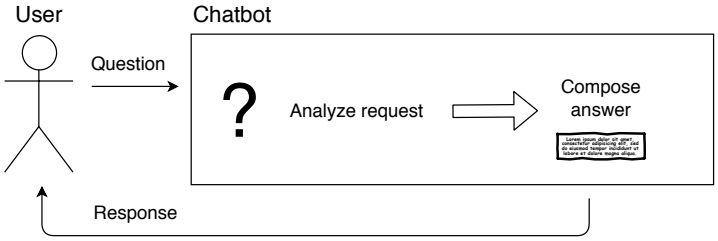


Figure 2.1: Chatbot abstract view. Inspired by [6]

Difference with other systems

When talking about chatbots, there can be a potential confusion with other types of systems, either because they are sometimes called "chatbots" as well, or because they share a certain number of similar characteristics.

Firstly, it is important to differentiate chatbots from what we could call "chatroom bots" (often called chatbots as well). While the first ones are able to understand natural language, the latter simply react to a finite set of commands relevant to chatrooms. While being very useful in specific cases, those bots are obviously way simpler and rely simply on keyword detection.

The same difference exists when we talk about phone automatic operators (or "interactive voice responses", cf. appendix A). Those are the automatic answering machines that can pick up the phone (usually in enterprise settings) and which ask the user to press certain keys to explain their request. Obviously, there is here no notion of understanding the user. However, the component of those systems which decides what to tell the user given what they chose previously is very reminiscent of one of the typical components of chatbots we describe in section 2.3.4. We could thus argue that such systems are actually very simple instances of chatbots.

Moreover, some more advanced phone systems exist (notably in use in the United States) where those operators actually ask the user a question and rather than expecting a key press, directly listen

through the phone microphone and try to understand what the user says. Those systems definitely correspond to what we call chatbot systems in this master thesis.

Finally, it is possible to relate chatbots as defined here them with other systems such as information query languages or command line interfaces, as they have common characteristics. Once again, one might argue that they should be considered chatbots as well, but we do not draw the line there in this dissertation.

Prominent types of chatbot systems

Different characteristics can be taken into account in order to classify chatbots. It seems generally accepted that the two most fundamental features are the chatbot's primary purpose and its interface to interact with the user. The latter will be discussed in details in subsection 2.3.2.

We will thus here outline a classification of chatbot systems based on their objective and the types of problems they address.

Chit-chat chatbot First, a chatbot's objective can simply be to make research about conversational agents. The purpose of such a chatbot is then just be to appear as human as possible and to be nice to interact with. That kind of chatbot is often tested against the Turing test [7], where the chatbot is supposed to make its interlocutor believe they are talking to a human being. An example of such a chatbot is Cleverbot [8] whose primary goal is to make small talk with people and learn from those conversations.

Assistant chatbot In this case, the purpose is to use the produced chatbot as an assistant: it would have some internal knowledge that the user could easily query using natural language. This kind of chatbot is more and more frequent on websites, especially commercial ones. Such a chatbot would for example be able to advise the user, to help them find a certain function on the website or to aid them to fix a problem they are encountering with a product or a service from the website. In other words, they fulfill a task of client support that is traditionally accomplished by human operators. In this dissertation, this is the type of chatbots we are mostly interested with.

System interface chatbot A last type of chatbot corresponds to the ones capable of being an interface to a more complex system. The user can ask such a chatbot to perform actions for them, either to change the state of the system or to query some information about it. This kind of chatbots is especially useful if there is a large number of tasks that the chatbot is able to perform or if those tasks would be tedious to perform using another type of interface. This kind of chatbot is related to query languages and to command line interfaces.

Note that the term "modern chatbot" usually refers to either of the two last types.

"Hybrid" chatbot As one could expect, the purpose of a chatbot can also be a combination of those listed above. The very popular *Google Assistant* [9] or *Amazon Alexa* [10] are examples of that: they usually assist the user, for example by telling them the time or the weather, but can also be asked to perform tasks such as ordering food or setting a timer, and can even have a small conversation with the user.

2.2.2 Qualities

As can be seen in appendix A, chatbots have been popular for quite a long time and are getting better and better from a user's perspective. Their consistent popularity is largely due to their underlying qualities. The principal ones are described here.

User-friendliness and power

Clearly, their main quality is ease of use and user-friendliness. Indeed, one does not need to have specific skills or knowledge in order to utilize them, as the interface takes place using natural language which anyone is familiar with. They are thus extremely appropriate in situation where non-technical people need to interact with a complex system, or need assistance with such a system. Of course, this quality is only present in well-designed chatbots: very often, chatbots can feel clunky to use. We will discuss potential causes of this clumsiness in chapter 3.

This user-friendliness is not just a question of not needing to learn to use them. Chatbots also are a very good way to let users have complex interactions with a system, without scaring them away, since not all the possible interactions are presented to the user, while still letting them easily access the option they want. For instance, while a user would need to search for one or several options in different menus and windows and click on different buttons to do something technical in a traditional interface, they can simply ask the chatbot to perform the action they want or give them the information they are seeking.

To summarize, chatbots can be seen as a user-friendly interface that would hide the complexity of a system.

Personalized user experience

Chatbots are also a very good tool to make a user experience more personalized. In this context, personalizing the user experience would mean to have a chatbot remember the name and situation of a user, and that whatever the time since the last conversation. Further more, researches [11] and implementations [12] have been conducted of chatbots recognizing certain personality traits by analyzing the user's utterances, and act differently depending on what it detected. Obviously, the chatbot in question must be designed to have personalized dialogs in order to benefit from this quality.

Easily scalable with few additional costs

Another quality, which certainly explains why chatbots are quite popular in commercial settings, is the fact that they allow to automate quite easily different tasks for which a large number of employees would otherwise be required. Customer support is an example of such a task, and we can indeed observe that mostly big companies are able to hire employees for that. The fact that chatbots are available at any time of the day or the week is undoubtedly a major quality as well.

Moreover, as a business brings in more customers, the amount of work required for these tasks grows rapidly. To fulfill them, this company will end up in need of hiring a prohibitive number of

people. On the other hand, making a chatbot handle those tasks scales very easily: the company can simply deploy more instances of the chatbot and load balance request between them.

2.2.3 Limitations

As for any technology, chatbots have a certain number of weaknesses and limitations. It is extremely important not to lose sight of those limitations when designing a chatbot, since some of them can be mitigated if the chatbot is carefully crafted. It is even more important to think about the limitations and qualities when choosing whether a chatbot is a good solution for the problem at hand.

We should note that the limitations explained here are inherent to chatbots because they are induced by their very definition. Other limitations and weaknesses can exist, but depend on the design approaches and technologies used.

Clumsiness in conversations

The most prominent and noticeable limitation of modern chatbots, as well as older technologies, is their clumsiness. Indeed, they often feel very "robotic" when interacted with, especially for badly designed ones. This is usually due to a bad perception of context, a bad recognition of user intentions and/or low memory capabilities. It also often happen that a chatbot does not understand a simple utterance because it is outside of the scope it was trained for. This can make the interaction with a chatbot irritating to users, especially for users unfamiliar with the technology. We describe a certain number of causes for that clumsiness in chapter 3.

Slow interactions

When employed as a user interface, chatbots are quite slow in comparison with other types of interfaces: clicking the right button is usually faster than uttering (by text or voice) what one wants. This limitation being offset by the potential power and ease of use of chatbots, we could argue chatbots are similar to command line interfaces, which are extremely powerful but can be quite slow to use in comparison to other interfaces.

Possibly incomplete information

A weakness that often appears with chatbots made with poor care is the fact that information delivered by the chatbot can be incorrect. This is quite problematic when the chatbot is meant for support purposes. However, this problem is not especially limited to poorly designed chatbots. Indeed, in some cases, delivering correct information in a limited amount of words (for time and readability reasons) is impossible, because this information might not be available to the designer, change too frequently or need lots of details to really be complete. It is thus up to the designer to decide whether it is acceptable to deliver incorrect or incomplete information.

Uncertain scope

Another weakness that is visible from the user's viewpoint is the difficulty to know what the chatbot is capable of. Indeed, a chatbot being designed for a given purpose, it won't be capable of helping with topics and questions that are out of its scope. This inconspicuity of the capacities of a chatbot is actually a consequence of trying to hide its power to the user, to prevent scaring them away. We could thus see it both as a quality and a limitation.

Intimidating interactions

We can notice a last limitation that has to do with users: it seems that a certain number of people don't dare using chatbots because they don't know what it is and think they might be disturbing an employee. This is quite often the case when a messaging window pops up without notice on a website that the user is visiting.

2.3 Internal functioning

We will now paint an overview of modern chatbot technologies. This section is about the general architecture which powers most modern chatbot systems. Nevertheless, different approaches can be adopted for some of their components, which will be discussed here as well.

It is important to understand that most chatbot designers use a framework to speed up chatbot creation. Hence, most of the logic and technical challenges are addressed (at least partly) by those frameworks and not by the designers themselves. On the other hand, the concrete chatbot design, i.e. the creation of the conversation logic and the concrete questions and answers the chatbot would have, will be discussed in chapter 5.

2.3.1 General architecture and usual components

Modern chatbots are usually composed of three main parts:

- a **User Interface (UI)**, which allows the user to interact with the chatbot, and vice-versa;
- a **Natural Language Understanding (NLU) component**, in charge of "understanding" the utterances of the user;
- a **dialog management component**, meant to decide what to answer and which actions to perform.

Broadly speaking, as represented in figure 2.2, the user chats with the chatbot thanks to the UI, by sending and receiving messages. When the user sends the chatbot a message, it is forwarded to its NLU component which parses it and turns it into exploitable data. This data is then exploited by the dialog management component of the chatbot, which formulates a suitable answer and performs the requested actions (if any). The answer is then sent back to the UI which transmits it to the user.

In a sense, we can see the UI as the front-end of the chatbot and the dialog management component as the back-end. The NLU component could be considered either as part of the back-end as well, or as a middle-end, depending on the definition of "back-end".

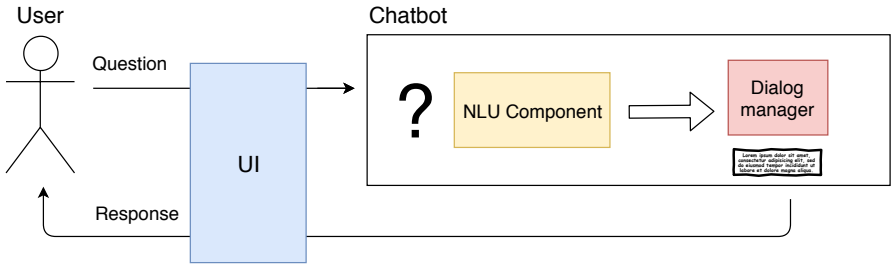


Figure 2.2: General architecture of modern chatbots

The broad functioning of each of those components is discussed in the next subsections.

2.3.2 User Interface

The User Interface is the component of the chatbot that allows the communication between the user and the core of the chatbot. One might argue that it is not really a component of the chatbot since it is not part of its core, but not having such an interface makes the chatbot impossible to use.

As we explained before, we can classify chatbots based on their UI. It seems that there are two main types of chatbots: text-based ones and voice-based ones, each with a different kind of interface. This master thesis is more interested in the study of first kind of chatbots than the latter one: except for this section, the remainder of this dissertation will be about text-based chatbots.

Text-based interface

Text-based chatbots employ a *Graphical User Interface* (GUI) to interact with their users. This GUI is always a simple chat messaging application, which both the user and chatbot can use to send text messages to and read messages from. Some chatbots employ an existing GUI, usually a messaging system from a social network; others use a custom-made GUI. (Such a custom-made GUI is shown on figure 6.6.) Some systems even support being used with both.

While very simple text messaging systems are sufficient to communicate with a chatbot, it is quite frequent to find additional features in GUIs. Those features are usually related to sending and receiving messages that are not texts, such as

- images or GIFs (animated images)
- hypertext links
- audio recordings
- files in different formats (PDF, zip, etc.)

Of course, in order to send or receive a message that is not text, a chatbot must be capable of assimilating and understanding it. Adding support for different kinds of messages can be quite tedious for the designer (or for the creators of the framework in use), which explains why most chatbots do not support sending and/or receiving complex types of messages. Indeed, some formats are harder to understand upon reception than sending, and vice-versa. For example, sending hypertext links can be done directly by sending the URL as plain text, while taking such a link into account requires the chatbot to follow it and parse the associated web page.

It is noteworthy to mention that a designer can prevent the user from sending some formats of message if and only if their chatbot makes use of a custom GUI. Otherwise, the chatbot should be set to tell the user it does not support such messages.

It is also frequent to encounter chatbots able to send the user a list of (clickable) answers to a question, ranging from a simple "Yes" or "No" to more complex choices. This can make the user's utterances more predictable, and consequently decrease the likelihood that the NLU component fails to understand it.

Voice-based interface

Instead of a GUI, voice-based chatbots use a *Voice User Interface* (VUI), also called *Sound User Interface* or *Auditory User Interface*. Namely, the chatbot interacts with the user using a microphone and a speaker, and makes use of voice recognition to turn utterances in text exploitable by the subsequent components. As opposed to text interfaces, the only kind of messages that can be sent and received are audio messages.

Of course, some chatbots are able to utilize both those interfaces. Some are even designed to seamlessly switch from one to the other, sometimes even during the same conversation. This is for example the case of *Google Assistant* on smartphones, which can be talked to by voice, but writes each utterance at the same time in a text chat which the user can interact with at any point of the dialog. Strictly speaking, this chatbot handles both interfaces at the same time.

Other interfaces

Most types of interfaces are not suitable for chatbots, as interactions in natural language are intrinsically complex. This complexity is not attainable using touch, olfactory or gustatory interfaces for example. That being said, we can mention that a chatbot and its user could theoretically interact through sign language (i.e. the natural language spoken by deaf people), using a camera and a screen with a physical representation of the chatbot. This would certainly be an interesting research topic, but it hasn't been explored yet.

2.3.3 Natural Language Understanding component

Intuitively, the *Natural Language Understanding (NLU) component* is in charge of understanding what the user said. Its purpose is thus to translate the messages of the user, expressed in natural language, into a more formal representation that is exploitable for later computations. This problem of transcribing information from a natural language to a machine-readable representation is very

common in Natural Language Processing (NLP). In fact, NLU is a subfield of NLP.

The problems this component addresses are usually divided in two disjoint tasks: "intent classification" and "entity recognition".

Vocabulary

Before getting deeper into their definition, we will introduce common NLU vocabulary that will be useful throughout this text.

Utterance space Also called the *message space*, it is simply the abstract space in which all user messages exist. Intuitively, we would place similar messages close to each other in such a space. However, this requires to define a notion of distance over that space, which is not especially necessary in the general case. Noticeably, unless the message size is bounded, this space is infinite.

Intent An intent is related to the intention the user has when making an utterance, i.e. the reason they are uttering this message. In the case of a conversation with a chatbot, this meaning is most of the time what they want the chatbot to do. More formally, each intent is a class of messages, that is to say a (potentially infinite) set of messages whose meaning is similar. Those classes are non-overlapping, meaning the intents make up a partition of the utterance space. Hence, each message is labeled as one and only one intent.

For example, the messages *Hi!* and *Hello* both could be classified as an intent labeled `greeting`.

Entity An entity refers to a noticeable part of an utterance, which contains specific information. It identifies a concept we will call the entity label, and has a value which can be exploited in the back-end of the chatbot. There can be any number of entities within a user utterance.

For instance, in the utterance *I'd like a hot coffee please*, *coffee* could be recognized as the value of an entity labeled `drink` and *hot* as the value of a `temperature` entity.

Typical modern approach

The intuitive task of understanding the utterances of the user is not only hard to solve, but also very hard to express formally. Therefore, the NLU component actually tries to solve simpler problems, whose solutions will allow to approximate understanding the utterance. As we said, the usual approach is to solve two disjoint problems: "intent classification" and "entity recognition".

Intent classification This problem corresponds to the task of classifying each user utterance as a certain intent, hence the name *intent classification*. In other words, the NLU model receives a user message in natural language as its input and returns a labeling intent as its output.

It is worth mentioning that this component does not receive any feedback from the dialog manager. It thus has no access to the context of the conversation, past utterances or a history of the errors it made. In other words, the NLU model present in chatbots is usually completely stateless. If this model performs well enough, it is not problematic for conversation management. Nonetheless, having a stateful NLU model for this task would certainly improve its comprehension at the cost of a larger amount of training data and a more complex design.

In the field of machine learning, it thus corresponds to a supervised multi-class classification problem. As any supervised problem, a set of examples provided by the designer to train the NLU model is required. In the context at hand, this dataset contains examples of utterances labeled with the correct intent. Of course, the usual pitfalls apply here as well, namely the potential overfitting of the produced model. As we discuss in chapter 7, creating a training set that avoids this is not as easy as might be expected.

As explained in the definition of intents, non-overlapping intents make up a partitioning of the message space. Since a chatbot has a finite dialog scope, a limited number of intents are specified. Training an NLU model with those classes only leads to misclassifications since it will also label utterances that are out-of-scope.

To avoid that, it is customary to add a partition corresponding to out-of-scope and not understood utterances. Another solution is for the NLU model to consider an utterance as out-of-scope or not understood when its confidence level is too low. This confidence level corresponds here to a probability estimated by the NLU model that its classification is correct. This latter alternative is most popular amongst modern chatbot systems. The confidence level is also often returned as an output of the NLU component.

This kind of problem has been researched for a long time. Consequently, a large number of algorithms exist to solve it [13, 14, 15].

Traditionally, models such as Hidden Markov Models [16] or methods based on TF-IDF [17] were used, but more recent methods such as deep learning or long-short term memory networks have also been used. The simplest but least efficient model would be simple keyword matching. Of course, those algorithms have different qualities and flaws. However, the point of this dissertation is not to describe or rank them. We can note that some algorithms will use a particular utterance space with a clearly defined notion of distance, while other ones will not need such precision. An example of the first would be TF-IDF; an example of the latter would rather be a deep learning algorithm.

Entity recognition The second task is to locate and label each entity present in an utterance, hence the name *entity recognition*. Specifically, the NLU component is supposed to find which parts of the utterance correspond to an entity, correctly label them and extract the entity value. This task can thus be seen either as a whole or as a three-pass task, first detecting the entities, then labeling them and finally extracting their values. Note that as for the intent classification and for the same reasons, the model here is stateless.

We should note that some systems consider the entity value to simply be the detected part of the utterance, making the value extraction trivial. Other systems support lists of synonyms for each entity value. Such systems then often solve the problem of entity detection and recognition by requiring a perfect match to detect them. Finally, more complex systems have more in common with intent classification: they require a training set which contains examples of entity values correctly labeled. This last type of system can sometimes return a confidence level as well. To conclude, a very large number of approaches to this problem, as well as algorithms to address them, exists.

Note also that this task is usually completely disjoint from the first one: intent classification does not have an influence on entity labeling. This being said, some systems cross-check the answers of both tasks and recompute their outputs based on the answer to the other task.

Limitations of this approach

It is undeniable that dividing this problem into two tasks allows to make the problem easier to solve. Additionally, it seems this approach is taken by virtually all modern chatbot systems, which may be explained by its efficiency: representing a user utterance as an intent and a set of entities is in most cases enough to make the required back-end of the chatbot. However, simplifying the problem like this induces a certain amount of limitations.

The biggest limitation is certainly the lack of information that the back-end has access to. Logically, having more information is always useful as the dialog manager can take decisions with a larger knowledge at its disposal. Several NLP algorithms, such as Sentiment Analysis or Phrase Chunking can be used to acquire additional information.

Another limitation is the fact that it is impossible for a user to send a message with two intents. Indeed, the NLU model works on a per-message basis, meaning it will classify each message as having a certain intent. However, it sometimes happens that the user is asking for two things in one message. For instance, a user could utter *I would like a black coffee and a doughnut*, while the chatbot would only support ordering one thing at a time. In this example, the chatbot would end up ignoring one of the entities in the best case, or completely misclassifying the utterance and answering something incorrect in the worst case. This limitation could theoretically be solved using a Phrase Chunking algorithm [18], but was never investigated in literature and would complexify and slow down the design process.

Finally, as we pointed out, having both models for the intent classification and the entity recognition be stateless algorithms may decrease the maximum attainable performance. The fact that it drastically reduces the complexity of design explains this choice.

2.3.4 Dialog management component

As we said, the dialog management component can be considered to be the back-end of a chatbot, as it is in charge of choosing what to answer and which actions to perform given the user messages. Its inputs are the results of the NLU component for the latest message, but can also include an internal memory that would contain data such as a history of the detected intents or the assumed current context of the dialog. This memory handling varies a lot depending on the chatbot system.

As one can imagine, the problem of creating the answer and choosing the actions to take (if any) can be tackled in a lot of different ways. Some systems, such as plan-based dialog managers [19, 20], confront the problem as is. They are thus divided into a back-end component making the choices and a Natural Language Generation (NLG) component. However, the back-end of those systems usually needs more input than what the NLU component can give them, and thus require a more complex NLU component. Moreover, such systems often require complex components whose current performance is not satisfying enough to be used in the industry yet.

Most recent systems usually simplify the problem in the following way: the answers are recorded in advance by the designer, and the dialog management component then chooses when to emit them. The same reasoning is done for actions: supported actions would be prepared by the designer, the dialog manager being left to decide when to perform them. In such systems, choosing which action to take or what to say is thus very similar from a designer viewpoint, hence we will

only talk about choosing the answer in the rest of this paper, without a loss of generality.

Once again, this problem can be tackled in a large number of different ways. However, the different approaches can generally be categorized as five different types [1, 21]. Each approach translates to a different architecture for the component, with its qualities and flaws. We describe those approaches hereafter, in their most fundamental form. Of course, in practice, changes can be applied to each approach. We will end this section with the table 2.1 summarizing them.

Finite State Machine architecture

The first and oldest approach to the problem of choosing which answer to emit is to use a Finite State Machine (FSM) as a dialog manager. On reception of a user message, the FSM changes state depending on the detected intent and entity values. An answer is associated to each state, and is emitted when this state is visited. Such a dialog manager is illustrated on figure 2.3, where each state is labeled with its associated question.

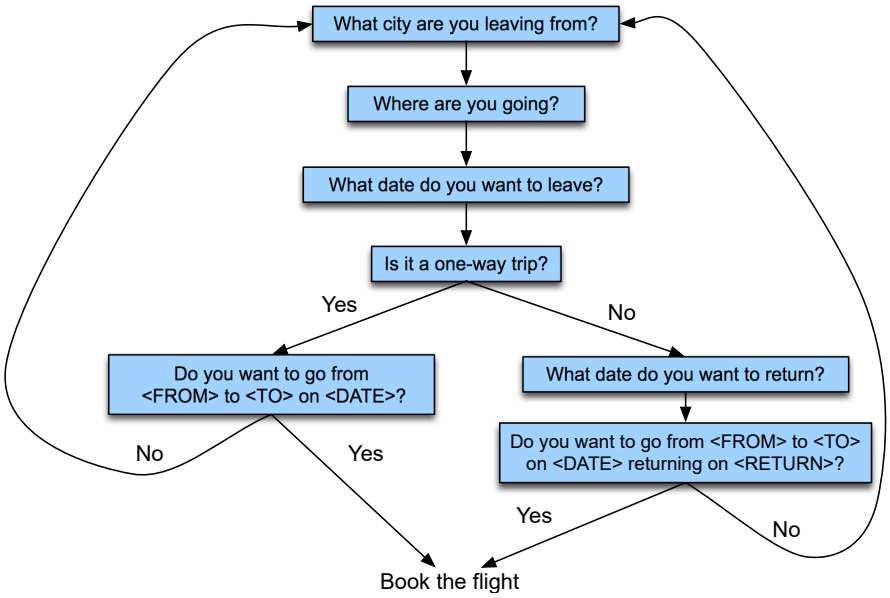


Figure 2.3: Simple example of an FSM dialog manager from [1]

This approach has the advantage of being easy to understand to both the designer and the user. However, it can be expected that the produced chatbot will feel very robotic as its answers will always be the same in the same context. This can be mitigated by defining a list of answers for each state and choosing one of them randomly when in that state.

Another advantage is that it is quite easy and fast to design a small chatbot using this approach. The FSM can indeed be designed on paper and then easily turned into a concrete system. This design process can be sped up further by using a framework which supports creating FSM and turning them automatically in the concrete system [22, 23]. Moreover, designing a chatbot this way is within reach of non-technical people.

However, if there is an error in the design and the chatbot consequently behaves in an unintended manner, it is immediately clear to the user. On the other hand, figuring out why such a behavior occurred is usually way harder. Precisely, it can be hard to know what decision(s) lead to this

situation, partly due to the fact that conversations can be intricate, as would be the corresponding FSM.

Plus, if the designer wants to expand the conversation scope, the only possibility they have is to increase the number of states. Consequently, the number of transitions grows, at an exponential rate in the worst case. Thence, such a chatbot quickly becomes impossible to maintain and hard to fully appreciate. Generally speaking, this approach is not suitable for chatbots intended to support more than a handful of intents. Adding memory capabilities to this approach would make it more tractable, but the mechanism would not correspond to an FSM anymore.

Finally, such chatbots are usually too restricted to be usable, even for quite simple interactions. Indeed, their architecture forces the designer to make it expect a type of message at each state, even though the user might want to answer something different. This behavior makes such chatbots quite tiresome and slow to interact with, since the user does not have the choice to guide the conversation where they want. This issue can be reduced a bit by adding an epsilon transition going from each state to a "reset" state. The generated chatbot then works better but still is quite tedious to use.

Conditional blocks approach

This approach considers dialog management as a sequential list of conditional blocks. A block is visited when and only when its associated condition is fulfilled. Such conditions take into account the outputs of the NLU component for the latest utterance, i.e. the detected intent and entities, and possibly the related confidence levels. Once entered, the block can either directly generate an answer, or jump to a subsequence of conditional blocks. There is actually a bijective correspondence between this concept and if-else statements in programming languages, which means it directly translates to a hierarchy of such statements.

As the conditions triggering a block are solely based on the latest user utterance, a chatbot which takes this approach is very shortsighted. In other words, it does not have any notion of context and can only reason on what it was just said, and is completely stateless. Indeed, upon message reception, its dialog manager tests the whole list of blocks in sequence until it finds one whose triggering condition is true. If the user sends the same message again (or a message with the same intent), the chatbot will answer in the exact same way. The types of behavior this allows are thus quite restricted and interacting with the produced chatbot will be very cumbersome.

In practice, support for recording and reusing some values later in conversation is added to such systems to extend its capabilities, making the component stateful. Such values include for example a history of the detected intents and entity values. As one can expect, this can greatly improve the usability and quality of the produced chatbot. We should note that this approach is more powerful than the FSM approach only when we add the possibility to record some previous values.

This approach has the big advantage of being able to define very complex conversational flows (with lots of sub-sequences to check), especially if memory capabilities have been added. However, as for the FSM approach, it is quite hard for the designer to identify where a misbehavior comes from. Nevertheless, this flaw is less present here than with the FSM approach.

Moreover, this technique has the advantage of scaling up better than the FSM one: when the conversation scope grows, the designer only has to add corresponding conditions at the right place in the sequence. Nonetheless, correcting the potential errors in the design becomes increasingly

hard, once again because conversations are typically quite cluttered. Consequently, this kind of approach works quite well for small chatbots but is not really suitable for larger ones, because of maintainability and debugging issues.

Goal-based approach

With this approach, the chatbot designer defines a set of "conversation goals". Each goal is triggered by the detection of a certain intent and contains a certain number of "slots" to fill with a certain information. As long as some slots are empty, the chatbot will ask the user for this information. After all slots are filled, the goal is considered to be met, and the answer (or action) associated with it can be told the user.

It is customary to replace some placeholders in those answers by the value stored in a slot, even though this is not a feature of the most basic goal-based method. It is also usually possible to jump out of the current goal if the latest user message has an intent that can trigger a new goal. Since this architecture relies on slots being filled, it is also sometimes called a *frame-based* or *form-based* architecture.

Most of the time, the information to place in slots is collected as an entity value. For instance, if the answer associated to a goal greets the user, this goal would require the user's name in order to personalize the message. The chatbot would thus ask the user for their name, which would be detected as an entity by the NLU component and placed in the slot.

This approach has several advantages. First, it is quite easy to understand and to craft from the viewpoint of the designer. As a consequence, building a chatbot using such an approach can be quite fast.

Then, it is quite easy to add support for new intents: the designer would simply define the intent to the NLU model, add a new goal triggered by this intent and with the relevant slot(s) to fill, and record an associated answer. Those scalability abilities make this approach suited for large chatbots. Moreover, it is relatively easy for the designer to find faults in their design.

However, it does not permit complex conversational logic. Indeed, the dialog manager is only able to wait for the user to trigger a goal, then possibly ask them a certain number of questions in order to fill the slots and finally utter the corresponding answer, or forget the current goal if another goal was triggered. The simplest version of this approach typically erases slot values when a goal is reached, making it stateful only during the fulfillment of a goal. The produced chatbot will thus never remember the conversation it just had and will seem smart only during short interactions.

Markov Decision Process architecture

This approach is based on Bayes formula and probability theory. It uses Markov Decision Processes (MDPs) and Partially Observable MDPs (POMDPs) to model the behavior of the chatbot. This approach has a lot of similarities with the FSM approach. As a reminder, an MDP can be seen as an extension of an FSM: the main difference comes from the fact that the transitions are probabilistic and an reward the agent receives for taking an action. In this context, an MDP is indeed characterized by a set of states, a set of answers (or actions) to be emitted each associated with a state and a reward for emitting them.

Such a dialog manager will thus choose what to answer dynamically, typically in function of the latest detected intent and entity values, the NLU confidence levels and the current state of the dialog. The choice is made in order to maximize the success of the dialog, while minimizing the cost. This optimization approach shows its proximity to probability theory and is the reason why MDPs are used inside the component.

Indeed, given the reward function, a policy specifying which answer to emit given the current state. Bellman equation [24, 25] can be used to compute the expected cumulative reward and choose which answer to utter. An iterative algorithm such as Q-learning would allow to find the best policy.

It is clear that such a dialog manager will work well if and only if the reward function is well crafted. This reward function usually depends on the dialog scope. Therefore, the designer of the chatbot has to define it by themselves. This is a clear disadvantage of this approach, as it makes chatbot design much harder.

Since they are quite close, this approach and the FSM approach have similar disadvantages. An important disadvantage is thus that such a chatbot is hard to expand, since the designer needs to define more and more states and transitions. Having to define the associated reward for new states is an additional concern.

That being said, the stochastic behavior of the produced chatbot has the advantage of seeming more natural to the user, since there is no clear determinism behind the chatbot's utterances. This clearly is an advantage over the other approaches.

As a conclusion, generally speaking, this approach is very suited for small chatbots with a small conversation scope. In practice, this approach is not used very frequently because of the mathematical complexity behind it and the difficulty of defining a suitable reward function.

Machine learning approach

The machine learning (ML) approach is similar to the approach that is usually taken for the NLU component. The problem of choosing what to answer is seen as a classification task: the dialog manager has to predict the correct answer from a finite list of answers, given the prior history of messages (or more specifically their representation after the NLU component treated them).

A machine learning model can then be trained using a set of examples of conversations provided by the chatbot designer. Note that while this supervised learning approach is mostly used, reinforcement learning can also be encountered. Some systems even use "interactive learning", where the designer has a conversation with the chatbot, telling it whether it made the right decision[26]. As for the different sub-problems addressed by the NLU component, lots of different algorithms can be used to solve this problem.

This approach is fundamentally different from the ones presented above, since the designer does not specify static rules anymore but rather lets the model figure them out. Therefore, this approach has several advantages the other ones do not have.

First, it is easy and fast to design a chatbot that uses this approach, since the designer would just have to give a (potentially large) set of examples of conversations and let the model train on this. For the same reason, this approach scales very well: if the conversation scope grows, the

designer just has to define new intents and entities, and retrain the model with additional examples.

Of course, this approach also has its disadvantages. The biggest one certainly is the fact that it is hard for a designer to know what the chatbot will do in specific cases, and especially in edge cases. Understanding and fixing an unintended behavior is consequently difficult.

It is also not trivial to know if the training set required to initiate the process of machine learning is diversified enough to create a model that generalizes well. Assessment techniques are often required to be able to realize whether it is the case or not. As we will see in chapter 7, it is not easy to assess the quality of the dialog management unit alone.

Hybrid approaches

We could add a sixth approach to our list, which contains all the systems which are a mix of previous ones. This kind of approach is usually taken to try and alleviate the disadvantages of the different methods chosen, while trying not to diminish their advantages. Of course, the resulting advantages and flaws depend on the mixed approaches and the way they are combined. Understandably, hybrid approaches are the most frequent method that is used in modern chatbot frameworks.

Also, we should note that the creators of a chatbot framework generally make some changes and add features to their system, rather than blindly using one of the approaches as described above.

Approaches summary

The following table summarizes the presented approaches. The term "behavioral control" relates to the possibility for the designer to control how the chatbot behaves given the situation. "Usability" refers to whether the produced chatbot is cumbersome to interact with.

Characteristic	FSM	Conditional blocks	Goal-based	MDP	ML
Design ease	+	+	+	- -	++
Debugging ease	-	-	+	-	-
Scalability	- -	-	+	- -	++
Capabilities	+	-	-	+	+
Behavioral control	++	++	+	-	- -
Usability (for users)	- -	- -	+	+	+

Table 2.1: Summary of the different dialog management approaches

Chapter 3

Linguistic challenges with chatbots

This chapter addresses the principal challenges and difficulties that chatbot designers encounter when producing a chatbot meant to behave naturally. As can be expected, all those challenges are related to linguistics.

3.1 Properties of human conversation

Before being able to design a well-crafted chatbot, it is important to study and highlight noteworthy properties of human-to-human dialog. Understanding the structure and aspects that play important roles in conversations indeed allows us to make a chatbot that feels more human-like and is thus more natural to interact with for the average user.

Nevertheless, we should note that the purpose of the chatbots we make is not to fool users into thinking they are talking to a human being. The point of making the chatbot more human-like is to have a system that is easy and natural to use, as average people are already familiar with talking to human beings. If we want to make a chatbot more natural to use, we need to understand what makes a conversation awkward or unnatural.

Let us thus list several important properties of human dialog, i.e. a conversation between two humans. We will then see how these properties can affect the way we make chatbots. We should point out that there are other properties which might be worth noticing, but we list here only those that seem most relevant to chatbot design.

3.1.1 Turn-taking

Definition

A first obvious characteristic of human conversation is turn-taking: one person talks, then the other one takes the floor, then the first ones takes their turn back, and so on. Of course, a conversation will feel more natural and pleasant to a speaker if their interlocutor speaks at the "right" time. We should mention that it seems the turn-taking rules depend on the culture and language of the interlocutors.

There is a large literature about the problem of deciding when a conversational agent should take the floor, which is studied in the field of Conversational Analysis (CA) [27]. While this problem might seem trivial, as we do it in every conversation we have, it is actually rather complex. Determining whether this is the right time to take your turn indeed requires analysis of the meaning, intonation, tone, time and many more variables. In speech, silences can even be significant (e.g. a long silence could indicate a refusal to answer a stigmatized question). It even happens in normal conversation that interlocutors speak over each other, meaning turns can overlap. It appears after measurement that those overlap are quite short: about 5% of the conversation consists of overlapping turns [27].

Obviously, this kind of problem is much more difficult when the amount of speakers is large. Nonetheless, as we said earlier, we will restrain ourselves in this paper to dialog, i.e. conversations between two actors.

For text conversations however, some aspects of conversations are limited, making it easier to deal with. For example, it is not possible to have overlapping turns in such context. Moreover, some variables such as tone and intonation are simply inaccessible to both interlocutors.

Implications on chatbot design

Turn-taking gives designers a structure that chatbots should always follow: it should let the user speak for a certain time, and then take its turn. As we are here interested in text-based chatbots, this difficulty is less important, as the user interacts with the chatbot using text messages which can be considered indivisible. There are thus two situations during which the chatbot could decide to take the floor, either after the user sent a message, or after a certain time elapsed since the last message was sent (whichever speaker sent it).

The usual approach is to make the chatbot take the floor after each user message only. Some designers also make the chatbot utter something if its last message was unanswered for a certain time. Those two solutions proved to be sufficient in practice. This being said, the problem stays quite difficult to solve in the general case, e.g. with voice-based chatbots.

3.1.2 Initiative

Definition

Initiative in a conversation is defined as follows: a speaker has the initiative when they are the interlocutor who is leading the conversation. This is thus a question of who is controlling the conversation at one point. Of course, which speaker has the initiative can change during the dialog.

Nonetheless, a speaker doesn't have the initiative when they are merely speaking. Rather, it depends on the information they are uttering and even more precisely on this information in relationship to the current state of the conversation. A speaker who has the initiative would typically change the topic of the dialog, start a sequence of question-answer turns or decide where the conversation should go. Initiative is thus related to turn-taking, as starting to speak when a conversation seems to be over (or could naturally stop there) implies taking the initiative.

Expectedly, a speaker who never takes initiative in a conversation will seem either shy or robotic

(depending on the length of their answers). Note that having a shy interlocutor is not especially a problem if the other speaker knows what they want from them.

Implications on chatbot design

Taking initiative into account, several approaches exist, characterizing different types of chatbots.

System-initiative approach The easiest approach would produce a chatbot that keeps the initiative for the whole conversation. Such chatbots are called *system-initiative* or *single-initiative* systems. Specifically, the chatbot asks the user questions and only accepts valid answers. Consequently, the user is not allowed to ask questions or answer anything that isn't a direct answer to the chatbot's query. Often, it will ignore, refuse or even misinterpret any unexpected utterance.

If needed, this behavior can be soften a bit by adding the possibility for the user to start the conversation over or to correct their last utterance anywhere in the conversation, using special "command" sentences called *universal commands* [28]. Obviously, this kind of interactions is not very natural. This kind of chatbot can easily be implemented using an FSM or a conditional block dialog manager.

As we could imagine, interactions with such a chatbot will seem very unnatural to the average person, making it tedious to use. Restricting the possibilities of the user to just answering the current question or resetting the system indeed doesn't feel like a normal dialog. Note that this approach has been used a lot in the past thirty years, notably with automatic phone operators (cf. appendix A).

User-initiative approach Another easy approach is to have the chatbot never take the initiative, making it a "reactive chatbot" (by opposition to a "proactive chatbot"). We could make a parallel with a question-answering machine or current capabilities of search engines which are often able to answer simple questions. Again, interactions with such a chatbot doesn't seem natural, as the user has to keep asking questions for the conversation to continue.

Mixed-initiative approach The last approach, enabling *mixed-initiative* systems, is the most frequently used nowadays, as it gives the most natural results of the three when well executed. The term "modern chatbot" often refers to those employing such an approach. Such chatbots allow the conversational initiative to shift between the user and the chatbot itself. The shift is often allowed only at some definite points of the dialog.

Working with a goal-based dialog manager allows to design such systems easily, since it will ask the user questions in order to fill slots, but lets them control the conversation by allowing them to choose which goal is triggered and by giving more information than they were asked, this information then filling other slots. Therefore, *mixed-initiative* systems is sometimes regarded as synonymous to *frame-based* or *form-based* chatbots.

3.1.3 Speech acts

Definition

Each turn in a conversation, and thus each message, can be seen as an action performed by the speaker. Indeed, an interlocutor always utters something with an objective in mind, otherwise they wouldn't say anything. These actions are usually called *speech acts*. We can see a speech act as a category of objectives behind uttering a sentence. This way of interpreting conversations was theorized by the philosopher Wittgenstein [29] and refined later on by Austin [30].

Different classifications exist. For example, Searle [31] proposed the following (overlapping) categories:

Assertives The speaker makes a statement about the state of the world.

Directives The speaker requests their interlocutor to perform an action.

Commissives The speaker informs their interlocutor about their willingness or plans to do something in the future.

Expressives The speaker communicates their psychological state about some facts or events.

Declarations The speaker changes the state of the world by means of the utterance (e.g. *You're fired* or *I pronounce you husband and wife*).

Obviously, those categories can be refined into even smaller categories, which could in turn be divided, and so forth, leading to a complete hierarchy of speech acts. Several different classifications and hierarchies have been proposed. As they are quite arbitrary, it is not clear whether one of them is inherently better than the other ones.

We should mention that there has been research about making speech act categories specifically for dialog systems. Those are then called *dialog acts* or *conversation moves*. The differences with speech acts is not very significant and the same remarks hold [32, 33].

Implications on chatbot design

Determining which kind of utterance the user said can be a very useful piece of knowledge when trying to answer. If we decide to use precise categories rather than broad ones as presented above, those categories are very reminiscent of the concept of intents we presented in subsection 2.3.3. As a reminder, the usual attitude is to consider the user always has an objective behind uttering a message, and the supported objectives, called intents, are predefined by the designer. Knowing what the intent of an utterance is is generally enough information to have a corresponding answer predefined. We could see those intents as really precise categories of speech acts.

We should mention that having access to the underlying speech acts hierarchy would allow the designer to specify a conversation logic that takes into account the broader categories. This is however not necessary to create a functional chatbot.

Note that this approach also solves the problem of conversation implicature, that is the fact that speakers often utter something, while the real intent they have in mind is not really what they said

but rather can be implied from it and a certain number of prior assumptions. For example, if a chatbot tells the user *We have 7 different types of coffees*, the speaker can assume it means *7 types of coffees and not more* which is actually not what the utterance states. Plus, the objective behind this utterance could be along the lines of *Tell me which type of coffee you want from the 7 and only 7 types we have*, which is once again not what is stated. Conversation implicature is a topic that is heavily described in the literature, but it is not a problem with the approach explained here.

3.1.4 Grounding

Definition

A very important thing human beings do when talking to each other is establishing common grounds, i.e. a set of things that are believed and accepted by both interlocutors. Therefore, speakers must "ground" each other's utterances. In other words, they must tell their interlocutor they heard, understood and accept their last utterance. Norman [34] calls this the *principle of closure* and Clark [35] phrases it this way: *Agents performing an action require evidence, sufficient for current purposes, that they have succeeded in performing it.*

We should note that grounding is important in any kind of interface: it is essential to show the user their actions have an influence on whatever system they are interacting with.

It is possible to ground utterances in different ways, ranking from weak to strong. In human conversations, a very weak grounding would simply correspond to the interlocutor visibly showing continued attention to their interlocutor. A stronger way could be to acknowledge their utterance by nodding or saying a few words such as *uh-huh* or *yeah*. The strongest grounding technique would be to repeat or rephrase part of the utterance verbatim, for example by asking the interlocutor whether their statement was well understood. As you can imagine, intermediate grounding methods exist. Note that a lot of different mechanisms may be considered as grounding, and that even for text-based conversations.

Here are three examples of grounding following the same utterance, from weak to strong. The words related to grounding are displayed in italics.

> — I'd like to have a small coffee with two sugars, please.
> — *Ok,* is it to take away?

> — I'd like to have a small coffee with two sugars, please.
> — *And* do you want a cookie with *your coffee*?

> — I'd like to have a small coffee with two sugars, please.
> — *A coffee with two sugars, is that right?*

Implications on chatbot design

As can be expected, not grounding, grounding too often or always using very strong types of grounding can make a conversation seem awkward. Therefore, it is arguably one of the most important things to get right when creating a chatbot. Indeed, the two simplest solutions would be to never ground, or to ground every utterance very strongly. As you can imagine, the first solution would produce a chatbot which doesn't seem to fully understand the conversation (and likely sometimes doesn't understand indeed); the second one would end up with a chatbot which sounds extremely doubtful and too cautious with its users.

Designers often decide to make the chatbot strongly ground only when the confidence level in the understanding is very low. On the other hand, choosing when and how to ground more weakly is much more tricky. The typical approach is to try and predict the different conversation flows and predefine the answers of the chatbot including a few words that weakly ground the latest utterance of the user. Clearly, this solution is not really satisfying, since it isn't automatically decided, but it is sufficient in practice. However, as a consequence, it may arise that an utterance is grounded in a way that sounds strange to the user. Moreover, this solution doesn't scale well: when the conversation scope grows, it gets increasingly harder for the designer to predict the possible conversation flows.

It is also worth noting that automatic strong grounding is usually not implemented in popular frameworks such as *Google Dialogflow* or *IBM Watson Assistant*. The designer thus often adds a predefined confirmation answer for the chatbot at some points of the dialog. This is one of the most prominent reasons why modern chatbots, while commonly able to understand most messages that are in their dialog scope, still don't feel as natural as we would want.

Chapter 4

Tools to make chatbots

This chapter presents different useful tools that can be used to help crafting a chatbot. As explained before, the point of this master thesis is to inspect tools that are currently popular in the industry, rather than brand-new, inexperienced technologies. We will specifically examine the tools that we used to build our chatbot.

4.1 NLU dataset generator

As we explained in subsection 2.3.3, the NLU component requires a dataset to train its intent classification model. This dataset would contain examples of messages that the user might utter, labelled with the corresponding intent. In order for this model to perform well, a dataset with enough variety is required. Otherwise, the model is very likely to overfit on the provided examples, which means correct user utterances that are not similar to provided examples could not be recognized accurately.

As the reader can imagine, it is quite tedious for the designer to create such a dataset, since many varied examples have to be defined, even more so when large amounts of intents are supported. This very problem occurred to us when we created our chatbot.

To speed up this process, chatbot designers commonly use a dataset generator. Such a tool allows them to only provide templates of examples which are then used by the tool to generate the dataset. As NLU algorithms usually expect a dataset formatted in a certain way, such tools can also handle the formatting of this data.

A lot of tools exist for that purpose, each with its own capabilities and intended to be used with a certain NLU system. Amongst them, we can cite Chatito [36], Chatette [37], Chatl [38], Expando [39], Tracery [40]. All those tools are open source projects. We can also note that the three first ones use a similar syntax for writing the templates, allowing for a certain interoperability between them.

To make the training dataset for our chatbot, we decided to use *Chatette* for two reasons: it is intended to create large datasets as we needed, and it was developed by one of the authors of the present dissertation meaning we could easily add required features.

Depending on the framework used, those tools can also be used to make such a training set for

the entity recognition task.

4.2 Standalone NLU components

Several pieces of software exist to only fulfill the role of the NLU component (cf. subsection 2.3.3), each using different NLP algorithms (not described) to fulfill their task. Those components are standalone in the sense no dialog management unit is provided to work along with it.

We can notably describe the following projects:

Product	Pricing	Open source	Performance
Facebook's *Wit.AI* [41]	Free	No	+
Microsoft's *LUIS* [42]	Free (limited in time)	No	++
Snips NLU [43]	Free	Yes	++
Ambiverse NLU [44]	Free	Yes	-

The performance measurements in this table roughly reflect the results of performance studies you can find in subsection 4.3.3. Note that those products being very often updated, those results may vary in time by quite a large amount. At the time of writing, *LUIS* and *Snips* NLU seem to be the most efficient ones (in terms of precision and recall).

As those tools only make up the NLU component of the chatbot system, another piece of software will be required to take the role of the dialog manager. However, it seems quite hard to find a standalone dialog manager: the creators of the tools cited above appear to expect chatbot designers to create their own. Custom dialog managers are indeed frequently encountered in practice, but are usually not complex enough to support complex conversations flows and large dialog scopes.

Of course, there also exist pieces of software that bundle both the NLU component and the dialog manager together (and sometimes some additional components), called "chatbot frameworks". They are described in the next section. Note that for open-source frameworks, the products cited above could be used as a replacement for their NLU component.

The chatbot we built up for this master thesis being quite large, we decided to use a framework rather than one of the standalone NLU components shown here.

4.3 Chatbot frameworks

4.3.1 Definition

A framework is a software infrastructure whose objective is to facilitate and speed up a developer's task. It gives them a ready-to-use architecture and components to solve common obstacles encountered in the type of applications it is related to. In other words, a framework allows developers not to have to start from scratch at each new project.

There are lots of different frameworks intended to create different types of software. Conceiv-

ably, a certain number exist specifically to build chatbots, each having its characteristics, tackling problems differently and offering different useful features. Those frameworks typically follow the general architecture we described in section 2.3.

The next subsection will enumerate the most prominent chatbot frameworks used in the industry and available for public use. Naturally, this list cannot be exhaustive, since there are hundreds of frameworks and pieces of software that have to do with chatbot design.

4.3.2 Available frameworks

Frameworks are very different from each other. After reviewing different ones, it seems however that the most conspicuous characteristic to categorize them is whether they are closed or open source.

Usually, closed source frameworks are meant to be used online and run on a server owned by the company which the framework belongs to. On the contrary, open source frameworks can usually be used on a local machine, and are typically developed to be lightweight enough to execute on slow hardware. Of course, it is also possible to put the produced chatbot on a privately owned server and make it communicate with the user through an internet connection. This means closed source frameworks cannot be used without access to an internet connection, while open source ones usually can. This is true both when designing a chatbot and when interacting with the produced chatbot. Open source frameworks are thus a better solution if the chatbot is meant to be used on a local machine or when data privacy is critical.

It also seems that closed source frameworks receive more funding, which means more of them are usable as a professional solution. It is important to acknowledge that several open source frameworks can also be used this way as well.

Closed source

Multiple different closed source solutions are available to the public. Those solutions usually use a "pay per request" system, even though they often also propose a free plan with a few limitations. The most noticeable solutions are the following ones:

Product	Price	NLU performance[1]
Google's *Dialogflow* [45]	Free	++
Amazon's *Lex* [46]	Free for 1 year	+
IBM's *Watson Assistant* [47]	Free	++

Chatbots made using these solutions can only be deployed onto the respective clouds of the companies that own the frameworks.

For performance and price reasons, we decided not to use *Lex* for creating our chatbot. We analyze *IBM Watson Assistant* and *Google Dialogflow* in more detail respectively in sections 4.4 and 4.5.

[1]Rough ranking based on studies referenced in subsection 4.3.3

Open-source

Several open-source solutions exist as well. Here is a non-exhaustive list of solutions that seem effective and are all available for free:

Product	NLU performance	Dialog management approach
Rasa suite (core + NLU)	++	Machine learning
Botkit	+	Conditional boxes
Botpress	-	Boosted FSM approach
DeepPavlov	+	Varies

It is worth noting that *Rasa suite* is composed of two Python packages, one for the NLU component (*Rasa NLU*) and one for the dialog management unit (*Rasa Core*). This allows to easily replace the NLU component by a standalone one for example.

Botkit is meant to be used to design extremely simple chatbots. On the other hand, *DeepPavlov* is meant to design lots of different NLP systems and not only chatbots. *Botpress* is a less well-known, younger project that seems to incorporate a lot of good ideas. Note that this last project does not seem to use the architecture we presented in section 2.3.

The *Rasa suite* seems to be the best choice among those solutions. Since another master thesis is being made using it, we decided not to use it for our own chatbot.

4.3.3 NLU performance studies

To rank the performances of the different frameworks and standalone NLU components we presented above, we used several performance studies [48, 49, 50, 51]. As we said, each project being often updated and improved, the results of those studies may vary as time elapses.

From those studies, it seems that *IBM Watson Assistant*, *Google Dialogflow* and *Snips NLU* have the best performance regarding the intent classification task. *Rasa NLU* is apparently less often analyzed, but seems to have a performance just below the three best ones.

Obviously, as the performances of most frameworks have a precision and a recall that are both above 95%, the difference is quite insignificant. However, having an NLU component that understands what the user says more often is obviously better.

Anyway, choosing what framework to use should not only be based on the NLU performance. The usability of the framework, its price and the possibilities allowed by its dialog manager are important aspects to consider as well.

4.4 IBM's Watson Assistant framework

4.4.1 Presentation and description

In this section, we will present and delve more deeply into the technical details of *IBM*'s system. We can first make a quick presentation of *Watson Assistant*.

The name *Watson* actually refers to Thomas J. Watson, a past chairman and CEO of *IBM*. This name is very often associated to a well-known appearance of an AI made by *IBM* in the TV show *Jeopardy!* a few decades ago. Since then, technicians at *IBM* have improved the abilities of their AI related projects and started new projects. All those projects are grouped under the name *Watson*. It thus does not identify a conversational AI anymore, but a bundle of AI related projects.

The framework for developing chatbots made by *IBM* is actually named *Watson Assistant* and used to be named *Watson Conversation*. Note however that *Watson Assistant* is often referred to as *Watson* by the people who use it.

Improvements to NLU component

Technically, we already mentioned that the NLU component present in *Watson Assistant* is extremely efficient. From our experience, it seems to be the best one in situations with very few training data. Being closed source, we cannot know how its creators managed to achieve such a good performance in such cases. We could assume that they use some kind of automatic generator of example variations, and/or a very good NLU model, but those are just hypotheses.

Improvements to dialog management

About the dialog management, *Watson Assistant* uses an improved conditional blocks approach. Its specific features are the following ones.

Slots and value placeholders First of all, the designer has the possibility to store the values of some entities, ask their value automatically to the user and fetch back the stored information to use it in conditions or in answers. This allows to do things that are very close to what a goal-based approach does, and more since there are conditions. Hence, the registries which values are stored in are called "slots". As a consequence, we could say its approach is a mix between conditional blocks and goal-based techniques.

However, those "slots" also allow to store values for the whole conversation and reuse it at any moment of the dialog. A common example would be to ask the user for their name and store their answer. The user's name would then be reused to personalize messages (e.g. "Welcome back, John!"). Obviously, in order to do this, the dialog manager must be able to parse the prepared answers and replace the placeholders by the right slot values. This can be considered as another additional feature.

List of answers To make the chatbot appear less deterministic, it is possible to give a list of answers for each conditional block. The dialog manager will then choose at random which answer to utter. It is also possible to ask it to sequentially loop over those answers each time the conditional block is visited.

Context and jumps Then, *Watson Assistant* includes a certain notion of context: when the dialog manager has handled a message, it does not get back to its former state, as in the basic conditional blocks approach. Rather, it stays at the same level in the hierarchy of conditions, allowing for different dialog situations to be treated using levels of conditional blocks. The blocks are still visited in a sequential manner on the same level.

As the astute reader will have understood, another mechanism must be added in order to be able to get out of a level of blocks. This is possible thanks to a feature called "jumps": after visiting a conditional block, the designer can ask the dialog manager to jump to another level in the block hierarchy. This feature allows to design even more complex conversational flows than the basic conditional block approach does.

Webhook After that, *Watson Assistant*'s capabilities can be enhanced by employing a webhook. A webhook is a custom-made piece of *Javascript* code which can be executed from within a conditional block. This code can take into account the context of the conversation (intent, entities, slots and which conditional block it is being called from), perform specialized actions and return the answer the chatbot is supposed to tell the user.

A webhook can thus be used to perform actions as per request made by the user, such as querying a database for example, but can also be used to add features to dialog management. We explore such an improvement in chapter 5.

System entities Another extremely helpful improvement that we can cite is the ability to use predefined entities, referred to as "system entities". Indeed, the developers of *Watson Assistant* defined a certain number of entities that are commonly used when designing chatbots. For example, system entities include people's names, numbers, currencies or dates. It is even possible to get the values of some of those entities, such as the date, in different formats, which can be extremely useful when trying to internationalize a chatbot for example.

Fallback intent Finally, a very useful improvement is the possibility to define a fallback intent. When a user message wasn't understood (by default when the confidence level for the detected intent is smaller or equal to 20%), the dialog manager will fall back into a conditional block and utter a prepared answer to tell the user their message wasn't understood. The NLU model can even be improved by labelling some messages as out-of-scope, such that when a message is classified as out-of-scope, the dialog manager executes the fallback conditional block.

Developer interface

Watson Assistant is meant to be used by designer from the browser, through a Graphical User Interface specially designed for *Watson* chatbots. This has the advantage of allowing the design process to be done from any device connected to the internet, whatever its operating system. We should note that we tested this developer interface on computers using several different browsers and operating systems without having any issue whatsoever. As we explained earlier, the produced chatbot is also meant to be used inside a browser, even though it is possible to integrate it in another application, such as a mobile application. Nonetheless, an internet connection is still needed.

Figure 4.1 shows this developer interface. We can clearly see the conditional blocks on the left of the image. During development, the designer can test the chatbot in the small messenger view that can be seen on the right of the image. Other views exist to define intents and entities.

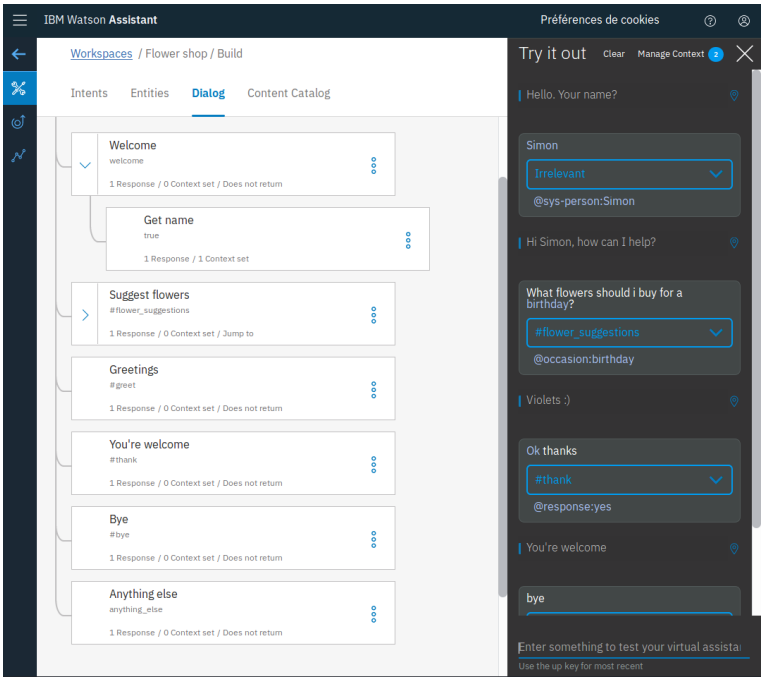


Figure 4.1: A screenshot of the developer interface of *Watson Assistant*

4.4.2 Available *Watson Assistant* modules

Watson Assistant being part of *IBM*'s *Watson* project, it is designed to have easy interactions with other modules from this project. This interoperability allows chatbots designers to expand the capabilities of their chatbot even further. That being said, we should note that after trying some of them out, the integration process is definitely not as easy as we could expect. Here is a list of several available modules that seem relevant for chatbot design.

Watson discovery This module is meant to automatically extract information from general texts. It can thus be used by a chatbot to handle out-of-scope questions. Indeed, rather than saying it didn't understand or cannot answer the question, the chatbot might make a search for the question in a set of predefined web pages and documents and exploit this module to summarize the information from the search results. This summary may then be displayed to the user as an answer.

Text-to-speech As suggested by its name, this module can transform a written text into an audio signal. A chatbot may thus read aloud its answers to the user, either as an accessibility feature (for blind people for example) or, along with the use of the *Speech-to-text* module (see below), to turn the chatbot into a voice-based one.

Speech-to-text This module does the opposite of the previous one: transforming voice recordings into text. It can thus once again serve as an accessibilitiy feature for disabled people, or help turn a text-based chatbot into a voice-based one.

NLU This module clusters several tasks related to NLU, such as sentiment analysis, emotion analysis or semantic roles recognition. As we said in chapter 2, having more information about user messages can always be convenient to make the chatbot seem more natural.

Visual recognition With this module, it is possible to process images, i.e. get a text description of the image. A chatbot can then support receiving images from the user. We could think of applications such as creating a chatbot that could tell the user information about a product they took a picture of for instance. There are of course many other applications.

Language translator This module allows to translate text from a natural language to another (supported) one. It could be employed to make the chatbot understand and answer in other languages than the one it was trained for. However, as of today, we shouldn't expect very good translations, and therefore rather bad performances for such a chatbot.

4.4.3 Pricing

Watson Assistant being a professional tool, a certain number of pricing plans are available to its customers. As expected, those plans have different limitations and prices [52]. Note that the limitations prevent the overloading of *IBM*'s servers.

Our analyses were made using the free plan, referred to as "lite plan".

Lite plan Maximum 10 000 API calls (i.e. handling of a user message) per month, maximum 100 intents and the chatbot is deleted if it hasn't been accessed for 30 days

Standard plan $0.002 5 per API call, no limitations on the number of intents and a service-level agreement (SLA) of 99.5%

Premium plan Extra features (such as *intent conflict resolution* and *search skill*), SLA of 99.9% and a better isolation for the data

Obviously, those prices are subject to changes as time passes.

4.4.4 Guarantees and data privacy

According to their terms of service, *IBM* guarantees they will keep private the developer's contents gathered on their service, unless this content is necessary to perform tasks related to the service (for example, the chatbot's relevant data will be sent to the website it is deployed onto) [53].

4.5 Google's Dialogflow framework

4.5.1 Presentation and description

As *IBM*, *Google* provides the public a chatbot framework. This service is called *Dialogflow*. In this section, we present it and describe its technical details.

Dialogflow was formerly known as *API.AI*, developed by the company *Speaktoit* and employed in an application with a similar name. It has been available to third-party developers since September 2014 and is also the framework exploited by *Google* to make their *Google Assistant*.

NLU performance

Technically, *Dialogflow* has one of the best NLU components in terms of performance. The fact that it is closed source prevents us from knowing what kind of algorithms it internally uses.

Improvements to dialog management

Its creators have taken a very interesting approach for the dialog management, that looks like conditional blocks with several key changes and improvements. A non-exhaustive list of additional features and improvements is provided hereafter.

Priorities and contexts The most interesting change is certainly the way *Dialogflow* chooses the conditional block to execute. Rather than having one conditional block per intent, *Dialogflow* allows the designer to define several blocks matching one intent. As the dialog manager matches the blocks in sequence, if nothing is made to differentiate two blocks associated to the same intent, the first one in the list will always be chosen. Two features allow to make the choice more dynamic: priorities and contexts.

As can be expected, priorities are labels that can be associated with conditional blocks and that are looked at by the dialog manager first. There are actually four priority levels (and an extra "ignore" level), making it possible to rank blocks. However, it seems that many more intermediate levels are accessible by programmatically tweaking some values. As the perceptive reader will have deduced, priorities are useless by themselves, as the same behavior can be achieved by reordering the conditional boxes. They are indeed useful only if they are used in conjunction with contexts.

Contexts are the way *Dialogflow* supports conversational state. At the end of the visit of a block, the designer can activate a context, conveniently called an output context. Upon message reception, the dialog management unit will visit the first block which correspond to the detected intent, with the highest priority and whose input contexts match as many currently active contexts as possible. Contexts can be deactivated in an intent or are deactivated automatically based on how long they have been active: when activating a context, the designer gives it a lifetime which is the number of conversation turns it will be active for.

Contexts also allow the designer to move entity values from one block to another one. Detected entities will automatically be associated with the active context(s). The next visited block can then fetch those values from its input context(s).

The combination of intent order, priorities and contexts allow designers to define quite complex dialog flows. This being said, they can be quite hard to completely fathom.

Follow-up intents A feature linked to contexts are follow-up intents. They allow to define a behavior that can be achieved using contexts, but that is very often required when designing chatbots. Obviously, it indeed happens very frequently that the designer wants to define the following type of dialog flows: the chatbot asks the user a question and depending on their answer, different blocks are visited. This kind of process can be achieved by activating a context after the bot has asked the question and define one intent per possible answer. The blocks associated to those new intents would require as input context the one activated after the question. Follow-up intents create exactly this kind of flow, and display them on the interface in a way that clearly shows the relationship between the different boxes, that is a hierarchy of contexts.

Triggering events There is actually another way to trigger the visit of a conditional block, using a feature called events. They are special triggers that don't correspond to the detection of an intent, but to some occurrence that is not directly linked to what the user is saying. The most notable events are the user "logging in" the conversation or them not answering for several minutes. Having a block triggered and thus the chatbot uttering something when such events happen can be useful.

Value placeholders Then, predefined answers in *Dialogflow* are more powerful than plain text. As with *Watson Assistant*, designers can put the value of detected entities (or values that were transmitted using a context) inside answers by using placeholders in their definition.

List of answers Similarly to *Watson Assistant*, it is possible to provide a list of predefined answers associated to a conditional block, in order to prevent the chatbot from feeling too deterministic. The dialog manager will randomly choose one of them.

Different answer formats Answers can also be in other formats than text: the produced chatbots are able to send images, quick answers (i.e. clickable buttons with predefined user responses) or even cards (images with a title, subtitle and interactive buttons). Obviously, sending such messages can only be done if the chatbot is deployed within a service whose UI supports those formats.

Integration on different platforms *Dialogflow* allows easily integrating its chatbots on a certain number of supported services. Such services include *Facebook Messenger*, *Slack* or *Skype* for example. It is possible for the designer to define replies that are platform-specific in two different ways. First, the format can be specific to the platform which they will be displayed on. Second, it is possible to define a list of responses for each platform, meaning the produced chatbot can answer different things depending on its environment.

Conversation end After that, a very simple additional feature is that blocks can be marked as conversation ends. When the block is visited, the chatbot will utter its last response and consider the conversation stopped. Depending on the platform it is deployed on, it will take certain actions, such as closing the conversation, logging off, etc.

Goals Another handy feature is the possibility to make a conditional block a goal (as in the goal-based architecture). In other words, when this conditional block is visited, the dialog manager will ask the user a certain number of predefined questions in order to fill some slots. Once the slots are filled, the chatbot can utter the response associated to this block. This answer can use the values of the slots.

Adding a goal into a conditional block is remarkably easy for the designer, as they just have to define the slots that should be filled and to specify the question to ask for a specific slot value. After that, *Dialogflow*'s dialog manager handles everything automatically.

System entities Then, as with *Watson Assistant*, designers can use system entities, which are entities predefined by the creators of *Dialogflow* and that are often confronted when designing chatbots. This way, there is no need to define them again. The value of those entities are formatted in an international way.

Webhook Yet another feature that *Dialogflow* shares with *Watson Assistant* is the webhook. As a reminder, it is a personalized piece of *Javascript* code that can be executed from a conditional block. It allows the chatbot to accomplish actions requested by the user and to improve its capacities.

Fallback intent Finally, another feature that both *Dialogflow* and *Watson Assistant* have is the fallback intent, i.e. a conditional block that is visited when the user message isn't understood. The designer can thus decide what the chatbot should say in that case.

Developer interface

As for *Watson Assistant*, chatbot designers are supposed to work with *Dialogflow* from the browser, using the developer interface. Again, we used it with different browsers and operating systems without experiencing any issue, other than a few visual mishaps. Figure 4.2 shows this interface: in the center, you can see the conditional blocks, with follow-up intents being represented in a hierarchical manner; on the right, you can see a test view, showing what the NLU component detected and what the dialog manager decided to do, given the latest user message.

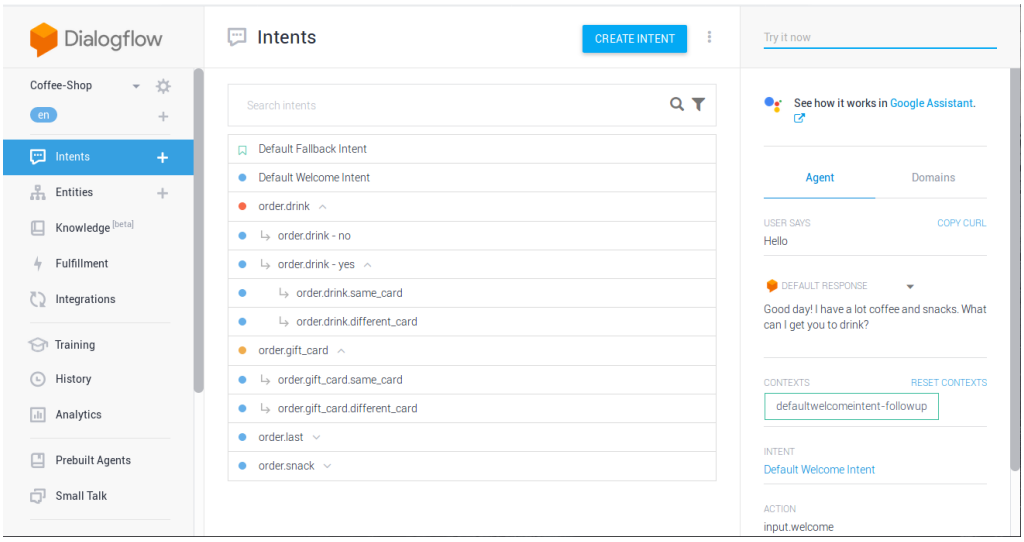


Figure 4.2: A screenshot of the developer interface of *Dialogflow*

Integration within other services

Contrarily to *Watson Assistant*, the produced chatbot is not especially meant to be used from the browser, but rather integrated within another service. They can be deployed with ease on quite a large number of platforms, even though their main objective is to be used with *Google Assistant*, which is *Google*'s platform for deploying chatbots. At the time of writing, the other platforms notably include *Facebook Messenger*, *Slack*, *Twitter*, *Skype* or *Telegram* [54]. It is also possible to build your own interface and exploit the provided API. To have more complex interactions, the webhook can be employed and allow interactions with other services.

4.5.2 Available *Dialogflow* modules

As for *Watson Assistant*, it is possible to add some features to a chatbot produced with *Dialogflow*, using plug-in modules. This being said, fewer modules exist for *Dialogflow* than for *Watson*

Assistant. However, this allows the integration process to be easier, since the interface between the main service and other modules need not be as complicated.

Here is a complete list of the modules that can be used to complement *Dialogflow*.

Speech-to-Text and Text-to-Speech Those two modules allow to make text-based chatbots voice-based, by simply using the Speech-to-Text module to transform user's voiced utterances into text messages, and the Text-to-Speech module to read the chatbot's answers out loud. Those modules are also often called *speech recognition* and *speech synthesis* modules. Such modules were presented for *Watson Assistant* as well.

SmallTalk This module contains a large number of predefined intents and conditional blocks (not displayed even when activated). If this is used, the produced chatbot is able to have smalltalk with the user, i.e. basic, mundane conversations. This can be extremely convenient to create a chatbot able to answer things that are not in its dialog scope, but that users often ask, such as *Are you a chatbot?* or *Thank you*. If this is not activated, the chatbot would always visit its fallback intent and tell the user it does not understand. No such module available for *Watson Assistant*.

Knowledge connector This is a module that can be very useful if it works well: when using it, if the chatbot does not know the answer to a question, it can search a predefined set of web pages for an answer. In other words, it gives the chatbot the ability to answer questions it wasn't trained for by summarizing relevant texts found by a web search engine. This seems to be powered by the same technology that makes "cards" on *Google*'s search engine, i.e. highlighted pieces of information coming from different web pages and that might answer the user's requests.

However, this module is still in beta and seems to often misunderstand the questions being asked. Plus, most of the time, designers don't really want their chatbot to try and answer every question they get asked. Moreover, it is not possible to correct a misinterpreted question by adding examples.

Phone calls This module allows the chatbot to communicate with users by calling them on the phone. A certain number of changes are made to the interface to allow the designer to control aspects relevant to voice interfaces, such as specify when the chatbot should make a pause in a sentence and for how long. This module is also in beta and does not seem to be available in all countries.

4.5.3 Pricing

Dialogflow is made available to third-parties by *Google*. There is a free plan, called the *standard plan* which comes with limitations, and two payed plans that are less limited [55]:

Standard plan Free for limited API calls, limited *knowledge connector* usage and limited speech recognition and speech synthesis (other services are also limited)

Enterprise Essentials plan $0.002 per API call, limited *knowledge connector* usage and $0.006 5 per 15 seconds of audio processed or synthesized (all other services unlimited but not free)

Enterprise Plus plan $0.004 per API call, unlimited free usage of *knowledge connector* and $0.008 5 per second of audio processed or synthesized (all other services unlimited but not free)

The limits are on the number of API requests per unit of time, here the number of handled user messages per day. For text-based chatbots that don't use any of the additional modules cited in those pricing, this is the only limitations we are interested in. The limits for the standard plans are 180 text requests per minute; those for the two other plans are 600 text requests per minute.

We should mention that it is advised even for small businesses to start with the standard plan, hence its name.

4.5.4 Guarantees and data privacy

The standard plan is subject to the *Google API's terms of services*, while the enterprise plans are both subject to the *Google Cloud Platform license agreement*.

Google API's terms of services are very permissive as you can do anything (legal) you want using the APIs provided. However, *Google* gives themselves the right to monitor your use of their APIs and to use the data they gather in any way they want. This can be a real problem for systems where privacy is an important issue.

The most notable difference for the enterprise plans is that *Google* guarantees it will not access or use the data in any way, unless you call their support in which case they might need to access some of it [56].

4.6 Frameworks comparison

The two frameworks presented in the previous sections, *IBM Watson Assistant* and *Google Dialogflow*, are those we considered to create our chatbot with. In this section, we compare them and explain our choice.

4.6.1 Similarities

As we could see, they are very much alike, both in the interface and in the features they offer. We can assume most of those similarities are simply due to the fact that both services tackle the same kind of problem. We will summarize here the most prominent similarities.

Global architecture First, as expected, the global architecture of both frameworks seems to be the same, containing two of the main components: the NLU component and the dialog management unit. As they are both meant to be integrated in third-party services, no UI is offered as part of the framework. We described this architecture in section 2.3.

NLU performance According to most studies, the NLU performance of both services are quite similar. Because of frequent updates, it is not always the same framework that is more performing.

In addition, performance not being only one measure but a summary of at least two (namely the precision and recall), it is not that easy to rank them. At the time of writing, it seems that *Dialogflow* has a better precision and *Watson Assistant* a better recall (cf. subsection 4.3.3).

As we already mentioned, from our experience, *Watson Assistant* works really well even when the training dataset only contains a few examples for each intent, which is not the case of *Dialogflow*. When the number of examples per intent is larger than about a dozen, this difference isn't noticeable anymore. If we use a NLU dataset generator, we can thus consider them to be similar in that regard.

Approach to dialog management Then, both services essentially take a similar approach to dialog management, as both their dialog manager use heavily customized conditional blocks approach. As we explained, they increase their capacities by adding specific features. Some of them are included in both of them, such as placeholders in answers, lists of answers to choose from, system entities or a fallback intent.

Webhook Moreover, a feature that both services provide is the ability, thanks to a webhook, to employ serverless functions, usually in JavaScript, executable from a conditional block.

Modules After that, for several modules usable with *Dialogflow*, there exist a comparable module for *Watson Assistant*: speech-to-text and text-to-speech exist for both services and *Dialogflow*'s *knowledge connector* corresponds to *Watson discovery*. However, all other modules usable with both service don't have a direct analogous one.

Developer interface Another obvious similarity is the developer interface we showed above for both frameworks. Indeed, both interfaces feature a tab to define entities, another one to create conditional blocks and a messaging app to test the chatbot on the right. Clicking on a conditional block opens a view which allows to specify examples for the associated intent, to define goals (in the goal-based meaning) and to list answers to tell the user. We can contrast that with the web interface of *Botpress*, which is completely different. This being said, this is partly due to their approach to dialog management which is closer to an improved FSM.

Pricing and plans The available plans for both services are quite similar: the same kind of limitations and possibilities are offered for free. Payed plans are slightly more expensive for *Watson Assistant* than for *Dialogflow*, but are both calculated with respect to the number of API requests that were made.

Supported natural languages Finally, both services can work with quite a lot of natural languages. Of course, those languages are the most prominent ones in the world, such as English, Mandarin or Spanish. The list of supported languages are not the same for each framework, but the ones that are relevant for a chatbot meant for UCLouvain, i.e. English, French and Dutch, are available for both of them. We should note that the NLU performance might vary depending on the language.

4.6.2 Differences

Even though a lot of things are similar between both frameworks, they are also different in several ways. Aside from the small differences we noted in the previous subsection, there are several

notable ones that will be listed hereafter. Notice however that some differences might disappear and new ones appear as both services are constantly being developed and changed.

Dialog management features First, as we saw earlier, lots of features of the dialog manager are restricted to one of the frameworks, forcing designers to use each of them differently. The most notable difference is certainly the way their respective dialog manager chooses which conditional block to visit. For *Watson Assistant*, the dialog manager stays by default at a certain level of the block hierarchy during several conversation turns, while for *Dialogflow*, the choice is made by inspection of contexts and priorities. Those processes are explained more in detail in the sections describing the frameworks.

We could add that it is possible to create dialog-wise variables in *Watson Assistant*, by using slots, which isn't possible using *Dialogflow*. However, using contexts to pass entity values from one block to the next one could allow for an equivalent but less powerful feature. It seems that it is possible to achieve the same conversational logic whichever service we use.

Available modules and integration process Another important difference is the number of available modules to boost the capacities of the frameworks. As we saw, there are a lot more modules usable with *Watson Assistant*, but most of them are not really relevant to chatbot design. We listed in subsection 4.4.2 those which make sense to be integrated in *Watson Assistant*. We can then see that several modules only exist for one of both services. If it is absolutely crucial to have a feature enabled by one of those service, the choice is then easily made. That beind said, it would be possible to implement a module by ourselves using the webhook in both services. Nevertheless, we should note that it is totally possible to make a satisfactory chatbot without the use of additional modules.

Something which should be mentioned is the fact that the integration process of those modules inside the corresponding framework is easier when using *Dialogflow* than *Watson Assistant*. We can assume this is due to the fact that there are fewer modules for *Dialogflow* than there are for *Watson Assistant*.

Deployment within third-party services We can make the same remark for the integration of a chatbot produced with those frameworks onto third-party platforms: this process is again easier with *Dialogflow*. A certain number of features that allow to customize the behavior of *Dialogflow* chatbots depending on the platform they are integrated on are even included, which can be extremely handy. We should also note that *Dialogflow* chatbots support being deployed on a wider variety of platforms out-of-the-box than *Watson Assistant* chatbots do. Generally speaking, the deployment of *Dialogflow* chatbots is easier than that of *Watson Assistant* ones.

Webhook integration The same remark can be made again for the webhook, which is very easy to activate and utilize within *Dialogflow* contrarily to *Watson Assistant*. A *Dialogflow* webhook can be stored on any cloud provider unlike in *Watson Assistant* whose webhook must be stored on *IBM*'s cloud.

Developer workspace sharing We should note that it is also extremely tedious to share a chatbot workspace between two people when using *Watson Assistant*. The process for doing that takes about a dozen (arguably senseless) steps and isn't explained anywhere in the documentation. Sharing a project between more than two people seems even more difficult. On the contrary, the

same process takes two clicks with *Dialogflow* and only requires the creator of the workspace to know the (*Gmail*) email address of the other developer.

Documentation Finally, we should mention that the documentation of *Watson Assistant* is quite cumbersome to read. First, the language is automatically chosen based on your location, and there is no easy way to change it (thus in Belgium, it is either Dutch or French), even if the documentation quality heavily depends on the language.

Then, it is quite hard to find: there is no links to it from the developer interface and this documentation is actually not even stored on the same website. We also noted that some information is simply lacking from this documentation, notably the sharing process we mentioned before.

Ultimately, it happens to find out-of-date documentation. It indeed seems that outdated documentation isn't set offline upon *Watson Assistant* updates, which occurs several times a year. This also makes it hard for search engines to reference the latest documentation. We should mention that *Watson Assistant* being often updated (particularly its web interface), it is often required to have the latest documentation. Not having access to it forces the designer to find their way in a new interface, possibly with new features or even without the feature they used to employ.

4.6.3 Choice of the service

For the chatbot created in the present master thesis, we had to choose which framework to work with and decided to use *Dialogflow* rather than *Watson Assistant*. The main reason for our choice is the overall ease of use of *Dialogflow* associated with a better documentation, and not a question of capabilities. Specifically, the fact that *Dialogflow*'s projects are easier to share and to deploy was a large factor in our choice.

Additionally, *Dialogflow* has the advantage to master small talk (or chit-chat), thanks to a module which does not exist for *Watson Assistant*. Making the other choice would have meant to define all the relevant intents ourselves, which would have taken a very long time and certainly be less performing than those defined by *Dialogflow*'s creators.

A last thing that made our life easier is the fact that the free plan of *Dialogflow* has no limits in time, unlike *Watson Assistant*'s free plan which deletes chatbots when they haven't been touched for 30 days.

Chapter 5

Conception of *JAQ*

This chapter describes the conception process of a chatbot from the point of view of the designer, using a framework such as Dialogflow. The conception steps are illustrated with the chatbot we made, JAQ, and the choices we made in that context will be explained.

5.1 Key steps

Before starting to concretely work within the developer interface, there are a certain number of things that have to be prepared, namely:

- Decide how the produced chatbot should behave like, notably its name and attitude;
- Decide which topics are in the dialog scope and gather data related to them;
- List the questions that the chatbot is supposed to support and the answer to those questions if possible;
- Define conversation flows, where the user and the chatbot have complex interactions;
- Assess the quality of those questions and conversation flows and fix potential problems.

These steps can be iterated and are schematized in the figure 5.1. One could recognize the Deming cycle[57] adapted to our case, whose steps will be described in the next sections.

5.2 Name and attitude selection

Before designing a chatbot it is important to decide its attitude towards the user. In other words, the designer should ask themselves whether it will seem kind, polite, friendly, angry, unintelligent,... This choice will not only guide the way answers will be written by the designer, but also may have an influence on the design of conversation flows.

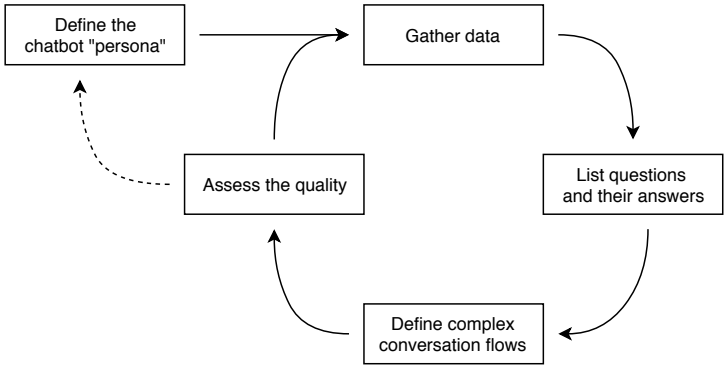


Figure 5.1: Conception steps

For our chatbot, we decided to make it seem polite, helpful and serious, since it will be associated with the university, which is an official and educational corporation.

A nice additional thing to choose is a name for the chatbot: it will make it more approachable and close to users, giving it some personality. This is however completely optional. After long discussions, we settled on the name *JAQ* which stands for *Just Answers the Questions*. (Note that our first choice was *LUC*, but that name was already in use for a similar master thesis at UCLouvain.)

Finally, a lot of chatbot designers choose some kind of a "persona", i.e. a whole character associated with the chatbot. Such a character design is nice if the designer really wants the chatbot to impersonate someone or something (a robot, an animal, etc). For our case, it seemed excessive; having a name and a well-defined attitude seemed enough.

5.3 Typical users description

It is important to know by who the chatbot is intended to be employed, as this will shape its dialog scope. We intend *JAQ* to be used by students or future students who come from abroad, i.e. Erasmus and international students, and expect them to ask questions mainly about:

- The living environment in Belgium and in Louvain-la-Neuve
- General information about the university
- The available programs they can follow
- The registration process
- The process of enrolling to courses and exams
- Available accommodations
- Contact information of people that could help them if needed

We thus know a certain number of topics that we would like our chatbot to support. We should mention that we do not expect to design a chatbot that answers precise questions about very specific situations.

Note that a second potential user are secretaries responsible for foreign students and who need to quickly check an information.

5.4 Gathering and preparation of the data

Once we have decided which topics the chatbot should support, we can gather data that covers them. This data will then be used to

- list all the questions that the chatbot should be able to answer;
- prepare the answers to those questions, containing correct information.

In our case, there are a certain number of ways we could find this data: our own knowledge as UCLouvain students, Frequently Asked Questions pages on the official website of the university, other websites related to it, employees of UCLouvain, etc. The main source of information we used are the FAQs that can be found on UCLouvain's website; the exact web pages we used are listed in appendix B. Obviously, all this data is not formatted in a well-defined way, nor is the information contained in there directly exploitable. We thus had to read through all this documentation, understand the information present there and make it exploitable, which cannot be done automatically.

To choose which questions *JAQ* should support, we had to imagine what being an international student would be like. We took inspiration primarily from the FAQs and divided all the information we could find into 13 (correlated) topics:

- Accommodation
- Belgium
- Contacting services and groups related to the university
- ECTS (European study credits)
- Exams
- Languages
- Registration in the university
- Schedule
- Study organization
- Presentation of UCLouvain
- UCLouvain's websites
- Welcome to Belgium, Louvain-la-Neuve and UCLouvain
- Welcome events

Once enough information was gathered about a topic, we listed all the questions about that topic that seemed relevant to us.

To create the answers to those questions, we then organized the gathered data in a way that let us see the big picture and easily retrieve some pieces of information. To do that, we used simple

HTML pages where it is easy to have hypertext links. We made one page per topic and linked to the source of the information and to related pages on each page. You can see one of them on figure 5.2.

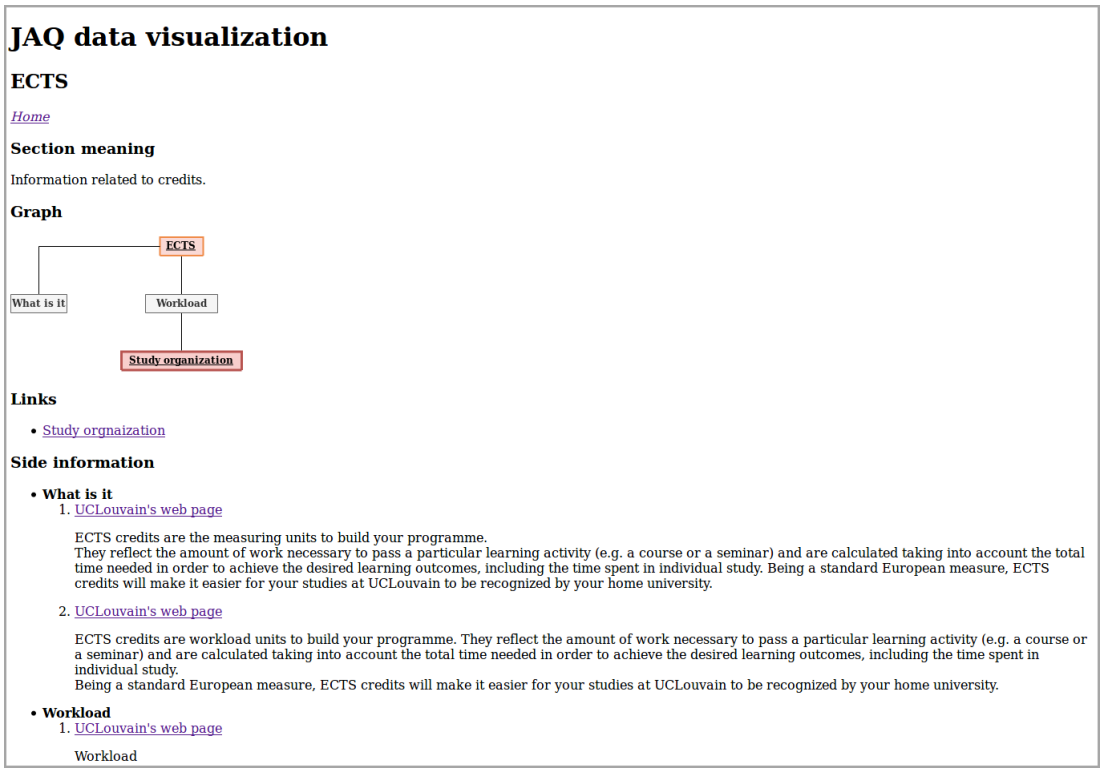


Figure 5.2: HTML page to organize the information

We should note that we tried to automate this gathering of data, but scraping the website of UCLouvain is quite difficult since its HTML code is quite cluttered. Moreover, we would have required an extremely powerful algorithm to find useful information automatically within the data we could gather. Manually gathering data then seemed to be the most effective, but extremely time-consuming.

5.5 Definition of simple question-answer pairs

Expectedly, there are a certain number of questions that are very simple, meaning they have a definite answer without requiring additional information from the user. In other words, the chatbot does not need any context to be capable of answering the question. We can thus define a certain number of pairs question-answer.

Here is an example of such a question:

Student: Is UCLouvain located in Belgium?
Chatbot: Yes, UCLouvain has several sites in Belgium. Its main site is in the city of Louvain-la-Neuve.

On the other hand, a question for which requires more complex interactions, since the answer depends on the situation of the student (namely do they come from the Schengen area), would be:

> *Student:* Do I need a visa to come to Belgium?

Note that a chatbot that would only support such simple questions would be very easy to make, even if we used a very simple approach to dialog management. Indeed, once the intent classification is done, the chatbot would answer the corresponding response. Of course, modern chatbots usually also include more complex interactions, where context and additional information are required.

Using the exploitable data we gathered, we thus made a list of simple questions and their associated answer, as well as a dataset containing examples of those questions so that the NLU model can be trained. We stored all this material in a specific format which will be described in chapter 6, so that we can automate most of the subsequent processes.

5.6 Establishment of complex interactions

As we explained earlier, adding more complex user-chatbot interactions will make the chatbot behave in a more human-like way. We showed in the previous section what we meant by complex interactions: questions whose answer depend on additional information the user has. In other words, such interactions need several conversation turns to complete.

To speed up conception and avoid having lots of different complex interactions to define, we decided to restrain ourselves to a generic type of interaction: when the user asks a complex question and an additional piece of information is required, it asks the user for it until it acquires it or the user change their mind and ask another question. We indeed realized that this kind of interaction is very frequent for complex questions in the case of a chatbot meant to answer definite question.

We should note that this type of interaction is very reminiscent of a goal-based approach to dialog management, with a slight difference: with the goal-based approach, the values of the slots to be filled are forgotten once the goal is reached. This can make the chatbot cumbersome to use in longer conversation, as it could ask for a certain piece of information several times. On the other hand, with our approach, the pieces of information keep being stored in an active context for as long as we want, meaning the chatbot does not forget them after it answered a question. It is also possible to store pieces of information that are not especially relevant for the current question, but might come in handy later on. From the viewpoint of the user, this has the advantage of allowing to ask a question and start giving required information, then changing their mind and asking another question, and finally coming back to the first question without having to provide the same pieces of information twice in the conversation. Of course the user should also be able to ask a question with the additional required information all together.

This kind of interaction is illustrated in the figure 5.3, which is read from the center outwards. The orange boxes represent the responses of the chatbot. Upon user input, we check whether its intent corresponds to a question, an entity supplying (e.g. *I am Belgian* would be classified as an intent for which the user gives their nationality) or not understood. If it is a question and the chatbot can answer it with the information it has, it will simply answer. If it is a question and an entity is missing, the chatbot will ask for the information. If the user input is simply providing an

entity value, the chatbot either answers the pending question (if any), or simply acknowledges it understood.

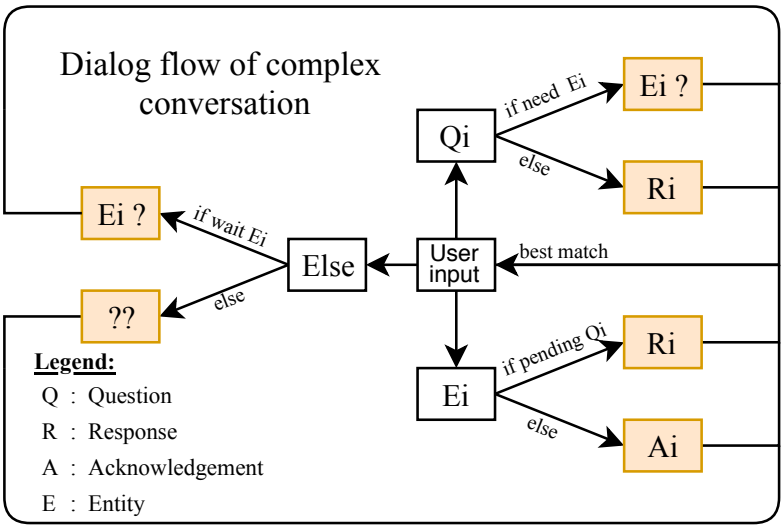


Figure 5.3: Complex flow conception

The fallback management was also improved compared to the simple question-answer pair: when the chatbot does not understand the user input and was expecting an entity value, it simply tells the user it did not understand and was expecting a certain entity, rather than going to the fallback intent (and thus only utter that it did not understand). Moreover, if the chatbot does not understand user utterances several times in a row, the fallback response changes, and shows examples of supported questions. This functionality aims to make the chatbot less irritating to use when the user does not know the exact conversation scope.

5.7 Assessment of the quality of the questions

Once the pairs of simple questions and answers, and the complex conversation flows have been created and included in the chatbot, we need to assess their quality. It is indeed extremely difficult as a designer to predict all the possible questions and interactions that the user would want to have with the chatbot, but it is possible to add support for the lacking ones later on. As shown on figure 5.1, making the process iterative allows to increase and improve the current support for questions of the chatbot. The process is thus building a chatbot from the list of simple questions-answers and complex dialog flows, assessing the quality of the produced chatbot and updating the questions and interactions accordingly.

We made two types of tests, at different points in the conception. First, we tested the chatbot ourselves, asking it questions inside and outside of its scope to check its behavior. Then, we asked external people to use *JAQ*, impersonating an international student, and we took note of what happened. There are five aspects that were unveiled by those tests:

Webhook fault There could be faults in the code of the webhook, messing up the chatbot's behavior or completely stalling the conversation, when the chatbot is supposed to have a complex interaction.

Misclassified question A supported question could be misclassified, leading the chatbot to answer an irrelevant response. In that case, we are likely lacking examples for that particular intent (for example, because we did not think of a specific way to express this question), or have too general ones for the intent that it was classified as. There is also a possibility of overfitting which we will discuss in chapter 8. The same kind of defect could exist for entity recognition task, meaning an entity value provided by the user could not be picked up by the dialog manager.

Unexpected question The user could ask a question that is relevant to the scope of the chatbot, but that we did not anticipated. We would thus need to add it to the list of questions to be supported, create examples for the NLU component to train on, and add an answer for it if that question is a simple one. If it requires more complex interactions, we need to define them. Those questions we overlooked were usually revealed when we made tests with external people.

Unnatural interaction The user could have an interaction with the chatbot that is not natural, unveiling a misconstruction in the conversational flow we defined. This error then needs to be fixed and the general approach to complex interaction could be revised if needed. Indeed, we needed several trials and errors before reaching the global approach to complex interactions we showed in the previous section.

Inaccurate answer An answer uttered by the chatbot could lack information or mislead the user. We can also mention errors such as typos in answer. We would thus need to fix this particular answer. We could find out that kind of defect when performing tests with exterior people.

Chapter 6

Implementation of *JAQ*

This chapter describes the tools and scripts we developed to ease and speed up the conception process of JAQ. We explain the reasons we decided to create those tools and the choices we made during their implementations. We then outline their technical abilities and functioning.

6.1 Developed tools

6.1.1 Initial motivation

As we explained in chapter 4, we chose to create *JAQ* using the framework *Dialogflow*, which was presented in section 4.5. We notably showed the developer interface which is available using a web browser. We could have employed this interface, but having a large number of intents inside it is quite cumbersome, since we would need to navigate around many conditional blocks. Moreover, the loading times between the different web pages being quite long and the amount of times we needed to change page quite high, this would have been very time-consuming. We thus decided to automate this process, making our own tools to manage large numbers of intents and conditional blocks. The source code of those tools, along with the formatted data we employed, is available at `https://github.com/gellens/Master_thesis_JAQ_code`.

We had two options to make those automatization tools: either we could use API calls to create new conditional blocks and populate them, or we could use *Dialogflow*'s abilities of importing and exporting a representation of the chatbot, called a workspace, in *JSON* format. (Note that the second option is also available for *IBM Watson Assistant*.) Since it was very likely that we would exceed the limitations of the free plan on the number of API calls, we settle for the second option.

More precisely, except for the scripts used for experiments, all the scripts we created are meant to produce importable files containing everything that portrays the agent we want to create, in *JSON* format. Note that it is not required to create those files from scratch as those scripts could use files exported from *Dialogflow*, and add information into it. This is what we did, as shown by the figure 6.1 where the files used as a base are listed. Strictly speaking, the workspace files thus contain the following information:

- The conditional blocks, with all the information they are associated with, i.e. the intent (as

a list of examples), the answer(s) to utter when visiting that block and the logic of context setting, value storing, etc;

- The entities, defined by their label, possible values and their synonyms;
- The code of the webhook and when to call it.

We can then easily import those files within *Dialogflow* using the web interface.

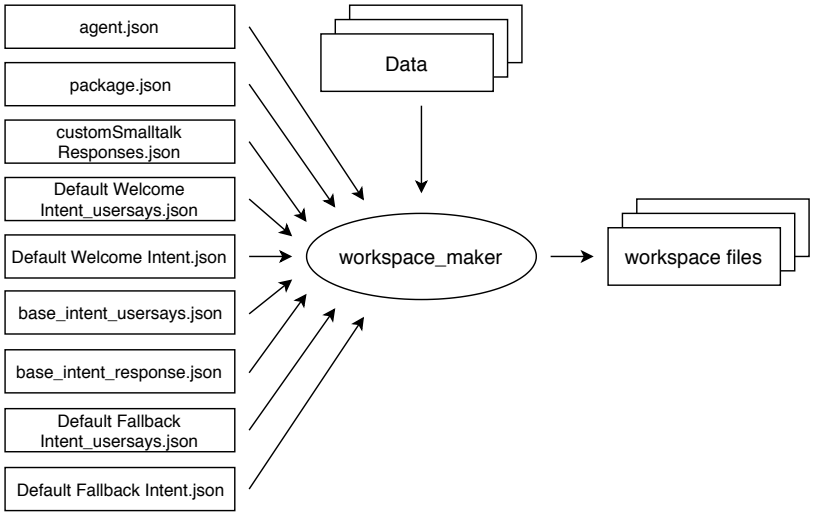


Figure 6.1: Base files used for the workspace generation

6.1.2 General structure

When we started developing those tools, we directly followed the principles of "separation of concerns". Specifically, we wanted to separate the code of our tools from the data that would be used to import the chatbot agent. Therefore, we put all those data in "data files" with an ad-hoc format, which would be then handled by the different scripts. We will explain the contents of the input files in subsection 6.1.3.

We wrote the scripts using the *Python* programming language, for the sake of ease and development speed. Also, we did not have hard efficiency and performance constraints. We conceived them iteratively, by adding features incrementally when we needed them. This procedure allowed us to create agents quite soon after we decided which framework we would use.

Aside from the *Chatette* NLU dataset generator we presented in section 4.1, our system is divided into three different modules:

- `workspace_maker`, which is in charge of calling all the underlying scripts and creating the files that will be imported into *Dialogflow*;
- `FAQ_maker`, which is intended to merge most of the files that contain the data characterizing the chatbot;

- `html_to_text`, which is intended to format the chatbot's answer where relevant so that they are displayable on *Facebook*.

This global architecture is represented on figure 6.2, where all the scripts and files are represented. The orange files are responsible for the complex interactions. The large white arrow represent a system call used to execute `Chatette` and the black arrow show which files are inputs and outputs to modules. The diamond arrow for the `html_to_text` module indicates it is used as a library inside `workspace_maker`.

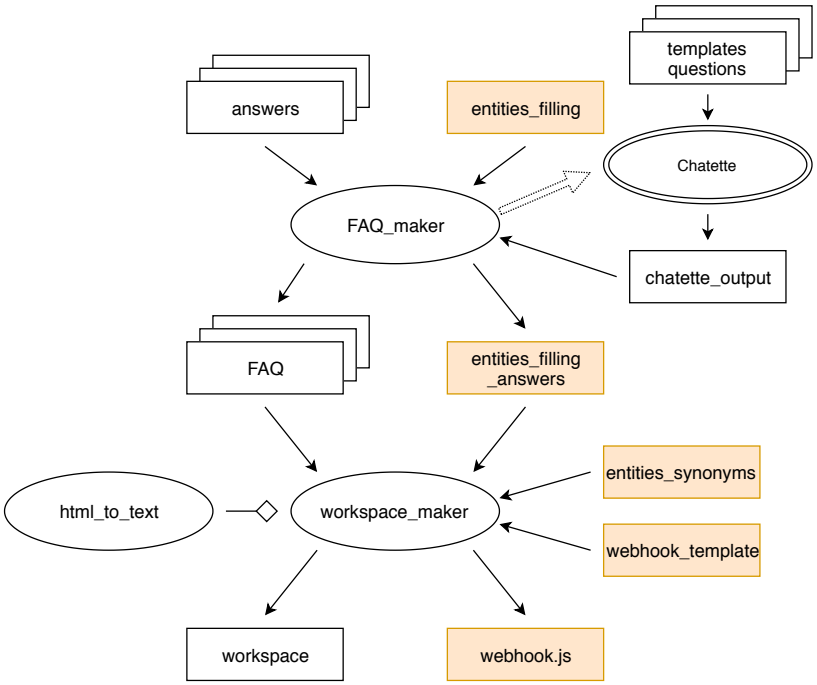


Figure 6.2: General structure of the developed tools

6.1.3 Contents of the input files

As can be seen on figure 6.2, there are a certain number of files that contain different categories of data required to create the final chatbot. We wanted to keep the files readable, so that we could work iteratively, adding questions, examples of utterances and answers at any point of our process. This allowed us to follow the cyclic development process described in chapter 5. For this reason, most of our files are in Markdown, which has the advantage of having a very light overhead while still being structured and human-readable [58].

We will now quickly go over the contents of each of the input files of our scripts.

- `answers` are Markdown files which contain lists of intents and their associated answer(s). There is also an example of the associated question. In other words, they contain almost all the data required to create simple questions.

- `template questions` contains templates that allow *Chatette* to generate many examples of user utterances. The generated sentences are associated to the same identifier than the one of

the question they are related to in the `answers` files. The syntax of this file is not Markdown, but a readable ad-hoc domain specific language.

- `entities_filling` contains a list of complex interaction, represented as conditions that can be made on entity values, as well as the question that needs to be asked to the user if that entity value is necessary. Those conditions are used in the complex interaction depicted in figure 6.3.
- `entities_synonyms` contains the definition of entities, with a label, a list of values and subsequent lists of synonyms (e.g. "Belgian" and "Walloon" both map to "BE" in our chatbot).
- `webhook_template` contains the skeleton of the webhook script, with a fixed part and a part indicating the code to generate. Code generation will be handled by the *Mako* engine. Therefore, this file is a mix of *JavaScript* and a *Python*-like syntax specific to *Mako* [59].

Some files are both outputs of some of our tools and inputs to other ones. Their contents are thus generated using data contained in files we previously described, and are as follows.

- `FAQ` contains a list of questions each having an identifier, a list of examples of user utterances that map to that question and a list of answers.
- `entities_filling_answers` contains the same information as `entities_filling`, i.e. a list of interactions in the form of conditions on entities and their associated utterance to query the user for it, each of those complex interaction being now associated with the corresponding answer.
- `chatette_output` simply contains a list of examples of utterances, with the associated intent identifier (thus here, the question identifier), in *JSON* format for legacy reasons and because it is very easy to parse.

Finally, we can describe the two files generated by all our scripts and importable into *Dialogflow*.

- `workspace` contains all the information required to create an agent in *Dialogflow*, in *JSON* format.
- `webhook.js` is a *JavaScript* file containing the webhook code called by the chatbot for complex interactions.

6.1.4 Description of the modules

`FAQ_maker`

This script is responsible of two things. On the one hand, it merges the contents of `chatette_output` and of `answers`, i.e. the question identifiers, associated examples of utterances and answer(s). In order for this to work, the question identifiers must be present in both files. This information gets compiled into the `FAQ` files, which thus contains all the data related to simple questions. Note that to do that, it needs to call *Chatette* and to execute it on the `template_questions` files.

On the other hand, it aggregates the contents of `entities_filling` that represent more complex interactions, and the associated examples of user utterances from `chatette_output`.

To summarize, it compiles data about simple questions in the `FAQ` files and data about complex interactions in the `entities_filling_answers` file.

Chatette

As explained in section 4.1, *Chatette* allows us to generate many examples of utterances based on templates we provide it. For example, from the template `What is {UCL/ UCLouvain}?`, it can generate the utterances *What is UCL?* and *What is UCLouvain?* It is more powerful than this very simple example, thanks to features like blocks of synonyms or recursive aliases[60].

This is a quite important part of the automation process, since we need many data to train the NLU component and that for each of the numerous intents. It allows to easily increase the variability of examples as well.

`workspace_maker`

The `workspace_maker` module is intended to build the workspace that will be imported into *Dialogflow*. This workspace contains everything that can be created using the developer interface in the browser, i.e. the intents, entities, associated answers and for all the platforms, their context information and whether the webhook needs to be called. This script is exploits `html_to_text` on the answers that will be displayed on *Facebook Messenger* (explained in detail below).

To create this script, we needed to carefully study the structure of a workspace file, as exported by *Dialogflow*, in *JSON* format. It seems indeed that it is not intended to be used this way, as there is no documentation about the use of *JSON* to characterize *Dialogflow* workspaces.

As we explained earlier, we also rely on the webhook to perform more complex interactions. Its code is generated in this module, by executing *Mako* to generate *JavaScript* code from the `webhook_template` template file we provide it. As complex interactions are always made using conditions on entity values, the `workspace_maker` script also uses the definition of entities contained in the `entities_synonyms` file.

html_to_text

This is the smallest module and is only meant to transform *HTML* code into raw text. We use this on the answers meant to be displayed on *Facebook Messenger*. Indeed, we designed our custom GUI in a way that allows the chatbot to use *HTML* answers, which means it can change the style of some pieces of text (e.g. make it bold, italic, etc.), show lists or convert clickable hypertext links to readable raw text. However, *Facebook Messenger* does not support displaying *HTML*. We thus need to transform the answers intended for *Facebook Messenger* into raw text and decided to do that automatically from the *HTML* answers rather than rewriting answers by hand.

For example, the HTML code `on <a href="https://site.com">this website</a>` would become the raw text `on this website (https://site.com)`. Of course, we wrote the answers keeping in mind that that treatment would be executed on them.

6.2 Implementation challenges

During the implementation of those tools, we had to address several challenges. Some of them were caused by our unusual use of the framework, some to the size of the scope *JAQ* is supporting and others to our willingness to support complex interactions.

6.2.1 Caused by the large dialog scope

The topics supported by *JAQ* are quite diverse, even though the potential questions can be very similar (from a vocabulary perspective). As you can expect, managing such a large scope automatically was quite a big challenge, for two main reasons.

First, gathering and extracting so much information and making it exploitable is not a trivial task, which is the reason we needed to do quite a lot of work by hand. Moreover, if *JAQ* was not a prototype, we would need to find a way to keep this information up-to-date, which is another challenge in itself.

Secondly, it was complicated to come up with an efficient technique to handle all this data accurately in a way that is scalable enough. Our solution of organizing the questions and answers within Markdown files makes it easy to support new questions, change some answers and list the currently supported topics and questions.

6.2.2 Caused by our usage of *Dialogflow*

A few of the challenges we faced are due to *Dialogflow*'s limitations and specificities. We will depict them here.

Unique fallback intent

When the user input does not match any intent, which means the chatbot did not understand the utterance, the fallback intent will be selected. This is in most cases very useful, as the designer can specify a list of answers that could be uttered by the chatbot when it fails to understand something.

However, it is not possible to have different fallbacks depending on a condition. Being able to check for the value of an entity could have been very useful for returning a different default message depending on the context, for example when handling a complex question. We had to use a webhook in order to work around this limitation.

Escalation fallback

As we explained earlier, having just one fallback can make the chatbot quite irritating to a user who does not know the dialog scope: they could keep asking out-of-scope questions, the chatbot answering each time it does not understand. To avoid that, adding a list of supported questions in the fallback answer was first investigated. However, this is quite tiresome to a user, as they get a long answer every time their question is not understood. We thus wanted the list of supported

questions to be uttered if the user fails to be understood too frequently. That being said, counting events is not possible in *Dialogflow* as is. Once again, we used the webhook to circumvent this limitation.

To be precise, the fallback intent calls the webhook code every time it is visited. The webhook can thus keep a count of how many times the user was not understood recently. An "escalation fallback answer", that is an answer that tells the user its message was not understood and provides several examples of supported questions, is returned by the webhook if the user isn't understood thrice in a row, or at most four times in at most thirteen turns. Those numbers are a consequence of the implementation, which uses the lifespan of a context to store its counter.

Nevertheless, a day before we handed over this dissertation, an update of *Dialogflow* broke this feature: it removed the possibility for the fallback intent to call the webhook. Due to a lack of time, we couldn't solve this problem.

Slot filling

As we explained in section 4.5, *Dialogflow* has a feature to give a conditional block a goal-based approach (cf. subsection 2.3.4). Nonetheless, we quickly realized that there was two main problems if we used this.

First, the user cannot change their mind once the chatbot is trying to fulfill a goal. For example, if we used this, the following interaction would have easily occurred:

> *Student:* Do I need a visa to come to Belgium?
> *Chatbot:* Can you first tell me your nationality?
> *Student:* Actually, tell me rather how expensive life in Belgium is.
> *Chatbot:* To know if you need a visa, first tell me your nationality.
> *Student:* No, I want to know if life in Belgium is expensive.
> *Chatbot:* What is your nationality?

Obviously, we would like to avoid that kind of behavior, since it is very annoying to the user.

Secondly, if the chatbot does not understand the entity value the user is providing, it will keep asking for it. This can stall the conversation, as the user might not know why they get asked the same question again and how to escape the loop. For example, the following conversation could happen, if *Cambodian* was not considered a supported nationality:

> *Student:* Do I need a visa to come to Belgium?
> *Chatbot:* Okay. First tell me your nationality.
> *Student:* I'm Cambodian.
> *Chatbot:* What is your nationality?
> *Student:* I am from Cambodia.
> *Chatbot:* Could you tell me where you come from?

We avoided those drawbacks once again by using a webhook. As we explained in section 5.6, the general architecture we used to manage complex interactions is a little more intricate than simple slot filling (in the goal-based approach to dialog management).

To be precise, when an entity is required for a question to be answered, we activate a context so that the chatbot remembers it is expecting an entity. Then, the user is allowed to ask something else and get an answer before providing the previously asked entity and have an answer for the first question. Since the context has a certain lifespan (measured in number of conversation turns), the chatbot can "forget" a question is pending if the user asks for other information for a long time (and hence refuses to give the value of the entity), as it would happen in a human-to-human conversation. Having different fallback intents depending on the active contexts allows to prevent the other problem: the chatbot can tell the user it does not understand the entity value that they provided.

Concretely, the webhook code to support complex interaction directly maps to the representation we showed on figure 5.3. The implemented logic using the contexts is showed on figure 6.3.

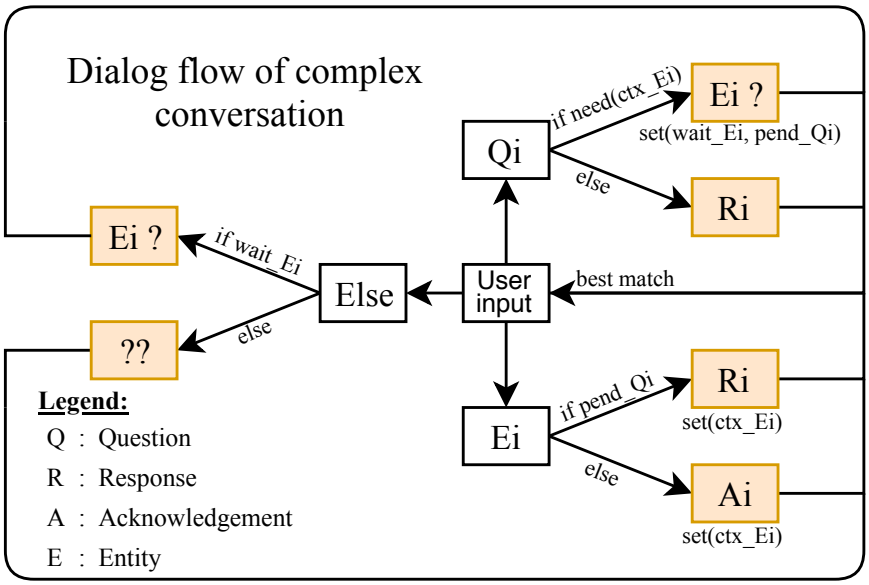


Figure 6.3: Complex flow implementation

When the NLU component detects a question that is associated to a complex interaction, we can employ *Dialogflow*'s context feature to check if the required entity is already contained in the corresponding context. If the context supposed to contain this entity value is not defined, the answer related to the intent will be uttered, asking the user for the required entity value, and a context `wait_Ei` will be activated to indicate which entity is expected. On the other hand, if the entity value is known, the webhook will be called and generate a relevant answer.

Its code contains a data structure which stores for each complex question the conditions (i.e. which entity is required) and the associated answers. Knowing the current intent and known entity values, the webhook can generate an appropriate answer.

When the NLU component detects the user is providing an information and there is a pending question (the context `pend_Qi` is active), the webhook is called to generate the correct answer in the same way as before. If no question is pending, it will activate the related context `ctx_Ei` so that it remembers it for later turns, and return an answer acknowledging an entity was provided and asking the user what they want.

Lastly, if the NLU component didn't understand the latest user message and an entity was

expected, it will tell the user it didn't understand and which entity it was expecting. If it was not expecting any entity, the returned message will simply tell the user the chatbot didn't understand.

Conditions on previous contexts

As explained before, for complex interactions, conditions on entity values and on active contexts were required. However, we also needed to make conditions on entity values that are stored in contexts activated previously, those entities being information the chatbot got in earlier conversation turns. This kind of entity condition is not possible within *Dialogflow*'s designer interface itself. This is another limitation the webhook allowed us to circumvent.

6.3 Integration within other services

As we explained in section 4.5, an important feature of *Dialogflow* is the ability to easily integrate the produced chatbot into third-party services, such as *Facebook Messenger*, Slack, Twitter,...[54] Those services represent the UI component of the chatbot. The "one-click" interface to do that is shown in figure 6.4.

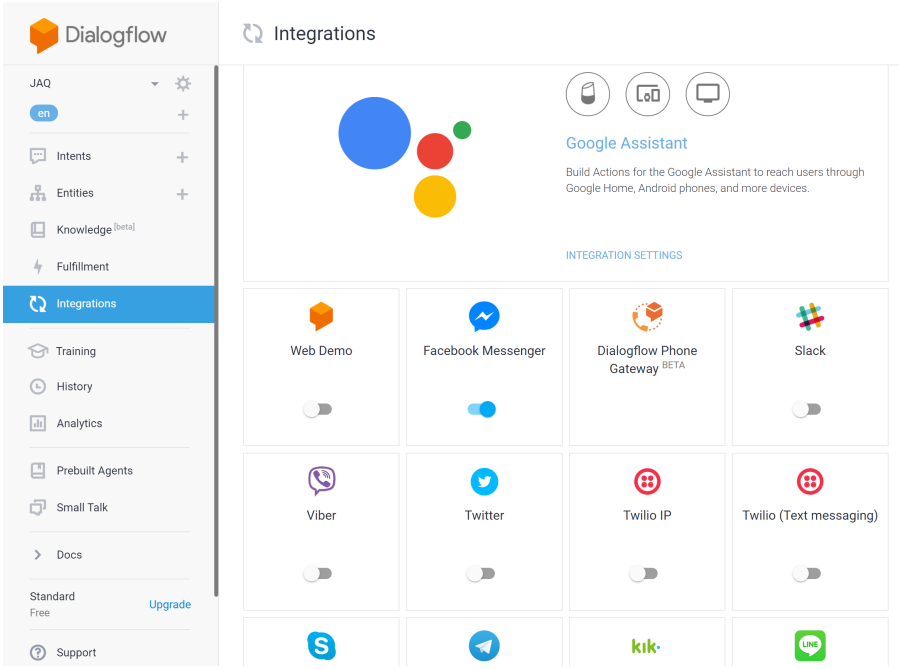


Figure 6.4: *Dialogflow* integration interface

As *JAQ* is a proof of concept, we decided to only integrate it in two different interfaces: a custom GUI and *Facebook Messenger*. The procedure we followed to create *JAQ* with *Dialogflow* and integrate it in those interfaces can thus be represented as in figure 6.5.

Figure 6.5: Integration structure

6.3.1 Custom User Interface

We built the custom UI which we integrated *JAQ* in, using *HTML/CSS* and *JavaScript*. This interface is meant to be executed on a web server and reached by users by visiting a certain web page. Building it as a web page allows to easily integrate it in another web page, such as the official website of the university. This interface can be seen on figure 6.6. A live version of this interface can be access at `http://tfe-gustin-gellens.info.ucl.ac.be` inside the network of UCLouvain.

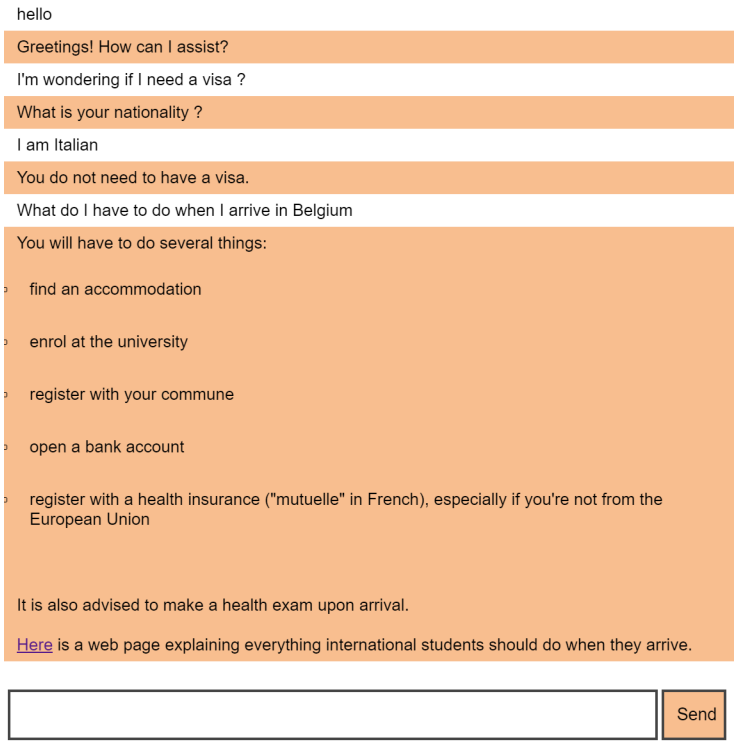


Figure 6.6: Web server interface

Internally, this interface works as follows. The user sends an HTTP request to the server to get the web page and then interacts with it using web sockets. Every time the user sends a message to the interface, the web server makes an API call which includes the user message to the *Google Dialogflow* server which hosts *JAQ*. This server handles the requests and returns the answer of the chatbot, which can then be displayed to the user. The step-by-step process to integrate a *Dialogflow*'s chatbot within this interface is explained in the section C.1.

6.3.2 *Facebook Messenger* interface

The reader might be familiar with *Facebook Messenger*, which is the messaging application included in the *Facebook* social network. It allows *Facebook* users to send each other messages in different formats such as text, images, audio recordings or files.

Integrating a chatbot into *Facebook Messenger* allows users of *Facebook* to interact with the chatbot as if it were a normal user of this social network, and this using the messaging interface. This use of *Facebook Messenger* is shown on figure 6.7.

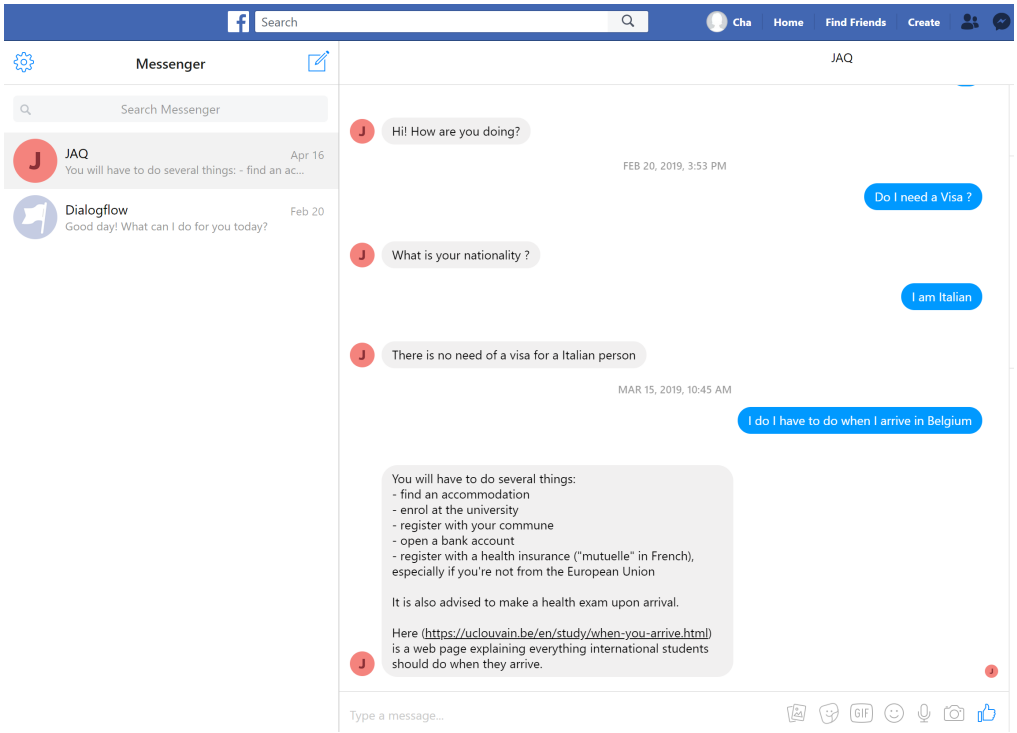


Figure 6.7: *Facebook Messenger* interface

The interactions between *Facebook Messenger* and *Dialogflow*'s servers are automatically handled, and the chatbot designers do not have anything to implement in order to get it to work. There is however a certain number of steps to follow in order for *Facebook* to accept the chatbot onto its platform and to get access to it. Those steps are described in appendix C.2.

We should mention that using *JAQ* from any *Facebook* account is not possible because it is still a "beta chatbot", which means it can only be accessed from a set of predefined accounts. To make it available to any account, we needed to follow a quite convoluted procedure which notably included sending our personal ID cards to *Facebook*. As this was not something that we were willing to do and *JAQ* was already accessible using the custom-made interface, we decided not to go through with that procedure.

Chapter 7

Experiments and analysis

This chapter describes the process we set up to assess different characteristics of the final version of JAQ. The experiments are described and the motivations for those measurements are explained.

7.1 Assessment process

When designing a software product, it is important to be able to assess its quality. Generally speaking, quality can refer to a lot of different concepts: correctness, performance, reliability, development costs, ease of use, etc. In our situation, we will mainly be interested in measuring the following characteristics:

- NLU performance
- Answering speed, i.e. response time
- User-friendliness and quality of the user experience

We made two types of tests to evaluate them: tests that could be compare to *unit tests*, and *real condition tests* with exterior people.

For the "unit tests", we simply defined a list of questions (the test set) that are inside and outside the *JAQ*'s scope and the correct intent they should be classified as. We then automatically checked that the classification was correct, i.e. that the right answers are replied, and record how many times the NLU model was correct. To make our measurement more applicable, we tried to make this test set with as few questions from the training set as possible.

Real condition tests worked as follows: we asked exterior people with different backgrounds to use the chatbot as an intended user would. We then invited them to answer a few questions about their appreciation of the system.

Note that it appears the *Loebner Prize Competition* [61], an application of the Turing test, is often employed as a criterion to assess a chatbot. However, it has been argued this did not represent a good measurement of quality [62].

7.2 NLU performance

7.2.1 Performance of the intent classification

Metrics definitions

When we talk about the quality of a chatbot, its NLU performance, i.e. its abilities of understanding messages, directly comes to mind. In machine learning tasks, we usually use metrics such as the precision and the recall to characterize their performance. However, those metrics are meant for binary classification problems. In our case, the NLU component is concerned with a multi-class classification problem. We could have measured the precision and recall for each intent, but we would have ended up with a huge number of measurements (two times the number of intents ≈ 500). We thus need different metrics.

To analyze this performance, we used the measurements we made during our "unit tests". We should note that we recorded not only whether the model classified correctly each utterance or not, but also the classification confidence returned by the model and if it labeled it as out-of-scope (i.e. not understood) or not. In table 7.1, we define a reduced number of metrics that should give us a general idea of the NLU performances.

Measurement	Formula	Evaluation focus
Accuracy rate	$\frac{tl+to}{N}$	Overall effectiveness of the classification
$\rightarrow$ True labeled rate	$\frac{tl}{N_s}$	Accuracy due to correct intent classification
$\rightarrow$ True out-of-scope rate	$\frac{to}{N_o}$	Accuracy due to correct out-of-scope classification
Error rate	$\frac{fl+fo}{N}$	Overall error rate
$\rightarrow$ False labeled rate	$\frac{fl}{fl+tl}$	Error rate due to incorrect intent classification
$\rightarrow$ False out-of-scope rate	$\frac{fo}{fo+to}$	Error rate due to out-of-scope intent classification

Table 7.1: Simple measurements to assess the NLU performance of JAQ. tl is the true (correctly) labeled, to the true out-of-scope, fl the false (incorrectly) labeled and fo the the false out-of-scope. The test set is composed of N questions, N_s of which are associated to an intent and N_o of which are out-of-scope.

Using a similar structure, we can easily analyze the confidence of the intent classification. These metrics are described in table 7.2.

Having metrics that are similar to precision and recall for the whole problem would however be interesting to have. We thus also use some of the reduced number of metrics for multi-class classification tasks from [63] and which are shown in the table 7.3.

We should note that, in those definitions, the classifier misclassifying a user message has the same impact on the measurements than if it simply does not find a match (i.e. if it considers it as out-of-scope). This is however not the case from a user perspective, for whom the first type of error will decrease the credibility of the chatbot, whereas the second one will decrease its perceived intelligence.

Measurement	Formula
Global confidence	$\frac{\sum_{i=1}^{N} c_i}{N}$
Confidence of true classification	$\frac{\sum_{i \in TL} c_i + \sum_{i \in TO} c_i}{\|\|TL\|\| + \|\|TO\|\|}$
$\rightarrow$ Confidence of true (correct) labeled	$\frac{\sum_{i \in TL} c_i}{\|\|TL\|\|}$
$\rightarrow$ Confidence of true out-of-scope	$\frac{\sum_{i \in TO} c_i}{\|\|TO\|\|}$
Confidence of false classification	$\frac{\sum_{i \in FL} c_i + \sum_{i \in FO} c_i}{\|\|FL\|\| + \|\|FO\|\|}$
$\rightarrow$ Confidence of false labeled	$\frac{\sum_{i \in FL} c_i}{\|\|FL\|\|}$
$\rightarrow$ Confidence of false out-of-scope	$\frac{\sum_{i \in FO} c_i}{\|\|FO\|\|}$

Table 7.2: Simple measurements on the confidence levels in order to assess the performance of *JAQ*. c_i is the confidence level for the classification of the question i and TL is the set containing the questions which were correctly labeled, as are respectively TO for the true out-of-scope questions, FL for the incorrectly labeled ones and FO for the incorrectly labeled out-of-scope ones.

Measure	Formula	Evaluation focus
Average Accuracy	$\frac{\sum_{i=1}^{l} \frac{tp_i + tn_i}{tp_i + fn_i + fp_i + tn_i}}{l}$	Average per-class effectiveness of the classifier
Error Rate	$\frac{\sum_{i=1}^{l} \frac{fp_i + fn_i}{tp_i + fn_i + fp_i + tn_i}}{l}$	Average per-class classification error
Precision_M	$\frac{\sum_{i=1}^{l} \frac{tp_i}{tp_i + fp_i}}{l}$	Average per-class agreement of the data class labels with those of the classifier
Recall_M	$\frac{\sum_{i=1}^{l} \frac{tp_i}{tp_i + fn_i}}{l}$	Average per-class effectiveness of the classifier to identify class labels
F1score_M	$\frac{2 \cdot Precision_M \cdot Recall_M}{Precision_M + Recall_M}$	Relation between data's positive labels and those given by the classifier based on a per-class average

Table 7.3: Metrics for multi-class classification tasks based on a generalization of precision and recall. l is the number of classes, tp_i the number of true positives for class i, fp_i its number of false positives, fn_i its number of false negatives and tn_i its number of true negatives. The M subscript represents macro-averaging.

Results

We show hereafter all the results of those experiments for *JAQ*. Table 7.4 shows the measurements defined in table 7.1; table 7.5 corresponds to table 7.2; table 7.6 to table 7.3

	labeled & out-of-scope	labeled	out-of-scope
true	0.727	0.737	0.333
false	0.273	0.269	0.500

Table 7.4: Simple measurements corresponding to metrics defined in table 7.1

When looking at the results in table 7.4, we can see that *JAQ* misunderstands about a fourth of the user messages. When we actually use it, it seems to misunderstand less often than that thanks to the fact that when the classifier misclassifies an utterance, it is usually not that far off. Indeed, upon misclassification, the chatbot often answers a question that is about the same topic.

We can also observe that it is quite rare for *JAQ* to not understand an utterance. This is due to the fact that by default, *Dialogflow* considers an intent not to be matched when the confidence in

general: 0.880	labeled & out-of-scope	labeled	out-of-scope
true	0.907	0.906	1
false	0.808	0.802	1

Table 7.5: Simple confidence level measurements corresponding to metrics defined in table 7.2

the intent classification is smaller than 0.3. Changing this value would make *JAQ* "trust" its intent classifier more, if we make the threshold smaller, and less, if we make it larger. Note that we can see on table 7.5 that the confidence levels are quite high on average, especially for true positives.

Average accuracy	0.993
Error rate	0.007
Precision_M	0.704
Recall_M	0.779
F1Score_M	0.739

Table 7.6: Measurements of metrics for multi-class classification results as described in table 7.3

If we take into account the fact that the classification problem is a multi-class problem, we get an average accuracy and error rate of the same order as what we could find in external performance studies (cf. subsubsection 4.3.3). The very high results computed for the average accuracy are due to its definition which averages the number of correct and incorrect classifications for every class. For example for 10 classes and 10 message tested, if a model always makes the wrong prediction, its average accuracy will be 90%. It is important to realize this when interpreting this measurement.

The average per-class precision and recall are in the vicinity of what we got in table 7.4. The same discussion thus applies. We can however note that the precision seems significantly smaller than the recall.

7.2.2 Performance of the entity recognition

We could do the same kind of measurements to assess the performance of the entity recognition. However, taking those measurements is way more complicated, since we cannot know whether an entity was recognized by simply inspecting *JAQ*'s answer as we did for the intent classification. Moreover, it seems that the part of the NLU component responsible for entity recognition in *Dialogflow* works by simply trying to match subsequences of utterances with examples of entity values. In other words, an entity is recognized if and only if its value matches exactly an example it was provided.

For all those reasons, we decided not to measure the performance of the entity recognition. It seems to work well enough in the context at hand. Note that some of the NLU performance studies we cited in subsection 4.3.3 measured this performance for *Dialogflow* as being quite high indeed.

7.3 Response time

Answering speed is another quality of *JAQ* that should be assessed. It is actually quite easy to measure: simply ask the chatbot a set of questions and record the time it takes to get a response. We can then use statistical tools to characterize those durations.

The response times we measure here are actually the sum of the response time and data handling of several servers. The information flows as follows: the user transmits a message to the web server that hosts the UI; this web server forwards it to a server at *Google* which is in charge of *Dialogflow*; this server runs *JAQ*'s settings on the user message and selects the chatbot answer; the answer is sent back to the UI server which displays it for the user to see. Note that we actually do not know what happens in *Google*'s data centers, it is entirely possible that more than one server is contacted to handle *Dialogflow* tasks. Nonetheless, we can consider that to be part of the handling time of the user message.

We are interested in measuring the response time of *Dialogflow* rather than the cumulated response time of *Dialogflow* and our server, since we could make specific experiments to test our server with a much larger precision. To avoid recording the response time of our server, we can either take the relevant measurements directly from the server or create a new program which uses the API of *Dialogflow*. We settled for the second option for the sake of simplicity.

Response time can be seen as a continuous-time random variable. In our analysis, we will consider the response time between subsequent questions to be independent. This eases the analysis, and should not be too far from reality, since *Google*'s servers certainly answer lots of different requests between the ones we send them.

There are a few different ways we could model the distribution of such a random variable: using a log-normal distribution, a Pareto distribution, a Gamma distribution, etc. All of them correspond to right-skewed probability distributions.

Rather than choosing one of them *a priori*, we plotted the histogram of our measurements and fitted the different distributions to see to which it corresponds best, as illustrated in the figure 7.1. To compare them we used the sum of squared errors (SSE) of the distribution as a criterion. The distribution that fits our measurements best is the Gamma distribution. It has two parameters: its shape k and rate λ, both positive real numbers. Its probability density function is:

$$f(t) = \frac{\lambda^k t^{k-1} e^{-\lambda t}}{\Gamma(k)}$$

where t corresponds to the response time. In the fitted distribution, the shape and the rate are respectively 1.86 and 0.034.

Knowing this, we can compute the mean and variance of our measurements, as well as those of the corresponding Gamma distribution. Its mean and variance are respectively computed as $\frac{k}{\lambda}$ and $\frac{k}{\lambda^2}$. The results we get are displayed in table 7.7.

We should mention that a warm-up time is necessary to have regular measurements of times. We could assume *Dialogflow* agents are not loaded on *Google* server all the time, but only when it detects they are being used. Moreover, it seems that the response time varies widely depending on the day and the hour of the day, but times measured less than a few minutes apart are quite similar.

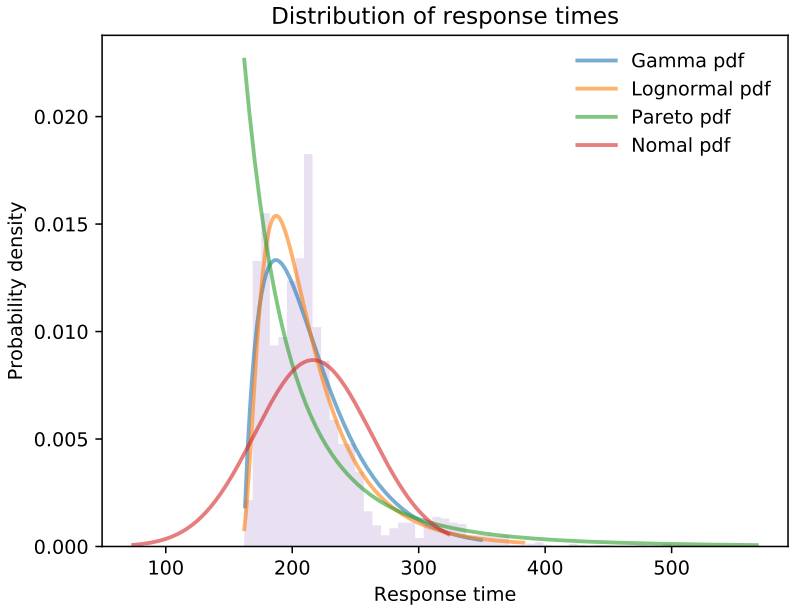


Figure 7.1: Distribution of response times (in milliseconds)

	data	model
Mean (ms)	216.42	216.42
Standard deviation (ms)	45.99	40.09

Table 7.7: Mean and standard deviation of the response time, from the measurements and the model

This experiment was made by measuring times in a short period of time, in order to get consistent measurements.

As a side note, we did not perform load tests on *Dialogflow* services, because it is not authorized by the Terms of Service and the free plan restricts us to one request per second anyway.

7.4 User experience

As we explained throughout chapters 1 and 3, one of the objective of this master thesis is to create an additional system for international students to acquire general information about the university. An important part of this was to make this system as nice to use as possible, whatever the background of the user is. We are therefore very interested in characterizing the quality of the user experience.

There are three aspects that have a large influence on user experience and user-friendliness:

- the ease-of-use of the User Interface,
- the abilities of the chatbot to understand and answer asked questions, and
- its behavior towards the user, which should not seem awkward or hard to understand.

Each of those is the responsibility of a different component of *JAQ*: its UI, NLU component and dialog management unit.

It is however quite hard to make clear-cut measurements of those notions, as they are not clearly quantifiable. A good way to quantify them is to actually ask a user to grade it on a given scale.

We thus performed tests were we selected a certain number of external people and asked them to use *JAQ*, impersonating the intended users, i.e. present and future students from abroad. We then used the *System Usability Scale (SUS)* [64] to grade the different qualities we are interested with. This test is quite standard in the industry to evaluate the usability of a given system. In this context, the tested user is asked to use the system (usually for the first time) for a given period of time without any help, and is then asked to rate 10 statement from "strongly disagree" to "strongly agree". Those ratings are typically divided in 5 steps and assigned a number (1 for "strongly disagree" and 5 for "strongly agree").

A calculation can then be done on the results to determine a score out of 100. It is important to note that this is not really a percentage, as a system that would get a score of 100 is generally not perfect from a usability viewpoint according to the tested user. This score simply allows to give an idea of how convenient and simple the system seems to this first-time users, and to compare different systems.

We can then asking several people to do that test and average aggregate their results. There exist multiple ways to aggregate such results, such as *Fleiss' kappa* [65], but we decided to simply compute the mean and median score for each question, and to compute the final score described above on those data.

The 10 questions asked in the standard test as well as in the test we performed are the following ones:

- I think that I would like to use this system frequently.
- I found the system unnecessarily complex.
- I thought the system was easy to use.
- I think that I would need the support of a technical person to be able to use this system.
- I found the various functions in this system were well integrated.
- I thought there was too much inconsistency in this system.
- I would imagine that most people would learn to use this system very quickly.
- I found the system very cumbersome to use.
- I felt very confident using the system.
- I needed to learn a lot of things before I could get going with this system.

After interrogating 11 people, we get the following average grade for each question in the same order as above:

Question number	1	2	3	4	5	6	7	8	9	10
Mean score	3,1	1,4	4,3	1,5	4	2,6	4,4	1,5	3,7	1,5
Median score	3	1	5	1	4	2,5	4,5	1	4	1

This yields a total mean score of 77.5 out of 100 and a total median score of 85. Of course, interpreting those results heavily depends on the type of system at hand. Nevertheless, knowing that the mean score for this test is 68, this result is very good, even though there is room for improvement. The system seems easy to use and understandable even to non-technical users.

Chapter 8

Discussion

In this chapter, we discuss whether the objectives were met or not, the different design choices we took and the results of the performance experiments we made. We then propose a certain number of possible improvements.

8.1 Fulfillment of the objectives

As we explained in chapter 1, this master thesis had two objectives. The first one was to explore the technical characteristics and capabilities of modern chatbots, which was done in chapters 2, 3 and 4. The second one was to design *JAQ* to be a Proof of Concept aiming to assist international students and which would be a good additional way for them to get information they are seeking.

We can thus divide this second objective into two parts. Firstly, *JAQ* being a Proof of Concept, it required to work well enough to be usable by non-technical people. Secondly, *JAQ* should be able to help international student in their specific situation.

8.1.1 Discussion of the results

To discuss whether *JAQ* is a good Proof of Concept of an assistant chatbot, we can look at the results of our different experiments we presented in chapter 7. We first determined if *JAQ* worked as intended, i.e. if it understood frequently enough the supported questions it was asked. Then, we asked non-technical people to utilize it and gauge what they thought of the usability of the system.

Performance

In terms of performance, we could argue *JAQ* works well enough to be usable: about three times out of four (cf. table 7.4), the question asked, if supported, is correctly detected and answered. Moreover, upon misclassification, *JAQ* often answers something that is related to the topic that the user would like to know about, making that misclassification less annoying than if it had given information utterly unrelated to the question at hand.

During the conception of *JAQ*, we roughly tracked the evolution of the NLU performance,

and could see that it varied. A certain number of things can make this performance vary, and consequently change the performance perceived by users.

***Dialogflow* settings** Our "unit tests" (cf. section 7.1) allow us to make comparisons of different settings and efficiently evaluate the induced effects.

The first thing that stood out during those experiments is that *Dialogflow* does not seem to have a deterministic model training. Indeed, running twice the same experiments on the same chatbot yields similar results, but training two models with the same training data and executing the same experiments produces different results. We can assume a quantization process is performed.

By comparing different configurations of *Dialogflow*, there are a few things that can be concluded:

- A small confidence threshold (0.3 compared to the default 0.6) improves the overall performances.
- Activating the *Smalltalk* module and/or the spelling correction feature does not impact performances.
- Augmenting the number of examples in the training set by only adding a question mark at the end of some examples does not improve performance.
- Increasing the number of examples in the training set never decreases performance.

Moreover, from a user perspective, we could observe that capitalizing the first letter of a message or adding a question mark at the end does not influence its classification.

High number of intents Logically, defining a lot of intents gives the NLU model more classes to classify user messages as, and thus more potential incorrect classifications. For that reason, we anticipated notably that activating the *Smalltalk* module would decrease the performances, as it would add many new intents. However, as we said, our experiments show this is not the case.

On the contrary, removing some of the intents we defined and whose examples were very close to examples of other intents improved the classification performances. Hence, a decrease in performance when adding more intents seems to come from the potential proximity between added intents and intents that were previously defined. We thus need to have a training set of good quality, which we discuss in the next paragraph. We should mention that *JAQ* being a Proof of Concept that such a chatbot could handle numerous different topics and questions, removing support for many questions was not an option.

Large number of examples As for any machine learning task, improving the training set can help overcome bad performances. Two problems can arise when building such a dataset. First, if the training data is not varied enough, there is a possibility for the model to overfit, which means the model would not generalize well and would have a tendency to misclassify valid points that are too different from the ones the training set. We should also mention that detecting overfitting in this context can be quite difficult. Secondly, as we explained earlier, having very similar examples labeled as different intents increases the likelihood of misclassifying utterances.

A training set of good quality should thus have a large enough number of varied examples for each intent in order to prevent overfitting, but examples should not be too close to each other between different intents.

Creating a large and diverse training set is quite easy to do with the help of an NLU dataset generator. Size and diversity is thus not an issue in our case. The final training set contains 243 intents and 112 examples per intent on average.

On the other hand, avoiding to have very similar examples of different intents is not a trivial task in this context: some questions are necessarily very close. For example, the sentences *"Give me the contact information of the Student Support Service"* and *"Give me the contact information of the Housing Service"* necessarily yield similar examples.

To visualize the proximity between the examples of different intents, we employed a custom-built model (as described in [66]). Each example is then represented on a scatter plot, where we only keep the two largest principal components for the classification. Such a plot is presented in figure 8.1. Even though the number of dimensions has been drastically limited, we can directly see some of the intents can easily be differentiated; other intents form clusters. Those clusters are the intents for which a misclassification is more likely.

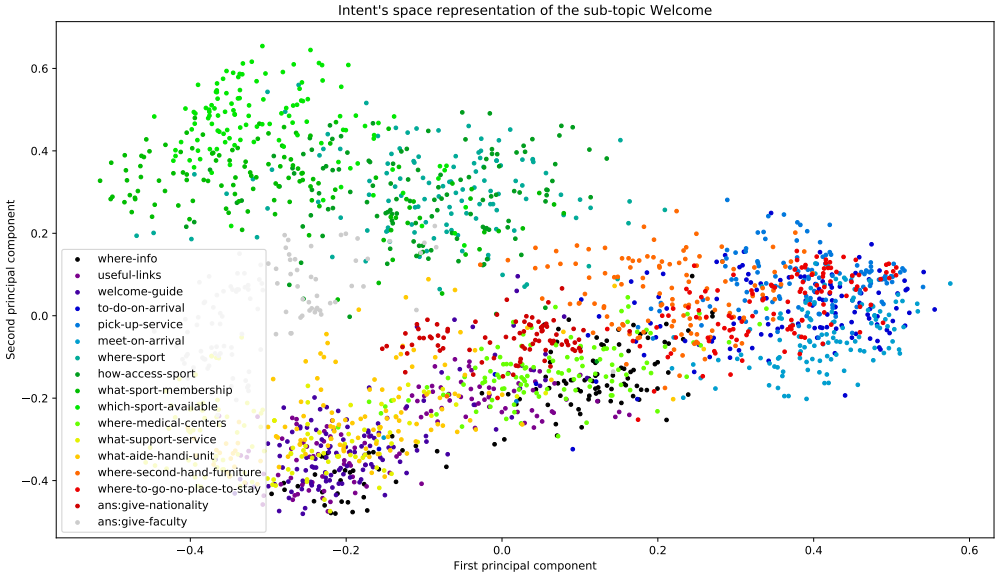


Figure 8.1: Intent's space representation

When modifying the training set, we could use our "unit tests" (cf. section 7.1) to check that our changes did not decrease the NLU performance. By being careful and executing those tests frequently, we were able to create a training set of good quality.

Typos in user messages After that, another thing that can influence the performance is the fact that users regularly make typos in their messages. Expectedly, a message with a typo is less easy for the model to correctly classify. However, during our real condition experiments, we could observe the model to be quite robust against small typos (missing a letter, replacing a letter with

another, etc). Nonetheless, there is one exception to this robustness: it seems that when two letters in a word are swapped, the classification cannot be done correctly.

Another type of typo is the omission of a word. Unless this makes the utterance to close to another intent, the model seems robust against that type of errors as well.

Confidence threshold Another observation that can be made seeing the results of these experiments is the high level of confidence returned by the model even on misclassification. Indeed, the model has a confidence of 81% on average when misclassifying and 91% when correctly classifying, and that with a high enough variance that it is not possible to differentiate them based on the confidence. In other words, changing the confidence threshold under which an utterance is considered not understood will not allow to exclude only misclassifications. Rather, increasing this threshold will exclude more incorrect classifications, but also more correct classifications, meaning the chatbot will trust its NLU component less frequently.

To determine which confidence threshold was best, we executed our "unit tests" on the same chatbot with different thresholds. As we explained earlier, switching the threshold from the default 0.6 to 0.3 allowed to improve the NLU performance the most. The experiments shown in chapter 7 were executed on a chatbot with this updated threshold. We should note that this increase is not very noticeable to the user.

It is worth mentioning that the NLU component present in *Dialogflow* seems to return a high confidence very frequently. In other words, it is often confident its prediction is correct and rarely considers the utterance as not understood (or out-of-scope). It can be argued whether it is better to have a chatbot misunderstand an utterance (and thus gives an irrelevant answer) or simply utters it did not understand. As the answer to this question is not clear, we preferred to keep the threshold that gave the best performance results.

Perceived usability

Asking exterior people to rate the usability of the system allowed us to have information about the impressions of users. The usability of a chatbot is not determined only by its ability to understand questions; we should also take into account the following notions:

- The user-friendliness;
- The enjoyment felt when interacting with the system;
- The usefulness of the system (discussed in the next subsection).

Seeing the results of the real condition experiments (cf. section 7.4), it seems that *JAQ* is perceived as usable by technical and non-technical people. Watching them use *JAQ* without interfering also proved that they seem comfortable with the system from a technical point of view: given the context that *JAQ* is a chatbot meant to help international students, they easily understand that they can type in questions and *JAQ* will answer. That being said, there are a few things worth mentioning that we could observe during those experiments.

Confusion caused by the waiting time We quickly realized during the first tests we performed that, since the response time of *Dialogflow*'s servers is quite large, users seemed to think the chatbot crashed while they were waiting for an answer. Therefore, we implemented a small change in the custom UI to indicate to the user the answer was coming up shortly. To do that, we added the common "triple dots" animated image that is frequently encountered in chat messaging apps, as a cue that the interlocutor is currently typing a message. This proved to improve the user-friendliness of the system, as users were directly aware that an answer was incoming.

Too informative messages Another peculiarity we observed some users carry out was their tendency to give the chatbot too much information. Rather than simply asking a question, those users would typically explain the whole situation they were in before asking anything, and that in just one message. Expectedly, the NLU model usually was not able to classify the message correctly, resulting in the question not being properly answered.

Seeing how *Dialogflow* is built, solving that kind of issue does not seem possible: we would need to add many examples of such messages in the training set. However, this would definitely result in having similar examples for different intents, tremendously decreasing the NLU performance, and thus the quality of the user experience as a consequence. Note that this is not a behavior that all people who interacted with *JAQ* had, and it does not seem correlated with their frequency of usage of chat messaging applications. Nonetheless, we can expect some of the international students who would use *JAQ* to behave this way as well.

Usability with non-English speakers During those experiments, we could finally observe how well *JAQ* worked when users spoke a poor English. This is extremely relevant in our case, since *JAQ* is meant to be used by people from abroad, whose native language might not be English. Having a chatbot that behaves well in this sort of situations is therefore important.

We could observe that, when a user message contained a typo or when the grammar was slightly off, the NLU component could still classify the intents with the same performance as before. However, when the question was phrased in an utterly incorrect way, the chatbot would either not understand it or completely misclassify it. In other words, the poorer the quality of the English in the user messages, the worse the user experience was. From a user-friendliness perspective, *JAQ* is thus able to cope with users who spoke a bad English, but not with users whose English is extremely poor.

8.1.2 Usefulness of the chatbot

Now that we know *JAQ* performs well enough to be easily usable by the intended users, we can discuss why a chatbot would be a good solution to assist international students.

We explained in subsections 2.2.2 and 2.2.3 why a chatbot is a good solution for a system that assists users and answers their questions, but also why it cannot be used as the sole solution available. As a reminder, a chatbot can be used easily by non-technical people, provides its user only with information that is relevant to them and easily scales as the number of users grows, but it is quite slow to use as an interface, can be awkward to interact with and could potentially give out-of-date information. *JAQ* being a chatbot, it has those qualities and limitations as well, and can therefore be a good solution to assist users, but not replace all the existing solutions.

We can now argue whether using a chatbot in this specific context is appropriate. In other words, is a chatbot suitable to help international students and future students from abroad before coming in UCLouvain? After having designed *JAQ* from the bottom up, we believe that we showed it is the case: it is usually able to answer the most prominent questions that those people have, and is quite easy and pleasant to use. We explained how we chose which questions to support in section 5.4.

8.2 Possible improvements

As can be expected, we could consider the design of a chatbot to never end, as it can always be improved: more questions can be supported, more conversational flows can be defined, more examples of utterances and entity values can be specified, etc. We present hereafter the most important improvements, both to enhance the latest version of *JAQ* and the framework we employed to implement it.

8.2.1 Improvements of *JAQ*

If we needed to work on *JAQ* for a few additional months, here is what we would improve in priority.

Expand the complex interactions

First, we would develop the complex conversation flows. As we explained in section 5.6, complex interactions rely on conditions on entity values. However, with the interactions *JAQ* supports, we did not need to support conditions on more than one entity value. A good improvement would thus be to make those interactions capable of supporting several "slots" to fill.

Add support for poorly represented topics

Then, we would add support for a certain number of currently unsupported questions, notably on the topics of study programmes and faculties. It would indeed be extremely useful to international students to know what programmes are available to them at UCLouvain, or what the specificities of a given faculty are.

Those are topics that *JAQ* does not support, because it required to define a very large number of additional questions whose syntactic structure is close to one another (meaning, *JAQ* would likely confuse them). Moreover, it is harder to obtain correct and complete information about those topics. We should note however that *JAQ* is able to answer quite a few questions about contact information of faculties, answer several questions about the *LSM* faculty and to give a hyperlink to the list of programmes for this year.

Add support for more languages

Another improvement we could make, which would take a long time and large efforts would be to support more languages than just English. We chose English as it was the most relevant language for international students, but adding support at least for the official language of UCLouvain, French, would be an interesting addition to *JAQ*. Supporting even more languages would also be a good thing.

However, we would then have a design choice to make: should the new language be added inside *JAQ* itself, making it able to switch language during a conversation? Should we make a "version" of *JAQ* for each language, making the users need to choose which version they want to talk to? Should we make a special system with a component that is just meant to detect the language of a user message and forward it to the right chatbot? Is *Google Translate* or a similar service efficient enough to simply use it to translate incoming user messages and outgoing chatbot messages? Of course, all those solutions have their qualities and drawbacks, and would require further investigation.

Improve the maintenance process

If we needed to support *JAQ* for more than a few months, we would certainly need to automate the process of updating answers. Indeed, the information *JAQ* provides can change, especially from one quadrimester to the next. Having an automated way to easily update a large number of answers, removing support for now irrelevant questions and adding support for new questions would be a tremendous help for maintenance tasks.

We did think of that when implementing our scripts, which is one of the reasons we stored the questions, answers and complex interactions in human-readable files. This way changing an answer, adding or removing support for a question or adding a complex interaction is easy and can be done directly with a few changes to the files that store these data. This being said, there are a certain number of things that could improved or created to ease the maintenance processes, such as a way to find all the answers that contain a piece of information.

Write several answers for each question

As we explained in section 4.5, a specificity of *Dialogflow* in comparison to other frameworks is its ability to associate different answers to one conditional block. In other words, its dialog management unit can choose which answer associated to the detected intent to utter at random. This allows the chatbot to feel less robotic, as it does not repeat the same answer if it gets asked the same question several times in a row.

However, the scripts we implemented to create an workspace importable in *Dialogflow* do not support this feature. All the questions that are present in the data files are associated with just one answer. Adding support for several answers in those data files and the scripts that handle them should not be very complicated and would improve the user experience. That being said, writing the new answers would certainly be tedious and time-consuming.

8.2.2 Improvements by changing the framework

If we could make some changes to improve the framework we used, there are a certain number of things we would do. Indeed, *Dialogflow* has a few limitations and drawbacks that would be worth trying to mitigate.

We also explained in chapter 6 that *Dialogflow* is not really meant to produce chatbots of the size of *JAQ*, even though we proved it is possible. Consequently, it can be expected that it lacks a certain number of features that would be useful in situations where a lot of conditional blocks, intents and entities are defined, and the chatbot is supposed to be able to have quite long conversations with the user.

Hierarchical intents

Something that could be useful when designing a chatbot would be to define categories of intents, rather than the intents themselves. Those categories would be akin the dialog acts we addressed in subsection 3.1.3. We could even imagine a hierarchy of categories, as a tree whose leaves are the intents and the different branches are the categories.

Then, being able to specify conditions on categories of intents rather than on the intent itself could be extremely useful in some situations. It would for example allow the designers to specify an answer to give in a certain context for all user messages that are not related to the topic conversed about at that moment.

Single intent matching

As we explained in subsection 2.3.3, the usual approach to NLU does not allow to match a user message with several intents. Consequently, if the user asks two questions in just one utterance (e.g. *When and how can I register into the university?*), just one of them will be detected and answered.

It is not easy to solve that kind of issue. One approach could be to define new intents that are combinations of existing intents, and to provide specific answers for those combinations. This would however be extremely tedious to add, as there would be an overwhelmingly large number of new intents to define.

A solution would thus be to modify the framework in a way that lets the NLU component match a user message with several intents. Then, we could imagine letting the designer specify an answer for supported combinations of intents and a fallback answer for all other combinations. This approach could even partly solve the problem that occurs when users give too much information in a message.

Automatic support for grounding

We defined grounding in subsection 3.1.4 and explained why it was so hard to automatically manage grounding in the answers of the chatbot. That being said, there could be ways to ease this handling (at least if we take only English into account). For example, there could be a way to ask the chatbot to add a grounding "marker" at the beginning of its answer, which would depend on the

way the question was asked. For example, for the same questions asked in two different ways, we could have:

> *Student:* Can you tell me how expensive the registration is?
> *JAQ:* Yes, I can. Can you first tell me your nationality?

> *Student:* Tell me how expensive the registration is, please.
> *JAQ:* Okay. Can you first tell me your nationality?

In a lot of cases, those few words could be predefined and automatically prepended to the answer.

Access to confidence levels

A feature that we felt was lacking in *Dialogflow* is the access to the confidence levels of the intent classification and the entity recognition models. We can only have conditions on matched intents, active contexts and entity values. It would have been useful to be able to make conditions also on the confidence levels. For example, rather than having utterances either understood or not, we could have the chatbot ask for confirmation if the confidence in the classification was in a certain range. We could imagine this behavior to be predefined by the designer.

Additional units in NLU component

As we said when we explained the general architecture of modern chatbot systems (cf. subsection 2.3.3), the NLU component of chatbot is usually composed of an intent classifier and an entity recognizer, but it need not be limited to this. Adding components that implement features such as sentiment analysis or phrase chunking could be very useful. For instance, knowing the current feeling of the user could allow the chatbot to behave differently when the user gets annoyed. Having additional information about utterances can always be good from a design perspective: it allows the designer to build dialog management on much finer aspects of the user messages.

User thumbprint

An additional feature that would have been very useful in the case of *JAQ* is to have some kind of support for a "user thumbprint", which would be a collection of information the chatbot gathered about the user during conversation. This feature would allow the chatbot to reuse this information later in the conversation. We could achieve a similar result by using contexts to store those values, but this would be very tedious to do.

That being said, our solution allowed the chatbot to forget some information once it was not relevant anymore, meaning the user could ask the same question for a different situation. For example, they could want to know the price of an apartment for disabled people, and the price of an apartment for people with no disabilities. Having a "user thumbprint" would require giving the user a way to seamlessly change their thumbprint.

Note that we are not talking here about a thumbprint to identify a user in different sessions, but rather a list of information we know about the user and that is restricted to the current conversation.

Chatbot network

Finally, as mentioned several times, it is not easy to cover a large scope of topics. A solution would be to create a chatbot per topic and organize them in a network. We would then have a prior classification mechanism that could decide which chatbot should answer the current user message.

Note that this would also allow to easily add support for a predefined topic, since it only requires to add a chatbot in the network. We could conceptualize having a whole library of predefined topics available to designers and that could easily be added to any chatbot.

Chapter 9

Conclusion

In this master thesis, we were interested in studying the modern approach to building chatbot systems, and to creating such a chatbot to assist international (present and future) students of UCLouvain. This chatbot, named *JAQ*, is intended to provide an additional way to obtain general information about the university. The difference with previous works was the large size of the topic it was supposed to cover.

We defined a chatbot as a program that is able to have a conversation in natural language with a human user. We specifically investigated the modern solutions that were text-based and supposed to be employed by a single person in a conversation. Modern chatbots usually all exploit a specific architecture which was described and discussed. The most prominent challenges to overcome were also described.

Several frameworks are popular to design modern chatbots. We looked further into two of them, *IBM Watson Assistant* and *Google Dialogflow*, and finally decided to employ *Dialogflow* to create *JAQ*. Other tools to help the conception were presented as well. Additionally, to ease the development of a chatbot with such a large scope, we implemented a whole system on top of *Dialogflow*. This system also improved maintainability.

Once the final version of *JAQ* was ready, we conceived several experiments to assess its performance and usability. We proved that the chatbot behaved as intended about 3/4 of the time, and that users perceived it as quite enjoyable and user-friendly, even though the user experience is undeniably not optimal. Nevertheless, users experienced sub-optimal behaviors because the free plan of *Dialogflow* made it quite slow, but not at a point of negatively impacting the convenience and user experience.

We then discussed the different aspects that impacted its performance and the user experience. We came to the conclusion that supporting a large scope makes it difficult to create a training set of good quality, which decreases the performance of the produced chatbot and consequently the user experience. Moreover, users sometimes also showed a certain number of unanticipated behaviors, which was detrimental to the perceived quality of the chatbot. To conclude, we can say *JAQ* seems functional for its intended purpose, and fulfills its role of a Proof of Concept.

Finally, although the produced chatbot was already operational and help students querying information, several improvements were proposed, both to enhance the latest version of the produced chatbot and the employed framework.

Bibliography

[1] J. H. Martin and D. Jurafsky, *Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition*. Pearson/Prentice Hall Upper Saddle River, 2009.

[2] UCLouvain, "Université catholique de louvain." `https://uclouvain.be`, consulted in October 2018.

[3] J. Lyons, *Natural Language and Universal Grammar: Volume 1: Essays in Linguistic Theory*, vol. 1. Cambridge University Press, 1991.

[4] D. Premack, ""gavagai!" or the future history of the animal language controversy," *Cognition*, vol. 19, no. 3, pp. 207–296, 1985.

[5] M. Warren, *Features of naturalness in conversation*, vol. 152. John Benjamins Publishing, 2006.

[6] E. System, "Chatbot: What is chatbot? why are chatbots important?." `https://www.expertsystem.com/chatbot/`, consulted in May 2019.

[7] A. M. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, no. 236, p. 433, 1950.

[8] R. Carpenter, "Cleverbot." `https://www.cleverbot.com/`, consulted in April 2019.

[9] Google, "Google assistant." `https://assistant.google.com`, consulted in June 2019.

[10] Amazon, "Alexa." `https://developer.amazon.com/fr/alexa`, consulted in June 2019.

[11] B. Laurel, "Interface agents: Metaphors with character," *Human Values and the design of Computer Technology*, pp. 207–219, 1997.

[12] IBM, "Watson - personality insights." `https://www.ibm.com/watson/services/personality-insights/`, consulted in April 2019.

[13] K. Nigam, J. Lafferty, and A. McCallum, "Using maximum entropy for text classification," in *IJCAI-99 workshop on machine learning for information filtering*, vol. 1, pp. 61–67, 1999.

[14] S. Ravuri and A. Stoicke, "A comparative study of neural network models for lexical intent classification," in *2015 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU)*, pp. 368–374, IEEE, 2015.

[15] S. M. Beitzel, E. C. Jensen, O. Frieder, D. D. Lewis, A. Chowdhury, and A. Kolcz, "Improving automatic query classification via semi-supervised learning," in *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pp. 8–pp, IEEE, 2005.

[16] R. Pieraccini, E. Levin, and C.-H. Lee, "Stochastic representation of conceptual structure in the atis task," in *Speech and Natural Language: Proceedings of a Workshop Held at Pacific Grove, California, February 19-22, 1991*, 1991.

[17] J. Ramos, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, pp. 133–142, Piscataway, NJ, 2003.

[18] G. Ngai and D. Yarowsky, "Rule writing or annotation: Cost-efficient resource usage for base noun phrase chunking," in *Proceedings of the 38th Annual Meeting on Association for Computational Linguistics*, pp. 117–125, Association for Computational Linguistics, 2000.

[19] R. Wilensky, "Planning and understanding: A computational approach to human reasoning," 1983.

[20] A. Abella and A. L. Gorin, "Method for dialog management," Dec. 3 2013. US Patent 8,600,747.

[21] T. McCross, "Dialog management." `https://tutorials.botsfloor.com/dialog-management-799c20a39aad`, consulted in October 2018.

[22] E. Tann, "Abot." `https://github.com/itsabot/itsabot/`, consulted in April 2019.

[23] A. Allen, "alexafsm." `https://github.com/allenai/alexafsm`, consulted in April 2019.

[24] R. Bellman *et al.*, "The theory of dynamic programming," *Bulletin of the American Mathematical Society*, vol. 60, no. 6, pp. 503–515, 1954.

[25] A. K. Dixit, J. J. Sherrerd, *et al.*, *Optimization in economic theory*. Oxford University Press on Demand, 1990.

[26] A. Nichol, "A new approach to conversational software." `https://medium.com/rasa-blog/a-new-approach-to-conversational-software-2e64a5d05f2a`, consulted in April 2019.

[27] S. C. Levinson, *Pragmatics*. Cambridge University Press, 1983.

[28] R. San-Segundo, B. Pellom, K. Hacioglu, W. Ward, and J. M. Pardo, "Confidence measures for spoken dialogue systems," in *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings (Cat. No. 01CH37221)*, vol. 1, pp. 393–396, IEEE, 2001.

[29] L. Wittgenstein, *Philosophical investigations*. Basil Blackwell, 1953.

[30] J. L. Austin, *How to do things with words*. Oxford university press, 1962.

[31] J. R. Searle, "A taxonomy of illocutionary acts," 1975.

[32] H. Bunt, "Context and dialogue control," *Think Quarterly*, vol. 3, no. 1, pp. 19–31, 1994.

[33] J. Carletta, A. Isard, S. Isard, J. C. Kowtko, G. Doherty-Sneddon, and A. H. Anderson, "The reliability of a dialogue structure coding scheme.," 1997.

[34] D. A. Norman, *The design of everyday things*. Basic books, 1988.

[35] H. H. Clark and E. F. Schaefer, "Contributing to discourse," *Cognitive science*, vol. 13, no. 2, pp. 259–294, 1989.

[36] R. Pimentel, "Chatito." `https://github.com/rodrigopivi/Chatito`, consulted in April 2019.

[37] S. Gustin, "Chatette." `https://github.com/SimGus/Chatette`, consulted in April 2019.

[38] J. Leicher, "Chatl." `https://github.com/atlassistant/chatl`, consulted in April 2019.

[39] M. Buck, "Expando." `https://github.com/voxable-labs/expando`, consulted in April 2019.

[40] K. Galaxy, "Tracery." `http://tracery.io/`, consulted in April 2019.

[41] Facebook, "Wit.ai." `https://wit.ai/`, consulted in April 2019.

[42] Microsoft, "Language understanding (luis)." `https://www.luis.ai/home`, consulted in April 2019.

[43] Snips, "snips." `https://snips.ai/`, consulted in April 2019.

[44] Ambiverse, "Ambiverse natural language understanding." `https://github.com/ambiverse-nlu/ambiverse-nlu`, consulted in April 2019.

[45] Google, "Build natural and rich conversational experiences." `https://dialogflow.com/`, consulted in October 2018.

[46] Amazon, "Amazon lex." `https://aws.amazon.com/fr/lex/`, consulted in October 2018.

[47] IBM, "Ibm watson." `https://www.ibm.com/watson`, consulted in October 2018.

[48] Intento, "Nlu / intent detection benchmark." `https://fr.slideshare.net/KonstantinSavenkov/nlu-intent-detection-benchmark-by-intento-august-2017`, consulted in October 2018.

[49] Snips, "Benchmarking natural language understanding systems: Google, facebook, microsoft, amazon, and snips." `https://medium.com/snips-ai/benchmarking-natural-language-understanding-systems-google-facebook-microsoft-and-snips-2b8ddcf9fb19`, consulted in October 2018.

[50] N. T. Canh, "Benchmarking intent classification services." `https://medium.com/botfuel/benchmarking-intent-classification-services-june-2018-eb8684a1e55f`, consulted in October 2018.

[51] R. Koplowitz and M. Facemire, "The forrester new wave™: Conversational computing platforms," Apr. 12 2018.

[52] IBM, "Ibm watson assistant - pricing." https://www.ibm.com/cloud/watson-assistant/pricing/, consulted in October 2018.

[53] IBM, "Ibm data responsibility perspective for watson data and ai." https://www.ibm.com/watson/data-privacy/, consulted in April 2018.

[54] Google Dialogflow team, "Integrations." https://dialogflow.com/docs/integrations, consulted in October 2018.

[55] Google, "Dialogflow - pricing." https://dialogflow.com/pricing, consulted in October 2018.

[56] IBM, "Google cloud platform conditions d'utilisation." https://cloud.google.com/terms/, consulted in April 2018.

[57] R. Moen and C. Norman, "Evolution of the pdca cycle," 2006.

[58] GitHub, "Mastering markdown." https://guides.github.com/features/mastering-markdown/, consulted in May 2019.

[59] M. Bayer, "Mako templates for python." https://www.makotemplates.org/, consulted in February 2018.

[60] S. Gustin, "Chatette - syntax specifications." https://github.com/SimGus/Chatette/blob/master/syntax-specs.md, consulted in August 2018.

[61] AISB, "Loebner prize." http://aisb.org.uk/events/loebner-prize, consulted in June 2019.

[62] B. A. Shawar and E. Atwell, "Different measurements metrics to evaluate a chatbot system," in *Proceedings of the workshop on bridging the gap: Academic and industrial research in dialog technologies*, pp. 89–96, Association for Computational Linguistics, 2007.

[63] M. Sokolova and G. Lapalme, "A systematic analysis of performance measures for classification tasks," *Information Processing & Management*, vol. 45, no. 4, pp. 427–437, 2009.

[64] J. Brooke *et al.*, "Sus-a quick and dirty usability scale," *Usability evaluation in industry*, vol. 189, no. 194, pp. 4–7, 1996.

[65] J. L. Fleiss, "Measuring nominal scale agreement among many raters.," *Psychological bulletin*, vol. 76, no. 5, p. 378, 1971.

[66] Google, "Assessing the quality of training phrases in dialogflow intents." https://cloud.google.com/solutions/assessing-the-quality-of-training-phrases-in-dialogflow-intents, consulted in May 2019.

[67] Chatbots magazine, "A visual history of chatbots." https://chatbotsmagazine.com/a-visual-history-of-chatbots-8bf3b31dbfb2, consulted in May 2019.

[68] Futurism, "The history of chatbots." https://futurism.com/images/the-history-of-chatbots-infographic, consulted in May 2019.

[69] J. Weizenbaum, "Computer power and human reason: From judgment to calculation.," 1976.

[70] K. M. Colby, "Ten criticisms of parry," *ACM SIGART Bulletin*, no. 48, pp. 5–9, 1974.

[71] A. P. Saygin, I. Cicekli, and V. Akman, "Turing test: 50 years later," *Minds and machines*, vol. 10, no. 4, pp. 463–518, 2000.

[72] D. G. Bobrow, R. M. Kaplan, M. Kay, D. A. Norman, H. Thompson, and T. Winograd, "Gus, a frame-driven dialog system," *Artificial intelligence*, vol. 8, no. 2, pp. 155–173, 1977.

[73] H. Reese, "Why microsoft's 'tay' ai bot went wrong." `https://www.techrepublic.com/article/why-microsofts-tay-ai-bot-went-wrong/`, consulted in June 2019.

[74] Google Dialogflow team, "Setting up authentication." `https://dialogflow.com/docs/reference/v2-auth-setup`, consulted in February 2019.

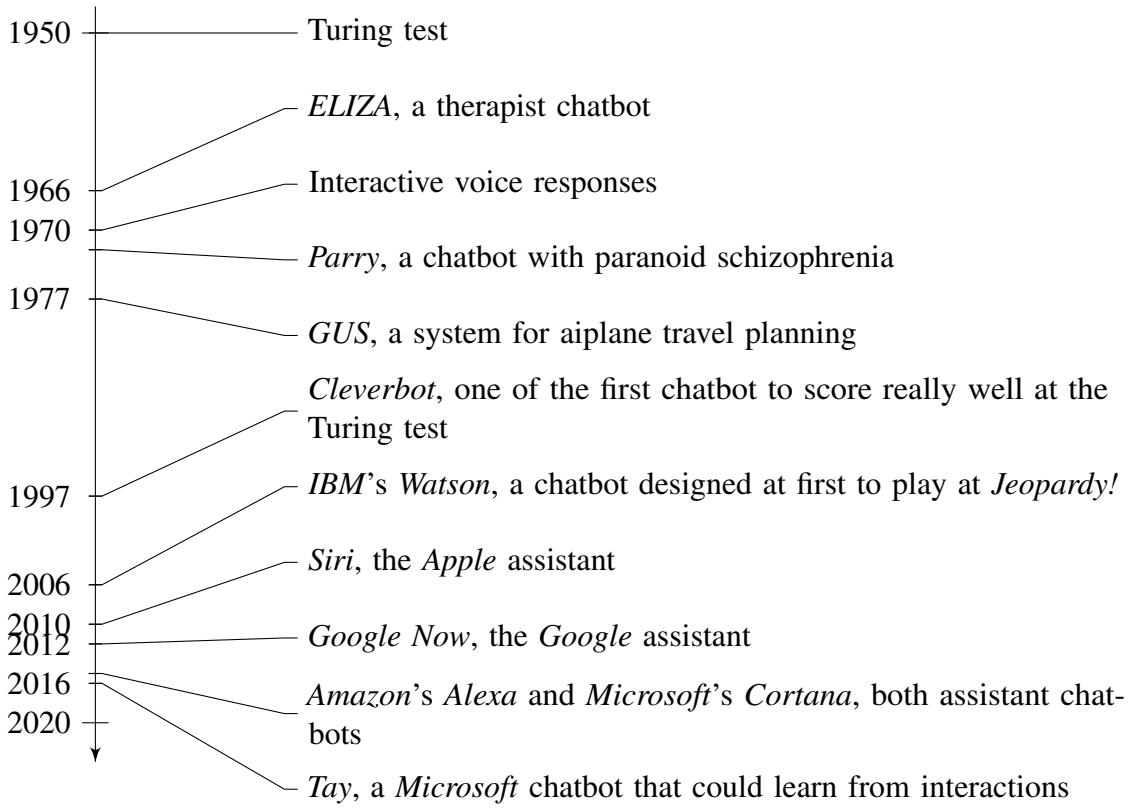
[75] Google Dialogflow team, "Facebook messenger integration in dialogflow." `https://dialogflow.com/docs/integrations/facebook`, consulted in February 2019.

Appendix A

Chatbot history

This appendix will paint a chronological overview of the history of chatbots. The point is not to be exhaustive about all the systems that have existed but rather to pinpoint different systems that were influential at the time they were released [1] [67] [68].

As can be expected, computer scientists have started very early to try and make a computer talk. While a lot of the early systems made for that purpose didn't really qualify as chatbots, a certain number of them were already quite close to the concept we defined in this dissertation. Moreover, since Turing created in 1950 his Turing test as a criterion to test whether a computer program was at the same level of intelligence of a human, many people tried (and often failed) to pass that test [7].



ELIZA The first chatbot we can cite is most definitely *ELIZA*, a chatbot that simulated a Rogerian psychotherapist and worked by using simple rules and pattern matching to detect sentences [69]. This program was written by Joseph Weizenbaum from MIT between 1946 and 1955 and was one of the first attempts at passing the Turing test.

Interactive voice responses Then, starting from the 1970s, a certain number of automatic systems to answer the phone started being created. Those systems are usually called *interactive voice response* and typically works by playing prerecorded audio messages and by asking the user to press buttons on their phone to express their needs. This kind of system is very useful to large companies which want to provide customer support by phone without hiring numerous employees for that, and is therefore quite ubiquitous nowadays.

However, as we explained in subsection 2.2.1, there have been attempts to make those systems less cumbersome to use by adding speech recognition capabilities to them. It seems this kind of systems can be frequently encountered in the USA starting from the 1990s. However, people who employed those system can testify they were quite cumbersome to interact with because they often failed to understand the user utterances and only supported a handful of messages.

Parry After that, a chatbot that is often described as a more advanced than *ELIZA* is *Parry*, written by Kenneth Colby from Stanford University [70]. This chatbot was meant to simulate a patient with paranoid schizophrenia. Contrarily to *ELIZA*, it appears it got very good results to a variation of the Turing test: doctors were able to make the difference between this chatbot and a real patient only 48% of the time [71].

During the thirty following years, many computer scientists, linguists and enthusiasts created chatbots for different purposes with increasing capabilities and results. We can cite the very influential *GUS* system for airplane travel planning [72] in 1977 or *Cleverbot* [8] in 1997, one of the first chatbot to consistently make very good scores at the Turing test.

From 2006 onwards, more and more chatbots were created by large companies and intended to be used in everyday life. We explained in section 4.4 the history of *Watson* and how it participated and won the TV show *Jeopardy!*

Siri and other phone assistant chatbots In 2010, *Apple* released *Siri*, an assistant chatbot for *Iphone's*. This chatbot was the first of a recent trend of phone assistant chatbots. We can notably cite *Google Now*, the assistant for *Android* phones and ancestor of *Google Assistant*, which is now built using *Dialogflow*, *Cortana*, the assistant for computers running *Windows* or *Amazon*'s *Alexa*.

Tay A recent experiment we can cite is certainly *Microsoft*'s *Tay*, a chatbot meant to run on the *Twitter* social network and learn from its interactions with other users. We can say this experiment was a failure, as *Tay* was shut down after 16 hours, because it had learned many provocative statements from other users and was repeating them [73].

Appendix B

Source of the information gathered for *JAQ*

In this appendix, we will list the sources used to feed our chatbot. Those were accessed between October 2018 and April 2019.

Accommodation

- Apply to LSM: `https://uclouvain.be/en/faculties/lsm/apply-to-lsm.html`
- Logistics & Accommodation Service: `https://uclouvain.be/en/study/accomodation`
- International exchange students - FAQs: `https://uclouvain.be/en/study/questions-frequemment-posees-0.html`
- International Welcome Guide - Practical Information: `https://uclouvain.be/en/study/practical-information.html`
- Logistics & Accommodation Service: FAQ: `https://uclouvain.be/en/study/accomodation/faq-frequently-asked-questions.html`
- On-line accommodation request: `https://uclouvain.be/en/study/accomodation/on-line-accomodation-request-0-1.html`
- Logistics & Accommodation Service - International student: `https://uclouvain.be/en/study/accomodation/international-student-registered-for-a-complete-study-year-or-complete-degree-0.html`
- Logistics & Accommodation Service - When I arrive in my accommodation: `https://uclouvain.be/en/study/accomodation/when-i-arrive-in-my-accommodation.html`
- Logistics & Accommodation Service - Before leaving my accommodation: `https://uclouvain.be/en/study/accomodation/when-i-leave-my-accommodation.html`
- Associations étudiantes - Représentation étudiante: `https://uclouvain.be/fr/etudier/representation-etudiante-agl.html`

Belgium

- LSM - FAQ: https://uclouvain.be/en/faculties/lsm/faq.html
- International Welcome Guide - When you arrive: https://uclouvain.be/en/study/when-you-arrive.html
- Apply to LSM: https://uclouvain.be/en/faculties/lsm/apply-to-lsm.html
- International exchange students - FAQs: https://uclouvain.be/en/study/questions-frequemment-posees-0.html
- International Welcome Guide - Administrative Information: https://uclouvain.be/en/study/administrative-information.html
- International bachelor's, master's and doctoral degree students - FAQs: https://uclouvain.be/en/study/questions-frequemment-posees.html
- International Welcome Guide - University Life: https://uclouvain.be/en/study/university-life.html
- International Welcome Guide - Practical Information: https://uclouvain.be/en/study/practical-information.html
- LSM - Upon arrival: https://uclouvain.be/en/faculties/lsm/upon-arrival.html
- International bachelor's, master's and doctoral degree students: https://uclouvain.be/en/study/international-bachelor-master-doctorate.html
- Course catalogue - Bachelor: https://uclouvain.be/en/study/bacheliers.html
- Course catalogue Master: https://uclouvain.be/en/study/masters.html
- Course catalogue - Advanced master: https://uclouvain.be/en/study/masters-de-specialisation.html
- One year at UCL—how much does it cost?: https://uclouvain.be/en/study/combien-coute-une-annee-d-rsquo-etude.html
- One year at UCL—how much does it cost? - Books and materials: https://uclouvain.be/en/study/livres-et-materiel.html
- One year at UCL—how much does it cost? - Food and living expenses: https://uclouvain.be/en/study/alimentation-et-vie-courante.html
- One year at UCL—how much does it cost? - Housing: https://uclouvain.be/en/study/logement.html
- One year at UCL—how much does it cost? - Social life, culture and sport: https://uclouvain.be/en/study/vie-sociale-culturelle-et-sportive.html
- One year at UCL—how much does it cost? - Travel: https://uclouvain.be/en/study/deplacements.html
- One year at UCL—how much does it cost? - Tuition fees: https://uclouvain.be/en/study/droits-d-rsquo-inscription.html

Contact

- UCLouvain student exchange coordinators: https://uclouvain.be/en/study/les-coordonnateurs-de-mobilite-a-l-ucl.html
- LSM - FAQ: https://uclouvain.be/en/faculties/lsm/faq.html
- UCL on social media: https://uclouvain.be/en/discover/ucl-on-social-media.html
- La vie étudiante - Associations étudiantes: https://uclouvain.be/fr/etudier/associations-etudiantes.html
- Logistics & Accommodation Service: FAQ: https://uclouvain.be/en/study/accomodation/faq-frequently-asked-questions.html
- Kot Erasmus website: http://koterasmus.be
- Contacts et informations pratiques: https://uclouvain.be/fr/etudier/aide/aide.html

ECTS

- LSM - Courses & Exams: https://uclouvain.be/en/faculties/lsm/courses-exams.html
- Managing your study programme - ECTS Credits: https://uclouvain.be/en/study/les-credits-ects.html
- LSM - FAQ: https://uclouvain.be/en/faculties/lsm/faq.html
- Apply to LSM: https://uclouvain.be/en/faculties/lsm/apply-to-lsm.html

Exams & grades

- LSM - Courses & Exams: https://uclouvain.be/en/faculties/lsm/courses-exams.html
- LSM - FAQ: https://uclouvain.be/en/faculties/lsm/faq.html
- Managing your study programme - Grades: https://uclouvain.be/en/study/le-releve-de-notes.html
- Apply to LSM: https://uclouvain.be/en/faculties/lsm/apply-to-lsm.html

Faculties

This part has not been explored due to its very large size.

- Taking classes - Faculties: https://uclouvain.be/en/study/les-facultes.html

Language

- Apply to LSM: https://uclouvain.be/en/faculties/lsm/apply-to-lsm.html
- LSM - Courses & Exams: https://uclouvain.be/en/faculties/lsm/courses-exams.html
- International exchange students - FAQs: https://uclouvain.be/en/study/questions-frequemment-posees-0.html
- International bachelor's, master's and doctoral degree students - FAQs: https://uclouvain.be/en/study/questions-frequemment-posees.html
- Language Courses - French: https://uclouvain.be/en/study/ilv/french.html
- Learning languages: https://uclouvain.be/en/study/apprendre-les-langues.html
- LSM - Upon arrival: https://uclouvain.be/en/faculties/lsm/upon-arrival.html
- ILV - Information sessions: https://uclouvain.be/en/study/ilv/information-sessions.html
- International bachelor's, master's and doctoral degree students: https://uclouvain.be/en/study/international-bachelor-master-doctorate.html
- ILV - Placement test: https://uclouvain.be/en/study/ilv/tests-de-placement.html
- ILV - FAQ: https://uclouvain.be/en/study/ilv/frequently-asked-questions-faq.html
- ILV - First Steps: https://uclouvain.be/en/study/ilv/first-steps.html

Registration

- LSM - Courses & Exams: https://uclouvain.be/en/faculties/lsm/courses-exams.html
- Apply to LSM: https://uclouvain.be/en/faculties/lsm/apply-to-lsm.html
- International bachelor's, master's and doctoral degree students - FAQs: https://uclouvain.be/en/study/questions-frequemment-posees.html
- Admission and enrolment procedure for exchange students: https://uclouvain.be/en/study/procedure-d-inscription-pour-etudiant-middot-e-d-echange.html
- International Welcome Guide - When you arrive: https://uclouvain.be/en/study/when-you-arrive.html
- International bachelor's, master's and doctoral degree students - Enrolment: https://uclouvain.be/en/study/s-inscrire-0.html
- International Welcome Guide - Administrative Information: https://uclouvain.be/en/study/administrative-information.html
- LSM - Upon arrival: https://uclouvain.be/en/faculties/lsm/upon-arrival.html

Schedule & calendar

- International bachelor's, master's and doctoral degree students: https://uclouvain.be/en/study/international-bachelor-master-doctorate.html
- International Welcome Guide - University Life: https://uclouvain.be/en/study/university-life.html
- Academic calendar: https://uclouvain.be/en/study/calendrier-academique-0.html
- International exchange students - FAQs: https://uclouvain.be/en/study/questions-frequemment-posees-0.html
- LSM - Courses & Exams: https://uclouvain.be/en/faculties/lsm/courses-exams.html
- LSM - FAQ: https://uclouvain.be/en/faculties/lsm/faq.html

Study organization

- LSM - Courses & Exams: https://uclouvain.be/en/faculties/lsm/courses-exams.html
- Course catalogue: https://uclouvain.be/en/study/le-catalogue-de-formations.html
- Bachelor - Minors: https://uclouvain.be/en/study/mineures.html

UCLouvain presentation

- Get to know UCL: https://uclouvain.be/en/discover/connaitre-l-ucl.html
- The University in figures: https://uclouvain.be/en/discover/faits-et-chiffres.html
- Get to know UCL - Missions, vision, values: https://uclouvain.be/en/discover/missions-vision-valeurs.html
- Get to know UCL - History: https://uclouvain.be/en/discover/history.html
- International Welcome Guide - University Life: https://uclouvain.be/en/study/university-life.html
- LSM - Upon arrival: https://uclouvain.be/en/faculties/lsm/upon-arrival.html
- International Welcome Guide - University Life: https://uclouvain.be/en/study/university-life.html

UCLouvain's website

- LSM - FAQ: https://uclouvain.be/en/faculties/lsm/faq.html

Welcome

- International bachelor's, master's and doctoral degree students - Get informed: https://uclouvain.be/en/study/s-informer.html
- International bachelor's, master's and doctoral degree students - Useful links: https://uclouvain.be/en/study/liens-utiles.html
- International Welcome Guide: https://uclouvain.be/en/study/international-welcome-guide.html
- International Welcome Guide - When you arrive: https://uclouvain.be/en/study/when-you-arrive.html
- LSM - Upon arrival: https://uclouvain.be/en/faculties/lsm/upon-arrival.html
- International exchange students - FAQs: https://uclouvain.be/en/study/questions-frequemment-posees-0.html
- LSM - FAQ: https://uclouvain.be/en/faculties/lsm/faq.html
- Logistics & Accommodation Service: FAQ: https://uclouvain.be/en/study/accomodation/faq-frequently-asked-questions.html
- International Welcome Guide - University Life: https://uclouvain.be/en/study/university-life.html
- Le Service d'aide aux étudiants: https://uclouvain.be/fr/etudier/aide
- Apply to LSM: https://uclouvain.be/en/faculties/lsm/apply-to-lsm.html

Welcome events

- LSM - Upon arrival: https://uclouvain.be/en/faculties/lsm/upon-arrival.html
- International bachelor's, master's and doctoral degree students - Welcome to the international students !: https://uclouvain.be/en/study/welcome-activities-in-louvain-la-neuve.html
- International bachelor's, master's and doctoral degree students - Welcome day at faculties https://uclouvain.be/en/study/orientation-days-13-25-september-2018-for-new-incoming-international-students.html

Appendix C

Integration of the chatbot

In this Appendix, we will concretely describe how we can integrate a chatbot made with *Dialogflow* within a web server [74] using *node.js* and within *Facebook* [75].

C.1 Integration of a *Dialogflow* chatbot in a *web server*

Access *Google Cloud Platform Service Account*

This integration requires the creation of a private key for the service account of the chatbot. To access the page of the *Google Cloud Platform Service Account* where this key is generated, we need to click on the link next to the text "Service Account" of the *GOOGLE PROJECT* section of the settings page in *Dialogflow*.

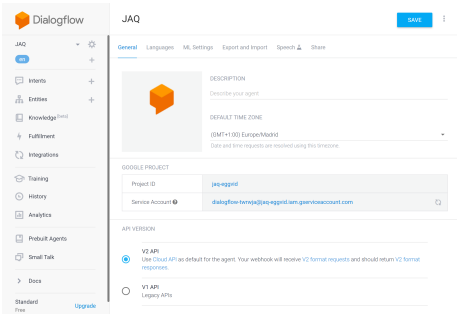


Figure C.1: *Dialogflow* settings page

Creation of a service account

There we can create a service account by clicking the button labeled "Create Service Account" (or "Créer un compte de service" in French) at the top of the page.

Fill in the details of the account

We now can fill in the basic information related to the account i.e. the name, the ID and a description. End this step by clicking on the button "Create".

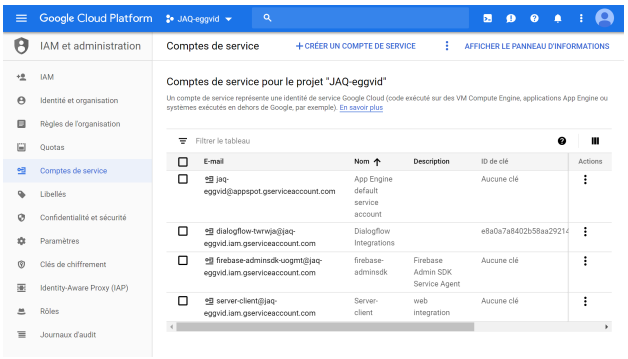


Figure C.2: *Google Cloud Platform Service Account* page

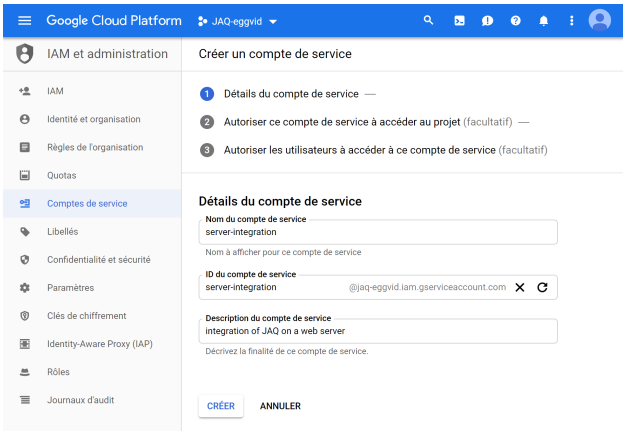


Figure C.3: Details of the new Service Account

Set the authorization of the account

Here, we need to grant sufficient authorizations to the web server. Nonetheless, we would like to give the minimum authorizations required for security reason. The most secure level is then "Dialogflow Agent Service" (or in French "Agent de service Dialogflow").

Generate the private key

There are two fields that we don't need to fill here. We can simply click the button "Create a key" (or "créer une clé" in French) at the bottom of the page. Then, we can choose a format for the key and the file containing the key can then be downloaded by clicking the "Create" button.

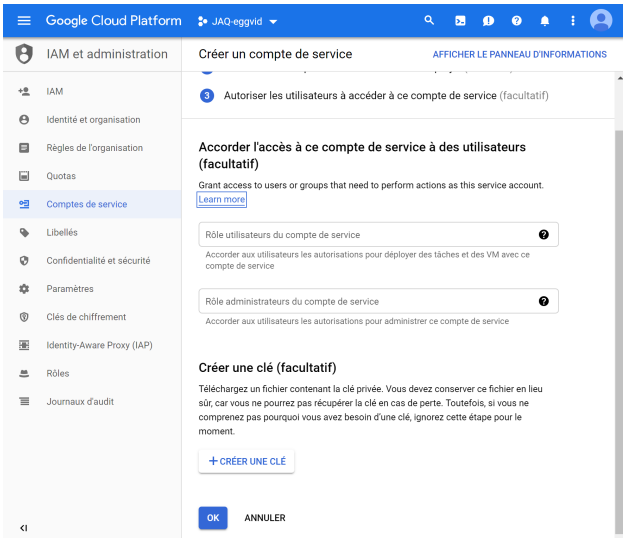


Figure C.4: Generate a private key for the new Service Account

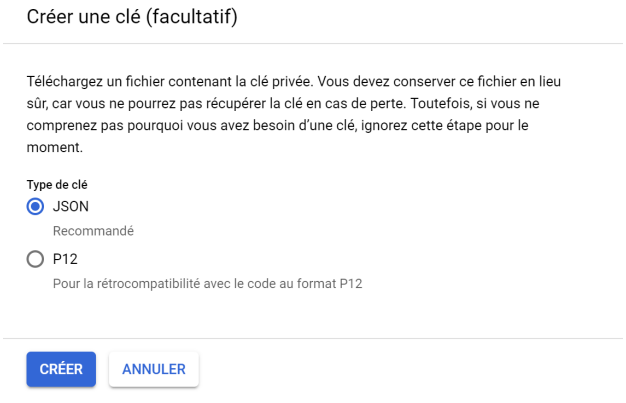


Figure C.5: Download the private key of the new Service Account

Set the private key

The private key can be, in the case of our implementation, be placed in the directory named "server". Then the name of its file should be added in the file "credentials_dialogflow_JAQ.js" in the field "credentialFile". The field "projectId" of the same file should also be filed by the id indicated on the setting page of the chatbot.

C.2 Integration of a *Dialogflow* chatbot in *Facebook Messenger*

In this section, we will detail the necessary steps to integrate a *Dialogflow* chatbot into a conversation on *Facebook Messenger*. This section is based on the documentation of *Dialogflow* [75].

Prerequisites

To proceed through the following steps, we first need a *Dialogflow* chatbot, a *Facebook* developer's account (which requires a basic *Facebook* account).

Create a *Facebook* application

From the home page for developers of *Facebook*[1], we can create a new application by clicking on the button labeled "Add new app" in the "My apps" drop down menu. We can then specify the name of the application in a pop-up window.

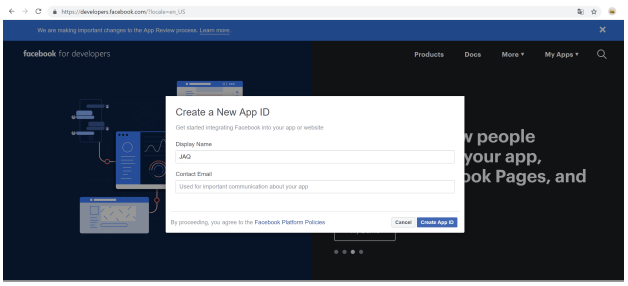


Figure C.6: Creation of a *Facebook* application

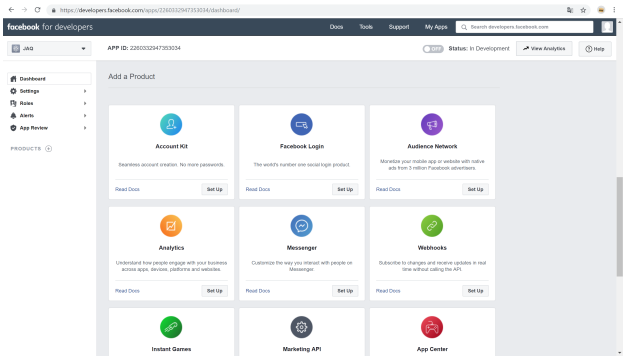


Figure C.7: Add the *Messenger* module to the *Facebook* application

Add *Messenger* to the application

Then, we can add the module *Messenger* to our application by going into *Dashboard*, scrolling down to the section "Add a Product" and clicking on the "Set up" button.

[1] `https://developers.facebook.com/`

Setting up the *Messenger* module

Once on the settings page of the *Messenger* module, we can see it is required to have a *Facebook* page to generate an access token. The creation of a new page will be described in the next subsection.

Once we have a *Facebook* page to integrate the chatbot on, we need to grant the chatbot access to this page. To do that, we simply need to accept that the page can send message through our *Facebook* account.

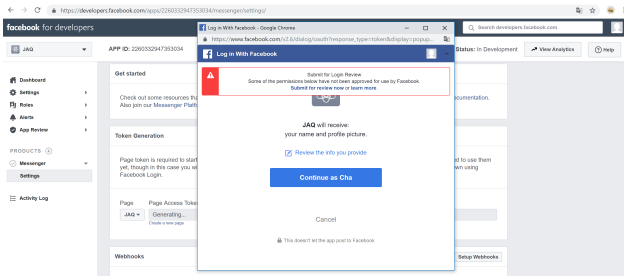


Figure C.8: Selection of the account

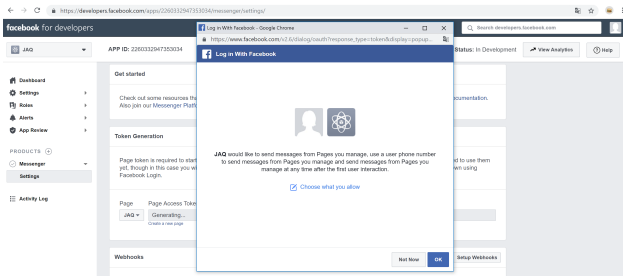


Figure C.9: Granting access to the *Facebook* page

Creation of a *Facebook* page

Here is a step-by-step explanation of how to obtain a page to integrate the chatbot on. In the settings page of the *Messenger* module, inside the section "Token Generation" at the very bottom, there is a link "Create a new page" that we can follow to create a new page. The page `https://www.facebook.com/pages/` allows to do the same thing.

We first had to choose a category for the page (in our case "Community or Public Figure"). Then, we are asked the name of the page.

Link the *Facebook* application with the chatbot

Now, on the *Dialogflow* page, we can activate the *Messenger* integration, in the "Integrations" section. We then get a pop-up window where we have to enter a "Verify Token" and the *Page Access Token* from *Facebook*. The "Verify Token" can be anything but is required to continue the configuration of the *Messenger* module. We will also need the "Callback URL".

Setting up the *Messenger* module (second part)

Finally, we can finish the configuration of the *Messenger* module by setting up the webhooks, by clicking on "Setup Webhooks". We then enter the "Callback URL" and the "Verify Token" from *Dialogflow*. We also have to check the *message* and *messaging_postbacks* options under the "Subscription Fields".

After closing the pop-up window by clicking "Verify and Save", we need to subscribe to our page by selecting it in the drop down menu.

Put the application online

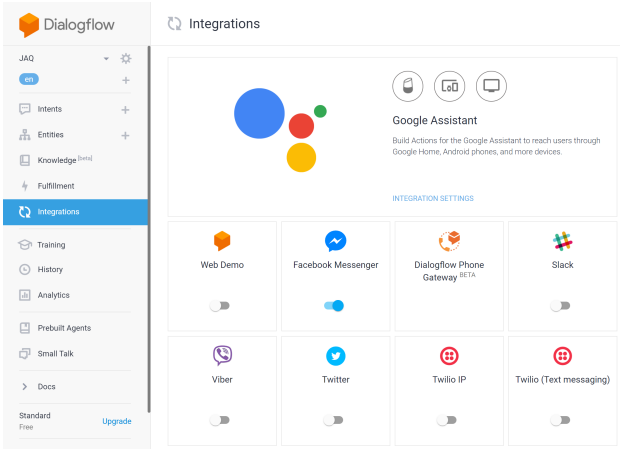


Figure C.10: Activation of *Messenger* integration in *Dialogflow*

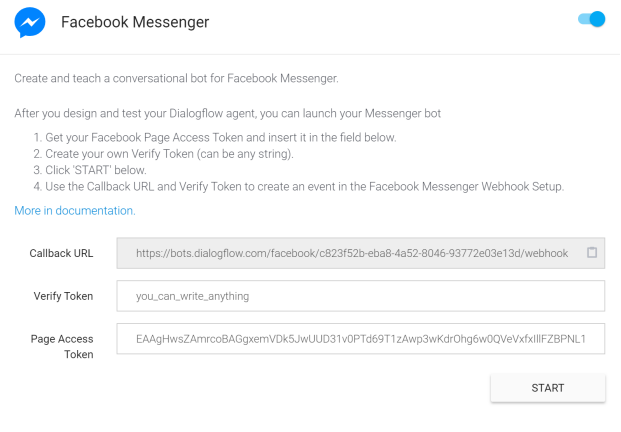


Figure C.11: Add tokens in the *Messenger* integration of *Dialogflow*

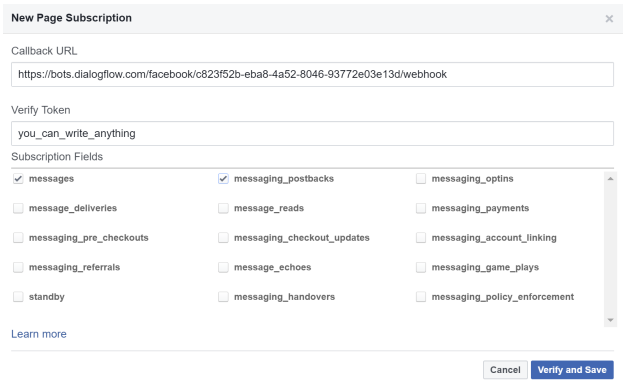


Figure C.12: Configuration of the webhooks of the *Messenger* module

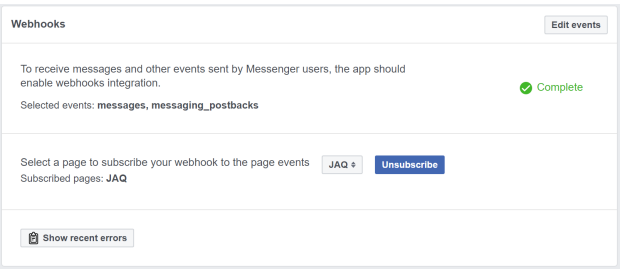


Figure C.13: Page subscription for the webhooks

We now need to activate the application. To do so, we still need to fill in some information in the "Basics" subsection of the "Settings" tab. There, we need to put a *Privacy Policy URL* and select the category "Messenger Bots for Business" for our application.

We can then make the application public by clicking the switch at the top of the page, and confirm by clicking the "Confirm" button.

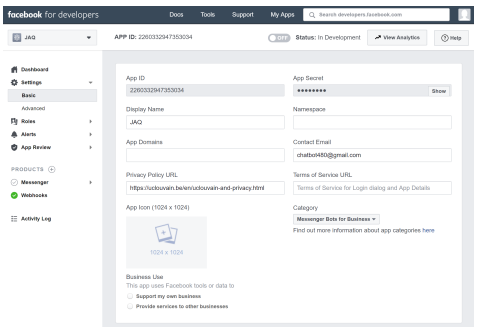


Figure C.14: Basic settings of the *Facebook* application

Test the *Facebook* integration

The simplest way to test the integration of our chatbot with *Messenger* is by getting the link to the *Facebook* page. To find it, we need to go in the "Settings" of our page then in the "Messenger Platform" and finally scroll down to the "Your Messenger Link" section. With this link, administrators of the page can talk to the chatbot through *Facebook Messenger*.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**