Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Computer Science and Engineering
Dipartimento di Elettronica, Informazione e Bioingegneria

DESIGN AND DEVELOPMENT OF REXY: A VIRTUAL TEACHING ASSISTANT FOR ON-SITE AND ONLINE COURSES

Relatore: Prof. Paolo Cremonesi

Tesi di laurea di:
Manuel Parenti Matr. 876085

Anno Accademico 2017–2018

Acknowledgments

I would like to express my deep gratitude to Professor Paolo Cremonesi, my research supervisor, for the opportunity he gave me to participate in this project and for his guidance and enthusiastic encouragement. I would also like to thank Dr. Luca Benedetto, for his continuous support and constructive recommendations during my work.

I wish to thank my family for their support and encouragement throughout my study.

Finally, I would like to extend my thanks to Monica, for being always present and encouraging me to improve every day, and to my study partners Alberto, Giorgio, Giovanni, Marco and Tommaso for the beautiful experiences shared during the years at Politecnico di Milano.

Abstract

Digital assistants are now present on the main mobile operating systems as native apps and can be found inside messaging applications and websites where they aid users with a wide variety of tasks. They pervade our lives, bringing useful information to our attention and answers to our requests. The educational domain can benefit from the adoption of such technology, because it allows to reach all the students of a course and help them by solving their doubts, navigating through the content of a course and proposing exercises.

The goal of this thesis is to show how a virtual teaching assistant can be created, in a modular and general enough manner, allowing its adaptation to different situations and courses. This work presents the details of the design of Rexy, an assistant that has been employed for the *Recommender Systems* course at Politecnico di Milano, with the aspects considered to choose the underlying technologies. The idea behind Rexy is to augment the answering capabilities of a teacher, by responding to frequently asked questions in an automatic way. This initial concept has been expanded in order to be able to answer to more general and unexpected questions, test the understanding of concepts and control the behavior of the assistant during its activity, to improve it with time.

Students can interact with Rexy through Slack, one of the most widespread messaging applications. Their messages get interpreted by the IBM Watson™ Assistant conversational computing service, which proposes the answers that are completed and sent back to students by a Node.js server. This intermediate application orchestrates all the flow of messages, extends the understanding capabilities of Assistant while offering additional functionalities to the students and teachers.

Sommario

Gli assistenti digitali sono presenti sui principali sistemi operativi mobile sotto forma di applicazioni native, possono essere trovati anche all'interno di applicazioni di messaggistica e siti web, dove sono sfruttati dagli utenti per svolgere vari tipi di compiti. Essi pervadono le nostre vite, portando informazioni utili alla nostra attenzione e rispondendo alle nostre richieste. Il settore dell'educazione può beneficiare dell'adozione di tale tecnologia, perché essa consente di raggiungere tutti gli studenti di un corso e aiutarli risolvendo i loro dubbi, guidandoli attraverso il contenuto di un corso e proponendogli esercizi.

Lo scopo della tesi è quello di mostrare come creare un assistente virtuale per la didattica, in modo modulare e abbastanza generale da permettere il suo adattamento a diverse situazioni e corsi. Questo lavoro presenta i dettagli di progettazione di Rexy, un assistente impiegato nel corso di *Recommender Systems* al Politecnico di Milano, insieme agli aspetti considerati nelle scelte delle tecnologie sottostanti. L'idea dietro al progetto è quella di estendere le possibilità che un insegnante ha nel rispondere agli studenti, occupandosi delle domande più frequenti in modo automatico. A questo scopo iniziale si sono aggiunte funzionalità per permettere a Rexy di rispondere a domande più generali e inaspettate, di testare la comprensione dei concetti di un corso e di controllare il suo comportamento durante il suo periodo di attività, per migliorarlo col tempo.

Gli studenti possono interagire con Rexy attraverso Slack, una delle applicazioni di messaggistica più diffuse. I loro messaggi vengono interpretati dal servizio di calcolo conversazionale IBM Watson™ Assistant, che propone delle risposte ad un server Node.js che le completa e le inoltra agli studenti. Questa applicazione intermedia orchestra il flusso di messaggi, estende le capacità di comprensione di Assistant ed offre l'adozione di funzionalità addizionali per studenti e insegnanti.

Contents

Introduction **1**

1 State of the Art **5**

1.1 Chatbots 5

1.1.1 From ELIZA to Present 5

1.1.2 Application of Current Chatbot Technologies 6

1.1.3 How Chatbots Are Perceived 7

1.1.4 Chatbot Architecture 8

1.1.4.1 General Architecture 8

1.1.4.2 Recognizing Intents and Entities 9

1.1.4.3 Conversational Computing Platforms 11

1.2 Virtual Teaching Assistants 12

1.2.1 Reviews of Virtual Assistants in Education 13

1.2.2 Jill Watson 13

1.2.3 Design of a VTA 16

1.3 Thesis Objectives 17

2 Software and Algorithms **19**

2.1 Watson 19

2.1.1 DeepQA Project 19

2.1.2 Watson™ Services 22

2.1.3 Assistant 23

2.1.3.1 Training Data 23

2.1.3.2 Dialog 25

2.2 Slack 28

2.3 Database Engines 30

2.3.1 MySQL 30

2.3.2 MongoDB 30

2.4 Clustering 31
2.4.1 Clustering Algorithms 31
2.4.1.1 K-Means 31
2.4.1.2 K-Medoids 32
2.4.1.3 Hierarchical Agglomerative Clustering 33
2.4.2 Clustering Evaluation 34
2.4.2.1 Adjusted Rand Index 34
2.4.2.2 Fowlkes-Mallows 35
2.4.2.3 Adjusted Mutual Information 35
2.4.2.4 Homogeneity, Completeness, and V-measure . . 35
2.4.2.5 Silhouette Coefficient 36
2.5 Spelling Correction 37

3 Statement of the Problem **39**
3.1 *Recommender Systems* at Polimi 39
3.2 Problems and Chosen Solutions 40

4 Architecture **43**
4.1 Rexy Architecture 43
4.1.1 Front End 44
4.1.2 Application Server 44
4.1.3 Database Server 45
4.1.4 NLP Component 45
4.1.5 Deployment 45
4.2 Databases 45
4.2.1 MOOC Database 45
4.2.2 MongoDB Databases 47
4.2.2.1 Administrative Database 47
4.2.2.2 Conversation History Database 48
4.2.2.3 User Database 49
4.2.2.4 Question Database 50
4.3 Watson Assistant Workspace 50
4.3.1 Intents 51
4.3.2 Entities 56
4.3.3 Dialog Tree 56
4.3.3.1 Dialog Nodes and Branches 56
4.3.3.2 Context Variables and Keywords 58

4.4 Node.js Application 61
4.4.1 Dependencies 61
4.4.2 Components 62
4.4.2.1 Classes 62
4.4.2.2 MOOC Modules 66
4.4.2.3 Administrative Modules 67
4.4.2.4 Question Modules 69
4.4.2.5 User Modules 70
4.4.2.6 Spelling Correction Modules 71
4.4.2.7 Watson Modules 71
4.4.2.8 Context Manager 73
4.4.2.9 Lookup Manager 74
4.4.2.10 Chatbot 75
4.5 Sequence Diagrams 77
4.5.1 High-Level Interactions 77
4.5.2 Initialization 79
4.5.3 Text Messages 79
4.5.4 Interactive Messages 89
4.5.5 Slack Dialogs 97

5 Results **99**
5.1 Interactions with Rexy 99
5.2 Maintenance and Portability 101
5.3 Scalability 102
5.3.1 Testing Scenario 102
5.3.2 Test Description 102
5.3.3 Results 103
5.4 Clustering of Users 104
5.4.1 Simulation 104
5.4.2 User Model 104
5.4.3 Techniques 105
5.4.4 Results 106
5.4.4.1 Scenario A - only intents 106
5.4.4.2 Scenario B - intents and entities 110

Conclusions **115**

Bibliography **117**

A Dialog Tree 121

B Use Cases 137

List of Figures

1.1 Basic chatbot architecture . . . 8
1.2 Chatbot with intent and entity classification . . . 11

2.1 DeepQA architecture . . . 20
2.2 Intent example . . . 24
2.3 Entity example . . . 25
2.4 Dialog tree example . . . 26
2.5 Dialog flow example . . . 28
2.6 Slack chatbot text message . . . 29
2.7 Slack chatbot interactive message . . . 29
2.8 Slack chatbot dialog . . . 29

4.1 High-Level architecture . . . 43
4.2 MOOC database . . . 46
4.3 Administrative database . . . 47
4.4 Conversation history database . . . 48
4.5 User database . . . 49
4.6 Question database . . . 50
4.7 MOOC classes . . . 63
4.8 Administrative classes . . . 63
4.9 Question classes . . . 64
4.10 User class . . . 64
4.11 Context class . . . 65
4.12 spell checker class . . . 65
4.13 MOOC modules . . . 66
4.14 Administrative modules . . . 68
4.15 Question module . . . 69
4.16 User modules . . . 70
4.17 Spelling correction modules . . . 71

4.18 Watson modules 71
4.19 Context Manager module 73
4.20 Lookup Manager module 75
4.21 Chatbot module 76
4.22 High-level view of interactions 77
4.23 High-level view with Assistant 78
4.24 High-level view with Assistant and databases 78
4.25 Text message sequence diagram number 1 79
4.26 Text message sequence diagram number 2 80
4.27 Interaction with lookup 81
4.28 Text message sequence diagram number 3 82
4.29 Interaction with contextual information 83
4.30 Text message sequence diagram number 4 83
4.31 Display of the list of concepts 84
4.32 Text message sequence diagram number 5 84
4.33 Display of a question 85
4.34 Text message sequence diagram number 6 85
4.35 Discovery query result 86
4.36 Text message sequence diagram number 7 87
4.37 Confirmation question 88
4.38 Teacher question 88
4.39 Interactive message sequence diagram number 1 89
4.40 Result after a user confirmation 90
4.41 Interactive message sequence diagram number 2 90
4.42 Result after a entity selection 91
4.43 Interactive message sequence diagram number 3 91
4.44 Managing a message that has been confirmed 92
4.45 Dialog with intent creation 92
4.46 Dialog with intent selection 93
4.47 Dialog with direct reply 93
4.48 Interactive message sequence diagram number 4 94
4.49 Interactive message with intent description 94
4.50 Interactive message sequence diagram number 5 95
4.51 Interactive message sequence diagram number 6 96
4.52 Interactive message with correct answer 97
4.53 Interactive message sequence diagram number 7 97
4.54 Dialog submission sequence diagram 98

4.55 Message received by a student from a teacher 98

5.1 Load test results 103
5.2 Adjusted Rand Index in Scenario A 107
5.3 Adjusted Mutual Info in Scenario A 107
5.4 Fowlkes-Mallows measure in Scenario A 108
5.5 Silhouette coefficient in Scenario A 108
5.6 Homogeneity in Scenario A 109
5.7 Completeness in Scenario A 109
5.8 V-measure in Scenario A 110
5.9 Adjusted Rand Index in Scenario B 111
5.10 Adjusted Mutual Info in Scenario B 111
5.11 Fowlkes-Mallows measure in Scenario B 112
5.12 Silhouette coefficient in Scenario B 112
5.13 Homogeneity in Scenario B 113
5.14 Completeness in Scenario B 113
5.15 V-measure in Scenario B 114

A.1 Dialog tree: part 1 122
A.2 Dialog tree: part 2 123
A.3 Dialog tree: part 3 124
A.4 Dialog tree: part 4 125
A.5 Dialog tree: part 5 126
A.6 Dialog tree: part 6 127
A.7 Dialog tree: part 7 128
A.8 Dialog tree: part 8 129
A.9 Dialog tree: part 9 130
A.10 Dialog tree: part 10 131
A.11 Dialog tree: part 11 132
A.12 Dialog tree: part 12 133
A.13 Dialog tree: part 13 134
A.14 Dialog tree: part 14 135
A.15 Dialog tree: part 15 136

List of Tables

4.1 Course intents . . . 51
4.2 Challenge intents . . . 52
4.3 Exam intents . . . 53
4.4 Lecture intents . . . 53
4.5 Course content intents . . . 54
4.6 Teacher intents . . . 54
4.7 Generic questions intents . . . 54
4.8 General intents . . . 55
4.9 General speech acts intents . . . 55
4.10 Entities . . . 56

B.1 Use Case 1: Lecture schedule . . . 137
B.2 Use Case 2: Next lecture . . . 138
B.3 Use Case 3: Exam in september . . . 138
B.4 Use Case 4: Exam possibilities . . . 139
B.5 Use Case 5: Starting day of the challenge . . . 139
B.6 Use Case 6: Second deadline of the challenge . . . 139
B.7 Use Case 7: One of the challenge rules . . . 140
B.8 Use Case 8: Definition of a specified concept . . . 140
B.9 Use Case 9: Synonym . . . 140
B.10 Use Case 10: References of a concept of the course inside the modules 141
B.11 Use Case 11: Description of a module . . . 141
B.12 Use Case 12: Description of a video . . . 142
B.13 Use Case 13: Definition without specified entities . . . 142
B.14 Use Case 14: Next event . . . 143

Introduction

Virtual assistants are employed in various industries to deliver information and perform a wide array of tasks, from communicating weather conditions to delivering customer care.

Analogous assistants can bring advantages in the educational domain: for example they enable the possibility to reach all the students of a particular course and try to help them to study better and revise concepts in a personalized way. These advantages are more noticeable in the case of large classes, where one or few teachers cannot have a personal interaction with all the students. The positive effects of the presence of a virtual teaching assistant (VTA) in classes are shown in [32] where different variations of virtual assistants have been compared against each other and against humans in teaching concepts from different domains.

One domain in particular can gain in terms of quality offered to students, that is e-learning with Massively Open Online Courses (MOOCs). Class Central reports that in 2017 the total number of people that enrolled to a MOOC reached 81 millions, distributed over 9.4 thousand online courses [46]. One of the main problems of MOOCs is the student's retention rate, which is typically lower than 50% [52]. During on-site classes students interact with each other and with teachers, while in e-learning students act as autodidacts. As stated in [31], this lack of interaction is one of the main reasons of low student retention . Engaging students with a VTA could lead to an higher retention rate as it can help them overcome their difficulties and lead them to the fulfillment of the learning objectives of the course.

The goal of this project is to design and create a VTA that can be used to improve the quality of teaching and of the learning experience both in the case of online and on-site courses. This improvement is linked with the possibility to scale and augment the teacher's ability to interact with students. The proposed VTA consists in a chatbot, an intelligent system that interacts with users via text messages on an official Slack workspace. This chatbot is capable of answering

students' questions about the content, the structure and the organization of the *Recommender Systems* course held at Politecnico di Milano.

Recommender Systems is an introductory course on recommender systems and in the current academic year, 2018/2019, the Professor Paolo Cremonesi is leading the class with a flipped classroom instructional strategy. In this scenario, teaching material is delivered in the form of online video lectures, while in-class lessons are reserved for making students engage with the content of the course and explore it more deeply.

The idea to make experiments with a virtual assistant originated from the intention of the Professor to hold a MOOC on the topics of the *Recommender Systems* course on Coursera, arguably the most famous and populated e-learning platform. The goal of this experiment is to be able to give a helpful service to students that normally would not be able to interact with the teacher as well as their on-site colleagues. Indeed, at the beginning of this project the test environment consists of the class of students of the on-site course.

In the scenario of university courses usually there are a single teacher and few teaching assistants that can answer a limited number of questions and requests asked by students. This holds true also in the case of *Recommender Systems*, where there is only one professor and one teaching assistant. These questions can be usually asked during lectures, where the answer reaches the whole classroom of students, or via e-mail where the answer is received only by the sender of the message, limiting the effect of the teacher's effort. In the case of online courses, the number of students can be enormously greater than the number of students enrolled to an on-site course, leading to a number of potential requests too large to be handled by a single or few teachers.

The students' requests can be classified into two main categories, those related to the syllabus of the class and those related to the concepts contained in the learning units of the class. The learning units can be lectures, videos or documents depending on the medium utilized to convey information useful for the students to learn and reach the learning objectives of the course. Among those requests there are frequently asked questions (FAQs) that are repeated many times and that can be dealt with in a more efficient way than trying to answer them one by one. There are also questions that need more human effort and knowledge to be answered correctly and in a satisfying way.

Having a VTA able to answer FAQs gives teachers the time to focus on more interesting and demanding requests. The work that is mostly related to this thesis is the research discussed in [35], in which Professor Goel from Georgia Tech

discussed the possibility of using VTAs in order to reduce professors' workloads. VTAs can make education much more scalable since students can solve most of their problems without asking the teachers for help. Previous research showed that it is possible to create and deploy VTAs to help students of online and offline courses, but it did not share the details on how such assistants can be created. The main difference from previous work consists in the focus on the VTA's architecture, showing how it is built, how it works and how it will be expanded in the future.

VTAs can also be leveraged to recommend learning material to students, increase student engagement and help students revising concepts from the courses. The chatbot I propose is quite general and it can be easily adapted to different courses, how it will be shown in a later section.

One of the VTA's key features is that it creates an environment where users can find information about the syllabus of the course in a natural way, that is using their words. It is able to give enrolled students the piece of information they need about lectures, exams, deadlines and specifications for a competition that lasts for the whole semester. Aside from information about the syllabus, users can interact with the chatbot while studying or revising concepts from the classes. The bot can support them by giving definitions of those concepts and finding the right sections of the course material where they can be studied. The sections consist of a list of learning units, that can have the form of videos or text documents. Another feature of the bot is that it can be used to test the user's understanding of the concepts that he should have studied through the course, therefore help them revise the concepts in which they are not very confident. This can be useful to gain in terms of user retention and in having more confident students taking the exams.

To give the chatbot the natural language processing capabilities needed in order to understand the student's questions, I leveraged IBM's Watson™ Assistant service. It stands at the core of the architecture of the chatbot, but it needs to delegate some tasks to a dedicated application to find the most fitting way to select answers, retrieve information and to bring some external features to the users. One of the first problems that comes up when creating a chatbot is that there might not be records of prior interactions between teachers and students. This is even more complicated when the domain of application is very specialized and there needs to be an expert of the domain to design the best dialog flows for the end users. This problem can be overcome by leveraging one of the commercial off-the-shelf software solutions on the market. In fact, the choice of Watson™ Assistant as natural language processing (NLP) service came through because it allows users to

create and deploy bots easily, since it takes charge of the task of understanding and recognizing the meaning of the messages sent to the chatbot. Watson™ Assistant allows to create a rule-based chatbot, by using this service one can identify the intent of each of the users' messages and the entities contained in them and define the rules for selecting the most appropriate answer in case any intent gets recognized. Therefore, an initial phase of design of the possible intents, entities and rules is needed and depending on the domain of the application the structure of the created dialog flows can be significantly different.

Whilst the purpose of the VTA described in this thesis is to help the students of Politecnico di Milano reaching the learning objectives of *Recommender Systems*, it can be used as a guideline to build assistants for fully online courses and for other areas of study.

The structure of the thesis project is described below:

- In Chapter 1 there is an analysis of the state of the art for chatbots, focusing on VTAs.
- In Chapter 2 there is a description of the most important software technologies and algorithms that made the development of the VTA possible.
- In Chapter 3 there is a description of the context in which the experiment is run and of the different problems taken into account to design and develop the VTA.
- In Chapter 4 the architecture of the VTA, the databases used to store its knowledge, the training data and dialog tree design, the architecture of the server application and the ways in which its modules interact are described.
- In Chapter 5 the results of the work and a preliminary study of how to model groups of users sharing similar interest, based on the interactions they have with the VTA, are presented.
- In the Conclusions there are the possible future developments for the VTA.

Chapter 1

State of the Art

1.1 Chatbots

Chatbots are intelligent systems that interact with users via messaging, text, speech or customized graphical interfaces. They enable users to retrieve information and perform specific actions with simple conversations, they are available 24/7, they can answer to multiple people at a time and they can outperform humans in terms of speed and accuracy in a narrow domain. Users can take advantage of the functionalities that bots offer on the channel they prefer, creating personalized conversations with a bot that can manage a great number of concurrent users. Something that is very difficult and expensive to reproduce with a group of humans, since the number of requests they can manage at a time is much more limited. Although, the quality of the answers is a completely different matter and it depends on the quality of the chatbot and more importantly on the domain of application and the expertise needed to write a satisfactory response.

1.1.1 From ELIZA to Present

Conversational agents are not a new branch in the field of computer science, though. In the 1960s, Weizenbaum published an innovative study on human interaction with a computer program named ELIZA. It was developed to demonstrate the superficiality of communication between humans and machines, its conversations were supported by pattern matching, the users' input messages were matched with predefined texts, and a substitution methodology, needed to insert part of the input message into the response text. This gave the users the illusion that the bot was understanding the meaning of their words. The way in which ELIZA responded depended on predefined scripts, that defined the rules of the

response to use in specific situations. The most famous script simulated the responses of a psychotherapist in a therapy session [50].

Dale discusses "the return of chatbots" in recent years and explains how the current interest in this technology is rooted in previous work on natural language user interfaces [30]. Dale highlights the impact of the Loebner Prize, which has taken the form of an annual contest designed to implement the Turing Test. It has inspired the community to design natural language interfaces in order to be more human-like. One instance of this active community is Pandorabots, a chatbot platform that includes more than 250,000 bot developers and that has spawned more than 300,000 bots as of 2018.

Chatbots and virtual assistants are now a trending technology, in fact they can be easily found in different websites, chat applications and in the form of native bots in many operating systems. The reason is likely related to substantial advances in computing technology and the wide adoption of mobile messaging applications.

First, recent advances in artificial intelligence and machine learning promise vast improvements in natural language interpretation and prediction capabilities, including improvements in machine translation. Some modern chatbots are able to talk about many different subjects, they have a wide variety of possible answers and can give the illusion of emotion by impersonating a character. All of these features make them more human-like than their predecessors (such as ELIZA) [47].

From a technical point of view, progress in conversational modeling suggests that models based on recurrent neural networks and sequence-to-sequence models will out-perform the rule-based conversational models typically applied to traditional chatbots in the task of prediction of the next sentence in a conversation [49].

Second, the wide spread of messaging apps is testified by the fact that the four top messaging apps surpassed the top four social networks in terms of global monthly active users in 2015, and they continue to grow [28]. Indeed, messaging apps exceeded 6 billion combined monthly active users in 2017 [51]. This growth makes it even easier for users to interact with chatbots in their daily lives.

1.1.2 Application of Current Chatbot Technologies

The most popular and pervasive chatbots or digital assistants are the ones embedded in the most widely used digital devices, such as Microsoft's Cortana, Apple's Siri, Google's Assistant and Amazon's Alexa. All of these applications can help users with some of the standard virtual assistant's tasks, which generally in-

clude scheduling meetings, checking and making appointments on a personal calendar, reading, writing and sending messages, playing music, and, with different smart solutions, controlling devices of a automation-enabled home. All of them are cloud based, meaning that the interpretation of the message and the preparation of the response is created on machines running on the cloud. Some of them also let users install widgets made by third-party vendors, to increase the number of functionalities offered by the standard assistant.

Aside from these widely known applications, chatbots can be help users of a website in performing particular tasks by providing a simple and intuitive interface, which doesn't require them to learn how to navigate a traditional graphical interface. Also, it is very easy to find chatbots that operate in messaging apps such as Facebook Messenger, Telegram, Slack, WeChat and others. Depending on the platform where the bot is deployed, different services can be offered to the users.

Different industries can benefit from using chatbots to engage with customers, leveraging the inexpensive and wide-reaching technology. For instance, people in the US can search and book trips using Hipmunk's chatbot, they can also request a taxi ride from Lyft via chat or voice. Students of Politecnico di Milano can receive assistance on administrative matters from a FAQ chatbot. These are just a handful of examples of how chatbots are employed in different businesses, on the internet there is a huge number of different chatbot solutions that can be applied in the most diverse fields.

1.1.3 How Chatbots Are Perceived

Chatbots are considered as human computer interfaces, based on natural language. Thus, it is important to study the way in which chatbots are perceived by their users and what they expect when interacting with them, in order to meet their requirements and improve the design of functionalities and conversation flows.

Literature is lacking empirical studies on the reasons why people write or talk to the modern wave of chatbots, but some early results are available. In [29] Brandtzaeg et al. present an experiment aimed at finding the reason and the way in which people use chatbots to fulfill their needs. The target user base for the experiment was people from the US aged between 16 and 55. The method used to gather data was an online questionnaire, published in April 2017 through an independent research company. After evaluating the responses, 146 of them were considered valid, meaning that they came from people that said they were chatbot users. The results show that 68% of participants reported productivity to be the

main reason for using chatbots. 42% of participants highlighted the ease, speed, and convenience of using chatbots as the main reason for using them. Also, they noted that chatbots provide assistance and access to information. Yet, a substantial proportion of participants (20%) reported using chatbots for entertainment. This shows that productivity-oriented bots may benefit from a empathetic appearance. The results of this study highlight the most important needs that users have, and that chatbot designers must take into account. In particular, productivity and social interaction may drive people to use chatbots in a stable manner over time. However, this study is subject to limitations. The authors note that the chatbot users that participated in this study were both self-selected and filtered by some initial questions. They are, therefore, not representative of the population at large.

1.1.4 Chatbot Architecture

In this section, a short overview of the software components that make up a chatbot is presented. A more complete description and exploration of those concepts resides in [39].

1.1.4.1 General Architecture

Chatbots, in their simplest form, can be represented as the union of a messaging interface and a component that processes users input and responds using natural language.

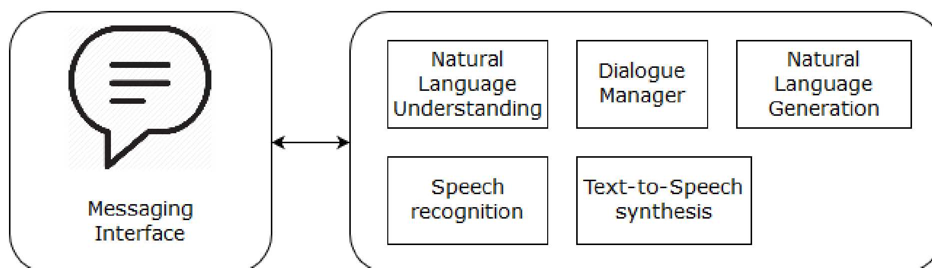

Figure 1.1: Basic chatbot architecture

The second component can have various capabilities, such as the ability to recognize the text of users' voice messages and reply with synthesized voice messages. Other additional features can be: the ability to converse in different languages, data analytics features, support for maintenance. Natural Language Understanding (NLU), Dialog Manager and Natural Language Generation (NLG) are

key components in the architecture of a chatbot, but they can have many different flavors.

The NLU component's job is to understand the meaning of input messages, recognizing the key details the users want to express. There are many ways to represent the meaning of sentences and the most common is based on frames and slot semantics.

The Dialog Manager controls the architecture and structure of dialogue. It takes input from the NLU component and maintains a state of the conversation. It then instructs the NLG module on how to create the output message.

The NLG module has the task of assembling the output messages to be sent to the users. The ideal solution would be to generate a sentence completely from scratch, creating a unique response for different input sentences in different contexts. This approach is the most difficult to model and develop, usually it involves using recurrent neural networks, but it has a significant advantage over simpler models that use predefined answers, which might feel more static to the end users. Another approach is to use retrieval-based models: for every input message, given the current context of the conversation, they search the best possible answer inside a database of answers by following a heuristic. For instance, in the case of an example-based conversational agent, the answers are defined for a group of input messages and for any of these input strings the reply is decided in a deterministic way. This simple model can be seen as a combination of a database of known input texts and a set of rules, that link every known input to a particular response. If a previously unseen message is received from the chatbot, it might look into his database of pre-recorded input messages and search for the most similar one. As a reply, the predefined answer for the most similar known input is used. Obviously, a measure of the similarity needs to be defined. Pattern-based heuristics can also be used to create very simple chatbots. They rely on patterns: when the chatbot receives a message, it goes through all the patterns until it finds a pattern which matches the user message. If a match is found, the bot will respond using the corresponding pattern. Pattern-based methods are very limited and in order to be able to respond to various questions, a great number of patterns needs to be created by hand and this is not an easy and quick task.

1.1.4.2 Recognizing Intents and Entities

A much more flexible approach is to classify the intent of a user's message. An intent is the user's intention. Or, in the case of questions, it represents the type of

question that can be asked by the user. The main idea behind this approach is that users can express a particular request in many different ways, e.g. to ask for the ingredients of a cake they can say "*Can you give me the recipe of a cake?*". Things get more complicated if the user asks for a list of delicious cakes in a message and then chooses one of them, after some messages he might ask: "*ok, how do I make it?*". In this case the intent of the user is always to get information on how to make a cake. In other contexts, the meaning, and so the intent, of the message might be completely different. Chatbots capable of understanding the intent, leveraging information about the context, are able to manage this kind of conversations easily. Usually, together with intents, chatbots recognize entities inside input sentences. This allows them to define more cases (and so answers) to manage and achieve a better understanding of the requests of the users. A named entity is a real-world object that can be referred to with a proper name. In the example of cakes *"cheesecake"* or *"red velvet"* could be two different values of an entity called *"cake"*. If a chatbot recognizes both the intent of request of ingredients and the *"cake"* entity, it can search for the ingredients of the requested cake inside a knowledge base of recipes.

With machine learning techniques it is possible to train an intent or entity classifier. These techniques need a training set composed of user messages and the corresponding intents and entities as examples, in order to find patterns in the data that will allow them to classify previously unseen input data. When the chatbot gets the intent of the message, it shall generate a response. The simplest way is just to respond with a static response, one for each intent or combination of intent and entities. Or, in order to create more dynamic answers, variables can be inserted in the response. When generating the answer those variables can be filled with the values of recognized entities or coming from the context of the conversation, such as the time or the location of the user. A question-answering bot can use a knowledge graph, use some algorithm to score the potential answers and select the most fitting one, as shown in figure 1.2.

At the start of the processing of an input message the bot tries to understand what the user is talking about. To do so it gives a score to the intents it already knows, using the intent classification module, and uses the intent with the highest score in the following task of identifying a compatible answer. The named entity recognition module searches for predefined entities inside the input message, it can recognize them using synonyms, patterns, regular expressions or even machine learning models. The answer generator searches through a database of answers, looking into a knowledge graph using the selected intent, the recognized entities

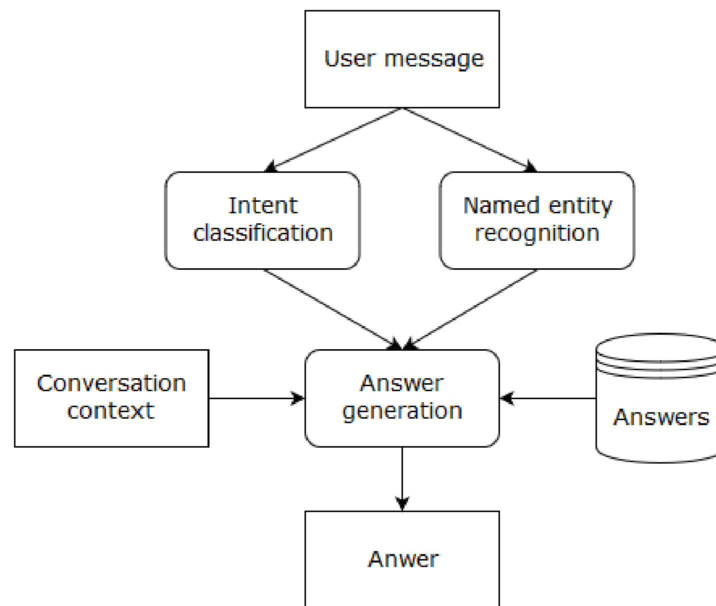

Figure 1.2: Chatbot architecture with intent classification and entity recognition

and some information coming from the context of the conversation to select the most appropriate answer.

1.1.4.3 Conversational Computing Platforms

The conversational computing platforms' market has several players. The services they offer can be used to design, create and deploy chatbots focusing on the design of the conversation, while outsourcing the task of analyzing the messages coming from the users. Usually, bots created with those platforms can be deployed to the most common messaging applications or to a proprietary web application. There are services coming from IBM, Amazon, Google, Microsoft and other online services such as Chatfuel [4], and free services, some of them are wit.ai [27] and Pandorabots [19].

IBM Watson™ Assistant (formerly Watson Conversation) [7] has a technically robust conversational platform with developer-friendly tools, being the solution used here, it will be presented in the following chapter.

Amazon's Alexa for Business™ [1] leverages AWS and Alexa's strengths. With more than 30,000 skills on Alexa, customized voice and chat experiences can be built on mobile devices and chat applications.

Google's DialogFlow™ [5] incorporates Google's machine learning expertise and products. It is optimized for the Google Assistant, but it can be leveraged to

deploy bots to wearables, phones, cars, speakers and other smart devices.

Microsoft has three different products: Microsoft Bot Framework™ [12], LUIS™ for natural language processing and intent management [13], Azure Cognitive Services for extended AI support [11]. Microsoft offers a single development environment to leverage all their products.

1.2 Virtual Teaching Assistants

Since chatbots generated a lot of interest, examples of the application of this technology in the education field are not lacking.

Chatbots have been used as a natural language interface to gather information for more complex applications. In [44] an approach to automated learning path generation inside a domain ontology supporting a web tutoring system is presented. In the proposed system, a chatbot has the task of asking users for their preferences on what they want to study as a first step. After having received enough information through a conversation, the system creates a learning path for the users automatically, telling them what chapters in a MOOC they should be consulting and in which order.

However, chatbots have been also employed as VTAs in online and on-site teaching scenarios. The motivation for using chatbots in this field is that they can reach every involved student and address his personal needs to help him achieve the learning objectives of a course.

A chatbot has been at the center of a study regarding a web-based teaching system for foreign languages [38]. The idea behind that study is that a chatbots' conversational abilities could be helpful for a student in learning a new language. The bot was instructed to respond to the students' sentences in order to let them practice with the words and sentences they previously learnt. Furthermore, the sentences that the chatbot could recognize were organized in more than 20.000 categories inside a knowledge base. Each category contained an input-pattern and an output-template. If a user typed something as an input, the program would look into the memory for a matching category. If a matching category was found, its output-template was retrieved and transformed to the output of the chatbot. If no matching category was found the chatbot selected a random sentence that could go well in any situation, to let the conversation continue. The results of the experiment showed that users would identify the teaching assistant as a bot quite soon (within 10 rounds) during conversations, since the capabilities of the chatbot

in understanding the input messages were poor and its responses were static. In fact, most of the users noticed that the answers from the chatbot were irrelevant with the topic and the context of the conversation.

Nevertheless, there are also experiments that got promising results, that show how a VTA could be helpful for students and teachers.

1.2.1 Reviews of Virtual Assistants in Education

B. du Boulay analyzed different researches regarding the effectiveness of intelligent systems as classroom assistants [32]. His work's main focus is on the comparative effectiveness of AIED (Artificial Intelligence in Education) systems versus human tutoring and the systems used for the researches are not meant to substitute teachers, but as a support for blended learning.

One of the studies that du Boulay references to is [48], In which, comparisons of different styles of tutoring are presented: no tutoring, answer-based tutoring, step-based tutoring, substep-based tutoring and human tutoring. The differences between answer-based and step-based tutoring are related to the granularity of the interaction between the assistant and the student. The step-based one consists of following a reasoning of different steps instead of giving hints about the final solution directly, like answer-based tutoring does. Substep-based tutoring has a granularity of detail that is even finer than the one a normal student would go through when solving a problem. The experiments ran on STEM (science, technology, engineering, mathematics) subjects, where a step-based tutoring can make sense. VanLehn reported that step-based tutors were "just as effective as adult, one\hyp{}to\hyp{}one tutoring for increasing learning gains in STEM topics".

Other meta-reviews are contained inside du Boulay's study and the overall result is that intelligent systems adopted in a blended learning scenario, with other teaching activities, is beneficial. In particular, this effect is more pronounced in large classes where teachers and teaching assistants have to deal with a great number of students. The studies showed also that higher levels of education benefitted from this technology more than lower levels, and particularly when applied to STEM subjects.

1.2.2 Jill Watson

This section is about the famous example of virtual teaching assistants employed by Professor Goel in his course at Georgia Tech, presented in [35], which

inspired this work.

In 2014, the Knowledge-Based Artificial Intelligence (KBAI) course at Georgia Tech had a few dozen students enrolled. The course has a dedicated forum on Piazza, Georgia Tech's Q&A platform, where students can post questions, to be answered by Professor Goel or his teaching assistants. The number of questions posted on Piazza was manageable, until the launch of the online version of the course which had nearly 400 students. In fall 2016, the number of questions posted by students of the online course exceeded the 12,000 units, while the residential students posted six times less messages. Thus, the activity of monitoring and responding to those requests took a large amount of staff time, more than the professor and his assistants could handle to provide high quality answers in a timely manner. Goel was worried that online students were losing interest over the course of the term. The strategy he followed to improve interactivity in the MOOC related to the KBAI course was to use VTAs to augment interaction with human teachers. He noticed that some of the questions that were posted were similar to each other, and that there could be a solution to deal with them efficiently, leaving more time for more creative questions. He had the idea that introducing a VTA in his staff might be the right solution to handle the FAQs. The first generation of the family of VTAs that Goel and his team created through the years consisted of a single assistant called Jill Watson. It was introduced for the spring 2016 online class. Jill assisted students in both Goel's physical class and the more heavily attended online version. The questions she answered were routine but necessary, such as queries about proper file formats, data usage, and the schedule of office hours; the types of questions that have firm, objective solutions. Jill's existence was revealed at the end of her first semester on the job, but most of the students were surprised because they could not understand it on their own.

Jill was built with Bluemix APIs, an IBM platform for developing applications using Watson™ and other IBM services. Goel uploaded four semesters' worth of data, consisting in more than 40,000 questions and answers, to begin training his VTA, but Jill wasn't perfect at the start. Her early test versions gave not only incorrect answers but also strange answers. Goel's team created a mirror version of the live Piazza forum for Jill so that it could observe her responses and flag her errors, to help her learn. Eventually, Jill continuously learnt from experience and became better and better. Then came a breakthrough, part of the intellectual property of Professor Goel. It included not only Jill's memory of previous questions and answers in the process of understanding the meaning of the user input, but also the context of her interactions with students. In order to maintain a high quality

in the responses of Jill, the team decided to put a very high threshold (97%) for the confidence needed for the bot to answer. Confidence is one of the output variables that IBM Watson™ APIs send back when processing an input message, it tells the probability that the bot's interpretation of the input text is correct. This probability is likely to be related to the similarity of the input message with questions used for the training of the bot.

The results of Jill's work showed that while she answered only a small percentage of questions, the answers she gave were almost always correct or almost correct. Goel's staff wanted to increase the range of questions and tasks covered by her. This led to the development of a new generation of bots.

Starting from the second generation, they used a proprietary software and open-source libraries instead of IBM Watson™ APIs, to address their needs of increasing the functionalities and capabilities of the bot. In particular, the newly created VTA, called Stacy Sisko, was designed to reply to student introductions and to respond to FAQs as well. Stacy took care of more than 40% of the introductions and was able to manage more routine questions than Jill. However, this might be due to the fact that the dataset of answer-question pairs was larger, since those were collected also during Jill's existence.

A third generation of bots was developed and was deployed in the spring of 2017. The third version of Jill Watson relied on an episodic memory. It mapped the students' questions into relevant concepts and then used those concepts to retrieve the best fitting answer from its episodic memory of questions. Two VTAs were added to Piazza: a new version of Stacy, that took care of the introductions, and the third version of Jill, whose job was to identify and respond to FAQs. The new version of Stacy was able to take care of more than 60% of the student introductions, while Jill's third reincarnation autonomously answered 34% of student's questions, and of all the answers she gave, 91% were correct. The improvement through the different generations was significant, starting from a small percentage of handled questions to a third of the total number of student requests.

The advantages that can be gained through the deployment of such assistants is still under study, but it is still not possible to say that the teaching activity improved or that the quality of the answers is comparable to the one that human assistants can offer. Even though Goel states that it is too early to determine if Jill Watson was able to lower the demands on the teaching staff for the task of question answering, there is some evidence that Jill reduced the burden related to student introductions and posting messages to the class. He also states that it is too early to have insights on the student retention or student performance of a course with access to Jill

Watson.

The experiments with Jill Watson are fascinating and many of the courses, especially online, could benefit from the introduction of such a VTA to reach every one of their students to help them. Though, it is not clear how to create such assistants for the education field. Furthermore, a course might not have a record of interactions between students and teachers, making it more difficult to train a virtual assistant.

1.2.3 Design of a VTA

The creation and deployment of a chatbot can be facilitated by the available conversational computing platforms, but a careful design phase needs to be executed to implement a chatbot that can act as a VTA. Despite an increasing interest for chatbots in education, clear information on how to design them as VTAs is scarce. The aforementioned platforms only take care of the implementation phase of bots, and not of all the design of the conversation flow and the knowledge needed to understand the user messages. These two tasks are left to the chatbot designers, that will need to shape the way in which the bot answers in every possible situation, and they will also need to decide what the bot should be able to understand and teach him how to do it with example sentences. Since each domain of application of a chatbot has different needs it would be very difficult to create a bot that could provide assistance in any situation. In fact, travel bots might be good in helping users find flights, but they might not be able to answer to a question that would be asked to a teacher. Bots applied to different domains also have different vocabularies, meaning that they can understand different user intentions but also different entities in the conversation. A travel bot should be able to recognize dates and cities, while a VTA should be able to recognize the concepts from the course he is employed in.

A formal methodology for designing and implementing a chatbot as an intelligent tutor for a university level course using commercially available conversation frameworks is presented in [45]. The proposed methodology is built upon first-order logic predicates and focuses on two phases: knowledge abstraction and modeling, and conversation flow. The knowledge abstraction phase consists of representing the intents and entities to be recognized by the virtual assistant as first-order logic predicates. Intents are used as names of the predicates and entities as the arguments of those predicates. The conversation flow phase consists of designing a tree in which the answer for the user queries is searched for, af-

ter they have been analyzed to find the corresponding intents and entities. The choice of a tree as structure for finding the right answers is compliant with the way IBM Watson™ and other conversational computing platforms deal with this task. This paper analyzes some of the problems that occur when designing a chatbot for educational purposes, giving some example on how to apply the proposed methodology in the field of mathematics, using IBM Watson™ 's services. However, this is just a first step towards formalizing conversational agents in education, as the authors explain.

Despite the approach mentioned in the previously mentioned paper can be of support for the design of the intents and entities, how to effectively create a VTA is not fully explained. Moreover, the creation of a dialog tree with a branch for each tuple of intent and entities could be labour intensive and difficult to maintain. For example, if one of the intents of the VTA corresponds to questions asking for a definition of a specific concept of a subject, for each concept of the course one should create a branch inside the tree with the definition of that specific concept. It would be easier to have a database containing this kind of knowledge, significantly lowering the complexity of the dialog tree. When a user asks for the definition of a concept, a single leaf in the tree can launch a query to the corresponding database and create the answer for the user. Furthermore, a method for considering the context of the conversation while deciding which branches to take is missing. Also, a comment must be made about the architectural choices for the development of a VTA. Relying only on a service such as IBM Watson™ Assistant could limit the capabilities and the functionalities of such VTA. For instance, it would not be possible to execute the previously mentioned query to the database of definitions. Also, the use of interactive elements in the user interface of the VTA is very limited.

1.3 Thesis Objectives

The goal of this research is to shed light on how to design and create a VTA for an online or on-site course, to augment the teacher's ability to interact with students and improve their learning experience. The main objective of this VTA is to support students by answering FAQs regarding the syllabus of the course, as well as questions about the content of the course. Indeed, in this thesis a complete architecture for the development and deployment of a VTA is presented. It integrates the conversational computing capabilities of a commercial platform, namely IBM Watson™ Assistant, with a server application and Slack, a messaging appli-

cation used as front-end for the VTA. The proposed solution augments the capabilities of IBM Watson™ Assistant in understanding the user messages and, in addition, provides functionalities that are not available in such service. Some of them are: a management of the context of the conversation that is more flexible than the one offered by Assistant, the insertion of interactive elements inside the conversation with students, control over the conversations between students and the VTA to maintain a high level of quality in the responses and to make the VTA improve on messages it doesn't understand, analytics on the user interactions, lookups in specific databases that store knowledge about the course in which the VTA is employed. Moreover, the choices for each component of the architecture are explained, starting from the intents, entities and dialog tree used on IBM Watson™ Assistant, to the design of the databases. To understand how the different components communicate and bring together useful information for the students, also the interaction between the nodes of the architecture is illustrated in the next chapters.

Chapter 2

Software and Algorithms

This chapter is written for the readers' convenience, to introduce them to the software technologies and algorithms involved in the solution presented. If the reader is already knowledgeable in the concepts presented he may skip part of or all the following sections.

In Section 2.1 IBM Watson™ is presented, starting from the experiment that brought to the different services that are available on the IBM Cloud platform. The features of Watson™ Assistant are described in Subsection 2.1.3.

Section 2.2 presents the characteristics of the messaging application Slack.

Section 2.3 introduces the database engines, which are MySQL and MongoDB, involved in the architecture of the VTA.

Section 2.4 offers an introduction to the clustering algorithms applied to find student profiles from their interactions with the VTA.

In Section 2.5 the technique used for correcting the spelling of course specific words is illustrated.

2.1 Watson

IBM Watson™ [6] is a question answering (QA) intelligent system developed in IBM's DeepQA project. Watson can answer questions posed in natural language from an open domain, leveraging advanced NLP, information retrieval, knowledge representation, automated reasoning, and machine learning technologies.

2.1.1 DeepQA Project

The DeepQA project started as a challenge to build a computer system which could compete with human champions on *Jeopardy!* a famous American quiz

show [33]. *Jeopardy!* pits three contestants against one another in a competition that requires answering rich natural language questions over a broad domain of topics. In 2011, the Watson computer system competed on *Jeopardy!* against two of the show's most successful former champions, winning the first-place prize of $1 million [34].

DeepQA's architecture is not specific to the *Jeopardy!* challenge, though. It has been adapted to different business applications including medicine, enterprise search and gaming.

What IBM's researchers built is a massively parallel probabilistic evidence-based architecture. The philosophy behind DeepQA's approach is to integrate many different algorithms, each looking at the data from different perspectives. In fact, for the *Jeopardy!* challenge, they used more than 100 different techniques for analyzing natural language, identifying sources, finding and generating hypotheses, finding and scoring evidence, and merging and ranking hypotheses.

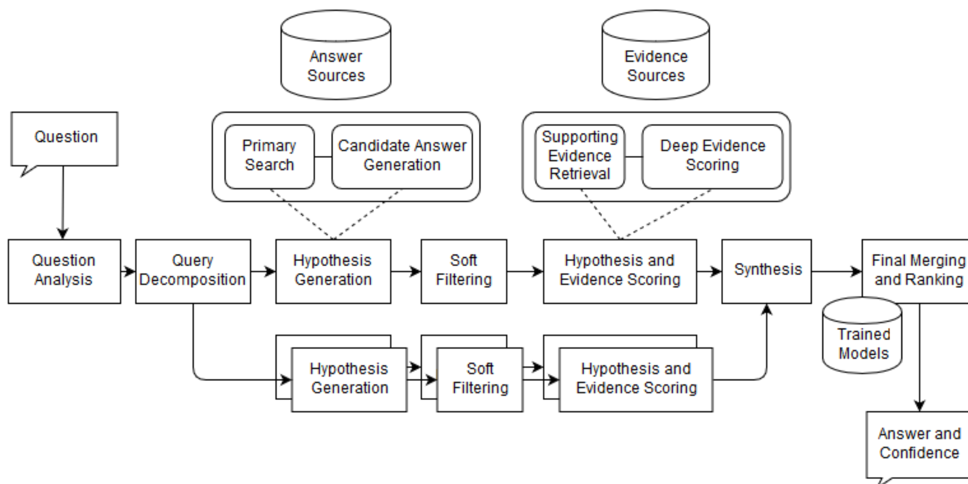

Figure 2.1: High-level architecture of IBM's DeepQA used in Watson™.

As can be seen in Figure 2.1 the task of answering a question with such an architecture is decomposed in many different subtasks.

Prior to any attempt to automatically answer a question there is a content acquisition phase, which is executed to gather the content to use as answer and evidence sources. The first step of the acquisition phase is to analyze example questions from the problem domain to produce a description of the categories of queries that must be answered, this is primarily a manual task. Given the kinds of questions and broad domain of the *Jeopardy!* challenge, the sources for Watson include a wide range of encyclopedias, dictionaries, thesauri, newswire articles, and

literary works. Starting from this corpus, DeepQA applies an automatic expansion process identifying useful documents, as well as structured and semi-structured content, from the web.

Then, the main steps applied to solve a QA task are:

- Question analysis: the system attempts to understand what the question is asking and performs the initial analyses that determine how the question will be processed by the rest of the system.
- Hypothesis generation: this step takes the results of question analysis and produces candidate answers by searching the system's sources and extracting text snippets from the search results. Each candidate answer is considered a hypothesis, which is examined by the system to decide whether it is relevant or not for the current query.
- Soft Filtering: scoring algorithms are used to prune the set of initial candidates down, to pass only a small set of candidates to the more intensive scoring components.
- Hypothesis and Evidence Scoring: candidate answers that pass the soft filtering threshold undergo an evaluation process that involves gathering additional supporting evidence for each hypothesis, and applying a wide variety of deep scoring analytics to evaluate the supporting evidence.
- Final Merging and Ranking: the remaining hypotheses are evaluated based on potentially hundreds of thousands of scores to identify the single best-supported hypothesis given the evidence and to estimate its confidence, or, in other words, the likelihood of it being correct.
- Answers Merging: Watson identifies equivalent and related candidate answers, that may have different surface forms. It does this with an ensemble of matching, normalization, and coreference resolution algorithms. It then enables custom merging and the calculation of combined scores.
- Ranking and Confidence Estimation: after merging, the system must rank the hypotheses and estimate confidence based on their merged scores. In this phase, a machine learning approach is adopted. it requires running the system over a set of training questions with known answers and training a model based on the scores.

This illustration of the different phases that a question passes through, in order to be answered, sheds some light on how the NLP tasks are performed from the different services that rely on IBM Watson™.

2.1.2 Watson™ Services

Watson™ is now accessible on the IBM Cloud platform through a wide variety of services, covering different problems related to NLP and information extraction [10]. They share the same roots, but each one of them focuses on a specific task or group of tasks:

- Discovery: this service allows to build exploration applications. It can be used to enrich proprietary, public and third-party unstructured data, to run business specific queries on it and gain insights.
- Knowledge Studio: it allows to create a machine learning model that understands the linguistic nuances, meaning, and relationships specific to the user's industry or to create a rule-based model that finds entities in documents based on rules defined by the user. These models can be used to enrich and tag documents, thus this service can be used as a first step in the integration of Watson services into an application.
- Natural Language Classifier: it uses machine learning algorithms to return the top matching predefined classes for short text inputs. It can help an application understand the language of short texts and make predictions about how to handle them.
- Tone Analyzer: it uses linguistic analysis to detect emotional and language tones in written text, both at document and sentence levels.
- Natural Language Understanding: with this service, developers can analyze semantic features of text input, including categories, concepts, emotion, entities, keywords, metadata, relations, semantic roles, and sentiment.
- Personality Insights: uses linguistic analytics to derive insights about personality characteristics of individuals from social media, enterprise data, or other digital communications.
- Speech to Text: it enables applications to derive text from audio files, allowing them to process speech conversations as if they were text messages.

- Text to Speech it provides voice-synthesis capabilities to applications.

- Assistant (formerly Conversation): this service can be used to build a virtual assistant that understands natural language input and searches for the most appropriate answer, mimicking the way in which a normal conversation goes.

2.1.3 Assistant

IBM Watson™ Assistant provides a framework for building and deploying virtual assistants, taking care of natural language understanding and retrieving suitable answers [9]. Chatbots created with this service can be integrated in external applications leveraging Assistant's APIs.

This service allows developers to configure a workspace in which the training data can be inserted, and the dialog flow can be designed. Watson™ Assistant 's APIs allow to manage the creation, update, retrieval and deletion of workspaces, intents, entities and dialog nodes, while also allowing the possibility to send messages and receive structured responses containing the selected intents and entities and the candidate answer [8].

2.1.3.1 Training Data

Training data is divided in two categories:

- Intents: they represent the goal a user has during an interaction with the assistant, therefore they are usually associated with the verbs or actions that users express. In the initial phases of the creation of a virtual assistant, all the intents that a user might have must be defined. For each intent, some example sentences must be added as training samples. Those should reflect the way in which the users of the final system will interact with the assistant (i.e. using the same lexicon or grammar constructions). Intents are identified by a unique name and they can have a short description. An illustration of intent is in Figure 2.2.

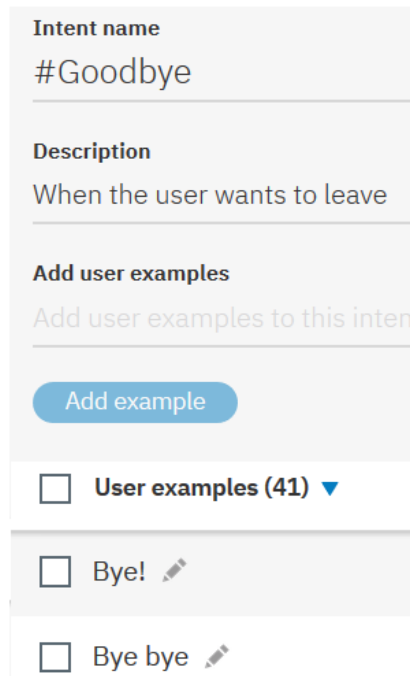

Figure 2.2: Example of intent in an Assistant workspace

- Entities: a named entity is a real-world object that can be referred to with a proper name. Entities give some additional information about the context of the analyzed text: indeed, they represent the object of a user's request and they can be used to tailor the answer to different situations. In Assistant, entities correspond to categories of terms and for each entity different values can be defined. For every value, synonyms or regular expressions can be specified in order to increment the ways in which entities can be recognized. Figure 2.3 shows how entities are presented in Assistant. If an entity value is contained in another entity value (e.g. "cat" and "red cat"), coming from the same entity category, only the longest matching value will be recognized. Some pre-built system entities that represent the most commonly used categories, such as numbers and dates, can also be added to a workspace. In order to allow some degree of flexibility in recognizing specific entities, fuzzy matching can be enabled, which checks for misspelling, partial matching and stem forms to better recognize the entities inside text messages.

Figure 2.3: Example of entity in an Assistant workspace

Assistant provides a natural language classifier trained to understand and recognize the specified intents and entities. The classifier is re-trained every time a change occurs in the set of intents or entities. When an input message is submitted to Assistant, it attempts to map the text to one of the intent of his knowledge base. It scores a confidence level for every intent, giving a higher rate to intents whose training samples are the utterances that are most similar to the input text. Assistant also looks for entity values, their synonyms and patterns inside the input text.

During the lifetime of an assistant, the messages it processes can be observed in the "Improvement" section of the service. This allows to watch how many messages are managed in different conversations and to monitor the assistant's actions. In fact, the responses and the recognized intent and entities for each input message are stored. It is then possible to correct some of the bot's interpretations, retraining the classifier in order to make it improve on the cases it misunderstood. This enables active learning, with domain experts instructing the assistant on how to classify the intent of sentences it struggles to recognize.

2.1.3.2 Dialog

The design of the dialog flow defines the way in which Assistant finds the answer for every input message. Assistant uses the results coming from the intent and entities classification phase, in order to select one of the answers that have

been prepared by the designer. After the first answer selection, the conversation can be guided by the assistant to gain additional information needed to perform a task, or it can also leave the user control the conversation.

The dialog flow is represented as a tree, as illustrated in Figure 2.4, and the assignment of each branch is to handle a specific conversation. Usually a chat starts with an intent and then moves towards its final goal with further refinements, that correspond to a better understanding of the context, thus those additional turns of the conversation focus on the recognition of entities. In each branch, nodes can be added to manage the different situations in which a request can come to, the different paths can be specified by the intents or entities that Watson recognizes or other factors such as context variables.

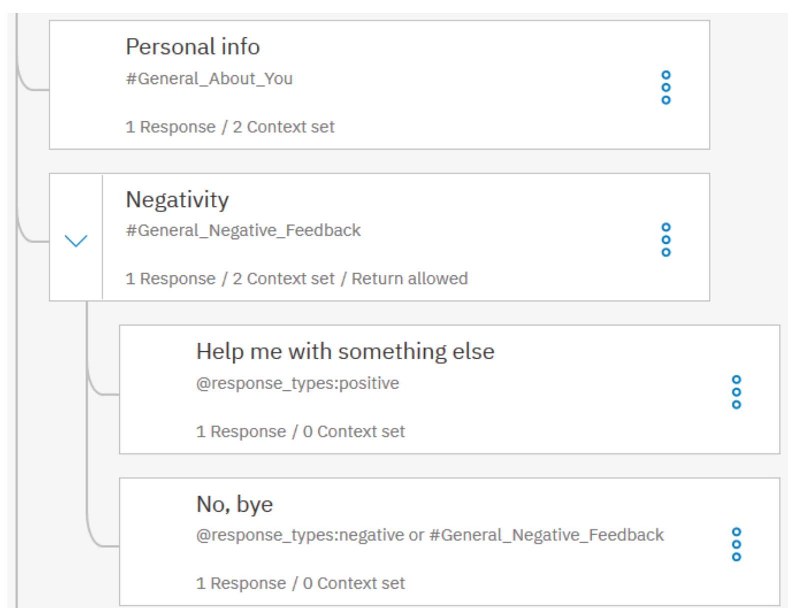

Figure 2.4: Example of dialog tree in an Assistant workspace

A dialog node must have at least one condition, that specifies when the node shall be triggered. Conditions can contain intents, entity values, and context variables values, that can be mixed using AND or OR operators. Intent names start with a "#" symbol, entity names start with a "@" symbol and context variables start with a "$" symbol. Dialog nodes also have responses, defining the answer for the input message. More than one answer can be specified, to change the way the assistant responds to the same sentence. Responses can also have associated conditions, thus reducing the need to create several dialog nodes and making the dialog tree more compact. In dialog nodes, context variables can be defined. They can be filled with values coming from recognized entities, or from external applications.

Once the first input message's intent and entities in a conversation have been recognized, the root nodes of the dialog tree are visited from the first (top) to the last (bottom), looking for a matching condition. If a condition is met, the corresponding node is triggered. If none of them matches, a special node at the bottom (thus having the lowest priority) is visited. This node recognizes a special condition, called "anything else", that accepts any input text and responds that the assistant didn't get the meaning of the message. Other special conditions are: "welcome", which is true during the first dialog turn, "irrelevant" that is true if the user's input is determined to be irrelevant by the Watson Assistant service, "true" and "false" to directly decide the outcome of the condition.

When a node has been processed, the conversation moves to its child nodes. If the following input message matches the condition of one of the children, that are always visited from top to bottom, that node is executed. Otherwise the dialog tree is traversed from the beginning, searching for the matching root node. The service continues to work its way through the dialog tree from the first to the last node, along each triggered node, then from first to last child node, and along each triggered child node until it reaches the last node in the branch it is following. This behavior is depicted in Figure 2.5.

When designing the conversation flow this behaviour needs to be considered, since the order of the branches must follow this priority mechanism. In order to find the correct answers, the dialog nodes with the most specific and restricting conditions have to be put in the higher positions, giving them a higher priority.

Nodes that address a similar subject can be organized in folders, to obtain a cleaner dialog tree. Folders can have conditions as well, so that the nodes inside a folder are executed only if the folder condition is met. Folders do not modify the order in which nodes are evaluated, they continue to be processed from first to last. When the service is exploring the tree, if it visits a folder that has no condition or a condition that is true, it processes all the nodes in the folder and then continues its search through the tree from there. Sometimes the conversation flow might need to be more complex, looping through dialog nodes or jumping from a node to another that is not in its children. This can be achieved, by specifying the action to perform when a node has been processed. The conversation can wait for the user input, skip the user input and jump directly to the first child node to evaluate its condition, or it can jump to a specified node. In the latter case, the service can wait for the user input before evaluating the target node's condition, evaluate it right away or skip directly to the target node's response.

Figure 2.5: Example of how the dialog nodes are visited

2.2 Slack

Slack is a team messaging application offering team collaboration tools and services [24]. Slack teams are organized in workspaces, where users can contact each other in persistent chat rooms (channels) organized by topic, private groups and through direct messages. All content inside Slack is searchable, including files, conversations, and people. Public channels allow team members to communicate without the use of email or group SMS. They are open to everyone in the chat provided they have first been invited to join the client. Private channels allow for private conversation between smaller sects of the overall group. These can be used to break up large teams into their own respective projects. Direct messages allow users to send private messages to a specific user rather than a group of people.

Custom applications can be added to a Slack workspace, to add some business specific functionalities. A bot user is a type of Slack application that can be added to a workspace [26]. Bot users are special kinds of users able to read and reply to conversational interactions in channels and direct messages, as can be seen in Figure 2.6. They can post text messages, upload files, images, URLs and they can also send interactive messages and dialogs to users.

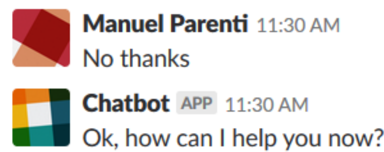

Figure 2.6: Typical text interaction with a bot user

Interactive messages can contain buttons and menus. While normal messages do not change over time, interactive ones evolve with user interactions. Message buttons are shown in Figure 2.7.

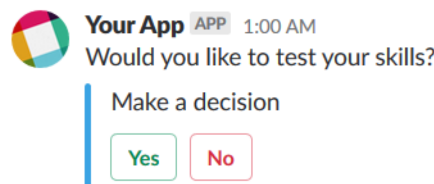

Figure 2.7: Example of a interactive message

Dialogs are special interfaces that allow applications to collect structured information. They are forms that can contain menus, text fields and buttons as shown in Figure 2.8. Instead of interactive messages, dialogs cannot be originated by normal text messages, but the users can trigger a dialog from an interaction with a message menu or button.

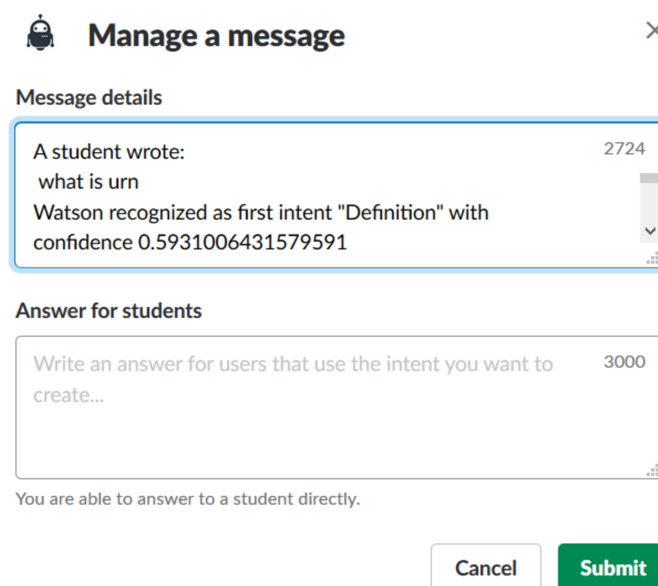

Figure 2.8: Example of a dialog in Slack

There are different ways in which an application can communicate with a Slack bot user. One of these is through the RTM API. The Real Time Messaging API is a WebSocket-based API that allows applications to receive events from Slack in real time and send messages as users. To begin an RTM session there is an authentication phase. Once connected to the message server it will provide a stream of events, including both messages and updates to the current state of the workspace.

2.3 Database Engines

2.3.1 MySQL

MySQL is an open-source relational database management system (RDBMS) [16], written in C and C++ and compatible with all major operating systems. As classical RDBMSs, it represents records as tuples, which in turn are stored inside tables. The rows of the tables are the records, while the columns are the attributes of the records. Every table has a primary key, that can be a single identifier or a combination of multiple attributes. Primary keys are unique for the table. One of the attributes that a table can have is a foreign key, that links a tuple with a tuple from another table. MySQL has a full support for indexes, allowing to make queries very fast on indexed attributes.

2.3.2 MongoDB

MongoDB is a non-relational document-oriented database management system [14]. It is free, open-source and cross-platform. It is classified as a NoSQL database program and it stores JSON-like documents, making the integration with applications that need to store and retrieve JSON objects very simple. A MongoDB document is the equivalent of a relational database's tuple. Homogenous documents are grouped in collections, that are the equivalent of tables. Every document has a unique identifier, associated to it when it is created, which is indexed by default. Secondary indices can also be created for the other fields (single or multiple) of the document.

2.4 Clustering

Cluster analysis is the task of grouping a set of objects into subsets in a way that objects belonging to the same subset are similar, and objects taken from different subsets are dissimilar. This similarity degree is specific for each context of application. For example, it can be the measure of the distance between two data points in a n-dimensional space. Clustering is an unsupervised learning technique, since it searches for groups in unlabeled data [37].

Clustering techniques can be used in many domains, an example is to divide a great amount of documents by their topic (Sci-fi, Fantasy, Economics). The formed groups should contain documents sharing the same or similar topics, depending on how the features are extracted from text. A simple model of a document can be a vector of the words it contains, with their frequency. An assumption that can be made is that documents with the same topic also share the same words, with similar frequencies. Clustering algorithms differ on many aspects, the main being the similarity measure, how data points are assigned to clusters and the convergence criteria (when the clustering process should stop).

2.4.1 Clustering Algorithms

In this section the clustering algorithms used to divide students in homogeneous groups are presented.

2.4.1.1 K-Means

K-Means is the most popular representative-based clustering algorithm. Representative-based clustering techniques represent each cluster using an entity that summarizes its characteristics. This approach makes the process easier to understand. With K-Means clusters are represented with their centroid μ, being the mean of the points in the cluster. The goal of K-Means algorithm is to find the best division of n objects in k groups, so that the total distance between the group's members and its corresponding centroid is minimized [36]. Formally, the goal is to minimize the sum of squared errors (SSE), defined as:

$$\sum_{j=1}^{k} \sum_{i=1}^{n} \left\| x_i^j - c_j \right\|^2$$

The most common algorithm, described below, uses a greedy iterative ap-

proach, following these steps:

1. Choice of the initial centroids. This can be done using different strategies, for example: random selection of the centroids, selection of points that are far from each other.

2. For the remaining points: assign every point to the cluster that has the closest centroid. To do so, the algorithm must compute the distance between all the data points and each centroid.

3. After all the points are assigned to a cluster, the values of the centroids of each cluster are updated.

4. Iterate until the clusters don't change.

K-Means has the advantage that it's fast, since it has a linear complexity O(n). On the other hand, it has a couple of disadvantages. The number of clusters needs to be chosen before running the algorithm and K-Means is dependent on the choice of the initial centroids; therefore it may yield different results on different runs.

2.4.1.2 K-Medoids

K-Medoids is a classical partitioning technique that clusters the data set of n objects into k clusters. The main difference with K-Means is that K-Medoids uses datapoints as centers of the clusters [40]. It is more robust to noise and outliers as compared to K-Means because it minimizes a sum of pairwise dissimilarities instead of a sum of squared Euclidean distances. A medoid is the object of a cluster whose average dissimilarity to all the objects in the cluster is minimal.

The most common realization of K-Medoids clustering is the Partitioning Around Medoids (PAM) algorithm that works as follows:

1. Initialization. Selection of the k initial medoids from the n data points.

2. Association of each data point to the closest medoid.

3. For each medoid m and each non-medoid o: m and o are swapped, the cost (sum of distances between points and their medoid) is recomputed. If the total cost of the clusterings has increased, the swap is undone.

4. While the cost of the configuration decreases, repeat step 3.

2.4.1.3 Hierarchical Agglomerative Clustering

Hierarchical clustering algorithms can be divided into 2 categories: agglomerative and divisive [37].

Agglomerative algorithms treat each data point as a single cluster and then successively merge pairs of clusters until all clusters have been merged into a single cluster that contains all data points. This hierarchy of clusters is represented as a dendrogram, which can be represented as a tree. Its root stands for the cluster that contains all the samples, and leaves stand for the clusters with only one sample. Hierarchical clustering does not require to specify the number of clusters before their computation, since it finds all the clusters from 1 to the number of samples. It is possible to decide how many clusters to retain, choosing when to stop the building of the dendrogram.

The steps of the algorithm are:

1. Compute a proximity matrix between all the clusters: at the beginning, each data point is treated as a single cluster, so the matrix contains the distance between each data point.

2. Find and merge the two closest clusters: the two clusters that are combined into one are those with the smallest distance, therefore they are the most similar.

3. Update the proximity matrix: since the clusters do not contain only one data point anymore, different distance metrics can be used. Average linkage defines the distance between two clusters as the average distance between data points in the first cluster and data points in the second cluster. Single linkage computes the distance as the minimum distance between a point from the first cluster and a point from the second one. Ward linkage minimizes the variance of the clusters being merged. Complete linkage, instead, computes the distance as the maximum distance between points from the first cluster and points from the second one.

4. Steps 2 and 3 are repeated until only one cluster, containing all data points, remains.

Hierarchical clustering has a time complexity of $O(n^3)$, much worse than the linear complexity of K-Means.

2.4.2 Clustering Evaluation

To assess the quality of a clustering, different validation measures have been defined and are well described in [53]. In this section some of the external validation measures are presented, as well as the silhouette coefficient, which is an internal measure. External validation measures exploit external information, from prior or expert knowledge about the clusters, such as the class labels of the data points. Internal validation measures are based on the notions of intracluster similarity and intercluster separation, thus they only focus on the distribution of the data inside clusters.

Defining a clustering partition as C and the ground truth partitioning as T, the building blocks for computing the external validation measures of Adjusted Rand Index and Fowlkes-Mallows are true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN). TP is defined as the number of pairs (x_i,x_j) where x_i and x_j belong to the same cluster in T and also in the same cluster in C. TN is defined as the number of pairs (x_i,x_j) where x_i and x_j do not belong to the same cluster in T, nor to the same cluster in C. FP is defined as the number of pairs (x_i,x_j) where x_i and x_j do not appear in the same cluster in T, but belong to the same cluster in C. FN is defined as the number of pairs (x_i,x_j) where x_i and x_j belong to the same cluster in T, but belong to different clusters in C. The total number of pairs (N) is defined as the sum of TP, TN, FP, FN.

2.4.2.1 Adjusted Rand Index

The Rand Index (RI) computes a similarity measure between two clusterings by considering all pairs of samples and counting pairs that are assigned to the same or different clusters in the predicted and true clusterings. It computes the fraction of true positives and true negatives over all the pairs, as $\frac{TP+TN}{N}$. The Rand index has a value between 0 and 1, with 0 indicating that the two data clusterings do not agree on any pair of points and 1 indicating that the data clusterings are exactly the same. The Adjusted Rand Index (ARI) is the extension of the Rand Index, corrected for chance. The correction establishes a baseline by using the expected similarity of all pair-wise comparisons between clusterings specified by a random model.

$$ARI = \frac{(RI - E(RI))}{(\max(RI) - E(RI))}$$

The Adjusted Rand Index has a value close to 0 for random labeling and exactly 1 when the clusterings are identical.

2.4.2.2 Fowlkes-Mallows

Definining precision as $\frac{TP}{TP+FP}$ and recall as $\frac{TP}{TP+FN}$, the Fowlkes-Mallows (FM) index is defined as the geometric mean of the pairwise precision and recall.

$$FM = \sqrt{precision \cdot recall}$$

Its highest value is 1, achieved when there are no false positives or negatives.

2.4.2.3 Adjusted Mutual Information

Mutual Information (MI) tries to quantify the amount of shared information between the clustering C and ground truth partitioning T. It is defined as:

$$MI(C,T) = \sum_{i=1}^{r} \sum_{j=1}^{k} p_{ij} \log \left(\frac{p_{ij}}{p_{C_i} \cdot p_{T_j}} \right)$$

where p_{ij} is the probability that a point in cluster i also belongs to partition j, p_{C_i} is the probability of cluster C_i, p_{T_j} is the probability of cluster T_j, r is the number of clusters in C and k is the number of clusters in T.

Adjusted Mutual Information (AMI) is an adjustment of the Mutual Information score to account for chance. It accounts for the fact that the MI is generally higher for two clusterings with a larger number of clusters, regardless of whether there is actually more information shared. For two clusterings C and T, the AMI is computed as:

$$AMI(C,T) = \frac{[MI(C,T) - E(MI(C,T))]}{[\max(H(C), H(T)) - E(MI(C,T))]}$$

where $H(U)$ is the entropy associated with the partitioning U, defined as:

$$H(U) = -\sum_{i=1}^{r} P(i) \cdot \log\left(P(i)\right)$$

where r stands for the number of clusters contained in U.

2.4.2.4 Homogeneity, Completeness, and V-measure

These three metrics are based on normalized conditional entropy measures of the clustering labeling to evaluate them, given the knowledge of the ground truth labels of the same samples contained in the clusters.

- Homogeneity: a partitioning satisfies homogeneity if all of the predicted clusters contain only samples that belong to a single class in the ground truth.
- Completeness: a partitioning satisfies completeness if all the samples of a given class are contained in the same predicted cluster.
- V-measure: is the harmonic mean of homogeneity and completeness.

All those scores have positive values between 0.0 and 1.0, with larger values representing the best clusterings.

2.4.2.5 Silhouette Coefficient

The silhouette coefficient is a measure of how close an object is to its own cluster (cohesion) compared to other clusters (separation). The silhouette ranges from −1 to +1, where a high value indicates that the object is well matched to its own cluster and poorly matched to neighboring clusters. In order to compute the silhouette for a clustering result, the silhouette coefficient s_i of each data point x_i needs to be calculated as:

$$s_i = \frac{\mu_{out}^{min}(x_i) - \mu_{in}(x_i)}{\max\left\{\mu_{out}^{min}(x_i), \mu_{in}(x_i)\right\}}$$

where $\mu_{in}(x_i)$ is the mean distance from x_i to points contained in its cluster y_i and $\mu_{out}^{min}(x_i)$ is the mean of the distances from $x_i$to points in the closest cluster:

$$\mu_{in}(x_i) = \frac{\sum_{x_j \in C_{\hat{y_j}}, j \neq i} \delta(x_i, x_j)}{n_{\hat{y}_i} - 1}$$

$$\mu_{out}^{min}(x_i) = \min_{j \neq \hat{y}_i} \left\{ \frac{\sum_{y \in C_j} \delta(x_i, y)}{n_j} \right\}$$

A value of s_i close to 1 indicates that x_i is close to points in its cluster and is far from other clusters. If s_i is close to 0 it means that x_i is close to the boundary between two clusters. If it is close to -1 instead, it means that x_i is closer to another cluster than its cluster, thus it may end in the wrong cluster.

The silhouette coefficient is defined as the mean s_i value across all the data points:

$$SC = \frac{\sum_{i=1}^{n} s_i}{n}$$

2.5 Spelling Correction

In the vocabulary of *Recommender Systems* there are abbreviations and complex words that students might not be familiar with, hence there is the possibility of misspelling them in text messages. Digital assistants that have automatic speech recognition capabilities already perform a spelling correction of the voice input, when transforming it to text: they pick the known words that correspond to the input signal with the highest probability. Text-based bots instead accept input messages as-is, thus they need an additional step if spelling correction is requested. Recognizing entities is an important task for a VTA, since it should be able to understand which concepts the students refer to, even when typos occur. The presented VTA is using spelling correction to enhance Watson™ Assistant's entity fuzzy matching and detect concepts of the *Recommender Systems* course in messages coming from students. Different techniques can be applied for this task, such as modeling the probability of spelling errors and use a Bayesian methodology to infer the most probable correct word [41], or leverage N-GRAM models to exploit context to find the correct words [42].

The method used to correct errors in course related words is presented by Peter Norvig in [43].

The way in which it works can be divided in four main parts:

- Language model: P(c). The probability that the candidate word c appears as a word of in a text. It is estimated by counting the number of times each word appears in a corpus of text and dividing it by the number of unique words that appear in the corpus.

- Error Model: P(w|c). The probability that a candidate word c is the word that the writer would have written instead of w. It is approximated in a very simple way: any known word with a null edit distance has the highest probability, known words of edit distance 1 have an infinitely lower probability, and known words of edit distance 2 have an even lower probability. The candidate words are produced in order of priority, if the typed word is contained in the known words it is chosen, otherwise the known words at edit distance 1 are searched and if there are none, the known words at edit distance 2 are looked for. After this search, if the candidate words are still an empty set, the original word remains untouched.

- Candidate Model. The candidate words to search inside the known words are generated by using simple edits. A simple edit to a word is a deletion

of one letter, a transposition of two adjacent letters, a replacement or an insertion of a new letter.

- Selection mechanism. Choice of the candidate with the highest combined probability.

Chapter 3

Statement of the Problem

This chapter presents the scenario in which a VTA has been created and then deployed, together with the aspects taken into account in the design of its characteristics and functionalities, which are shown in section 3.2 and further explored in the next chapter.

3.1 *Recommender Systems* at Polimi

Recommender Systems is a course of the master's degree in Computer Science and Engineering at Politecnico di Milano. It explores the leading approaches in recommender systems, from the underlying theory to their application in a real-world scenario.

Teaching is divided in different activities: students learn from an online course the core concepts and during in-class lectures the teacher's goal is to elaborate on the notions to explore them in further detail. The online course, that will be referred to as MOOC, is accessible through an official platform used for sharing teaching material and it is composed of eight modules, containing video lessons and their transcriptions. Each module has introductory and wrap-up videos with the presence of the teacher, and videos made of slides with a voice-over that explains their content.

The course's students apply what they learn during theoretical classes in a competition [22], which mimics the annual ACM RecSys challenge. Students participating to the in-class challenge are required to implement recommender algorithms, whose quality is compared against predefined baselines and against the results of the other participants. This challenge is hosted on Kaggle and it runs throughout the whole semester in which the course is held. Most of the students'

evaluation comes from it, but there are also regular oral exams for students not participating to the challenge.

Every semester in which the course is held, the teacher receives questions from his students. Some of the requests are engaging and require knowledge and expertise to find an appropriate response, while others are fairly simple to respond to. Those are questions regarding the syllabus and organization of the course, thus requiring time and effort that could be implied for replying to the more challenging questions. In the case of an online course, the number of students and questions raised by them might be significantly higher, demanding more time to reply to. *Recommender Systems* is going to be published as a MOOC on Coursera and the teacher, Paolo Cremonesi, is exploring new ways to enhance his ability to answer to both online and on-site students' requests and give them assistance rapidly and efficiently.

3.2 Problems and Chosen Solutions

The problem that the presented work is trying to solve consists in responding to FAQs regarding the organization of *Recommender Systems* and helping students to browse the material. This is realized with the employment of a VTA, accessible through an official channel for the course. This solution could be adapted to any course, giving students the possibility to interact with a virtual assistant through natural language sentences. Its introduction could be even easier in cases where an official public channel for teacher-student communication is already existing.

FAQs belong to just one of the problems that can be addressed by such a VTA, others being student engagement, student performance assessment and personalized learning. In fact, in order to perform student assessment, a virtual assistant can ask multi-choice questions about the content of a course, store the answers given by students and possibly suggest material for revision of such concepts. In a scenario in which a VTA is integrated in the same platform that delivers material such as MOOCs, the degree of personalization of the interactions with students can be even higher, leveraging information about the student's activity and engaging with him when he finishes a part of the course or has some troubles.

The main problem that afflicts the creation of such a VTA is the lack of a corpus of interactions between students and teachers. In this scenario, trying to predict what the students will ask when designing a chatbot is an arduous task. Also, understanding the meaning of the messages coming from students is no less, because

even for requests that can be predicted they can include a mix of words that could be easily misinterpreted.

A first version of the implemented solution is made accessible to students enrolled to the course in the academic year 2018/2019, to observe how they interact with it and to improve its functionalities and understanding capabilities. The VTA created for the course at issue is named Rexy, from the combination of the two words Recommender Systems. Because of the lack of data, the probability that some requests get misinterpreted by the VTA is high and it was expected that it could not reply correctly to all of the students' questions, so an active monitoring of its actions was performed, in order not to let important requests unanswered and let Rexy confuse the students. Since the communication channel of the bot is an instant messaging application, users always expect an answer. Therefore if the assistant is not very confident in the interpretation of the input message, it will either ask for clarifications or say it did not understand the request. Confidence is computed by Watson™ Assistant for every analyzed message and plays an important role in the choice of the response for that message. To guarantee a certain level of quality in the answers sent from the VTA to the learners, especially in the early phases of the experiment, the bot forwards to human teaching assistants the questions that cannot be interpreted with enough confidence. They can easily decide whether the question should or should not be managed by the bot: in the first case they can tell the bot whether its interpretation was good or wrong, and they can instruct him on how to read the request and how to respond in the future cases; in the latter case the human designed reply is sent to the user that asked the original question. This is part of an active learning strategy performed in two different ways, by asking students if they think their messages are interpreted correctly in what will be referred to as "confirmation questions", and by asking teachers their opinion on the interpretation with "teacher questions". This mechanism is essential to make the VTA continuously learn from experience and manage in a better way known cases and new cases during its lifetime.

Another important requirement of this project is the adaptability of the VTA to other courses. Indeed, a VTA that is able to converse with students from any course, with just some substitutions in his knowledge base can be arguably much more useful and interesting. This drove the design choices of this project, from the database schemas to Assistant's training data. The resulting VTA has some course-independent conversation capabilities, such as the ability to talk about the syllabus of a course and the concepts explained in the MOOC. For the first one, a simple adaptation of the responses the VTA shall give is required, but this holds

true even for *Recommender Systems*, which has small changes in its organization every year. For the latter one, there is the need to fill the knowledge base with course-specific information, as it will be discussed in Chapter 5. In the testing scenario it was necessary to include some capabilities to understand and converse about the competition. This is specific to *Recommender Systems* at Politecnico di Milano, but it can be seen as a proof of how such a VTA can be expanded and adapted to any specific course simply by following the same design guidelines of the other functionalities.

A future line of research will be focused on the personalization of the conversations, giving Rexy the possibility to engage with students based on their personal needs. One of the first steps to enable personalization is to identify groups of students and their representative profiles.

Chapter 4

Architecture

In this chapter the main components of Rexy are presented, starting from an overview of the architecture, exploring the structure of the knowledge bases and finishing with the schemas of interactions between the different modules in the different use cases.

4.1 Rexy Architecture

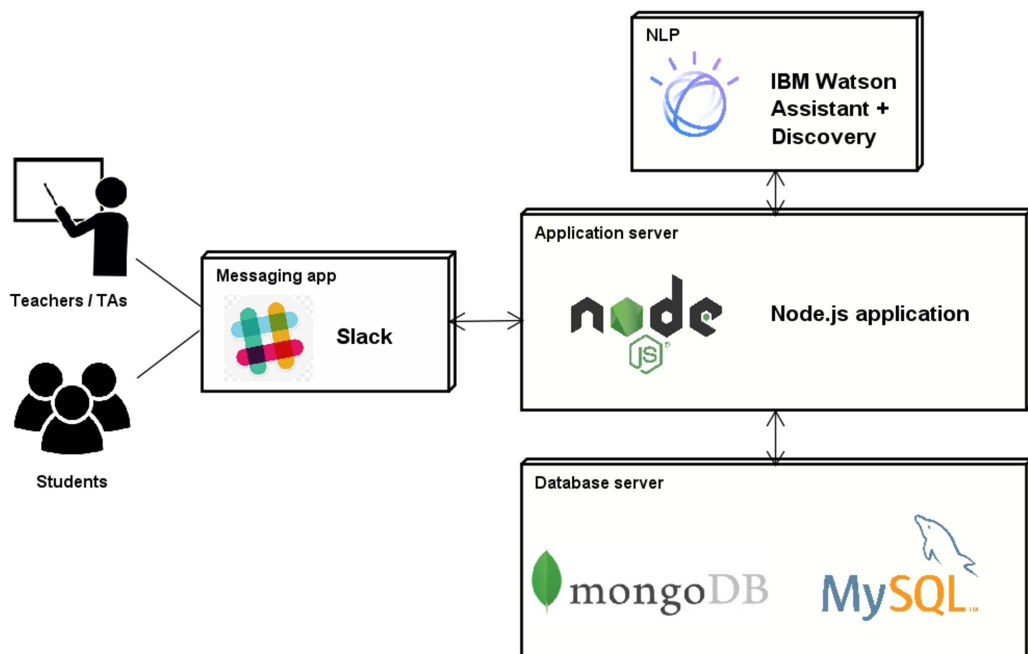

Figure 4.1: High level architecture of the VTA

The VTA presented in this thesis has different components, that are integrated to cover its various functionalities. The High-Level architecture is illustrated in Figure 4.1.

4.1.1 Front End

The front end of the VTA consists of a messaging application, which can be accessed through mobile devices and computers. The proposed VTA lives as a bot user in a Slack workspace. Slack offers a natural way of grouping communities in teams and it fits the scenario of a classroom of students, hence the choice of it for the front end. In Slack users can also chat with each other and collaborate in shared channels. In addition, Slack allows bot users to send interactive messages and dialogs, that enrich the way in which students can interact with the assistant, for instance with multi-choice questions. This platform also allows bot users to start conversations on their own, instead of waiting for a user input.

4.1.2 Application Server

The application server's job is to process the requests coming from the users and orchestrate how the NLP component and database server interact in the search of the best possible answer. It waits for any message received by the Slack application and, when one is caught, it is forwarded to the NLP module, which runs intent classification and entity recognition tasks and returns a response for the user. In some cases the NLP module does not have enough information to create a complete reply, thus it instructs the application server to retrieve the needed piece of information, which can be related to the context of the conversation or to the course itself. This server is then responsible of replying to the user and alerting human teaching assistants in the case the NLP module is not able to understand the request. These different types of interactions are explored with several sequence diagrams in Section 4.5. The application server is built with Node.js. This choice comes from the good performance of this runtime environment on real time applications that need to manage multiple simultaneous requests. Since a VTA needs to be able to handle different users writing at the same time, process their messages to understand the meaning of their requests and access knowledge bases to respond, the non-blocking nature of Node.js functions makes this framework a good fit for the job.

4.1.3 Database Server

The database server stores knowledge about the organization of the course, such as lectures, exams and challenge deadlines. This information can be easily managed by a Node.js server if it is stored in a JSON-like format, hence the choice of a MongoDB database. Another task it performs is to store the information regarding the interactions between the chatbot and the end users. This type of information can easily reach large numbers of records, thus requiring scalability. A MongoDB database is employed for this task too. There is also a database storing information about the concepts explained in the course. It follows a relational schema, so it is managed with a MySQL database.

4.1.4 NLP Component

To understand natural language utterances coming from the users and retrieve suitable responses IBM Watson™ Assistant is used through its APIs. This service has been chosen because of the capabilities it offers in understanding natural language and categorizing the intents of the input messages. It allows to define intents and design how to find the appropriate responses in an intuitive way, and it is even able to recognize intents with a limited number of examples.

Discovery is instead used to query the text documents that cover the *Recommender Systems* course, in order to answer questions regarding the main concepts of the course that are not covered by Assistant.

4.1.5 Deployment

For testing the VTA and during its activity in the *Recommender Systems* course, the application and database servers run on an Amazon Web Services (AWS) Elastic Cloud Computing (EC2) instance. The virtual machine runs Ubuntu Linux 16.04 on a single vCPU and it has access to 1 GiB of RAM and a 20 GB SSD storage.

4.2 Databases

4.2.1 MOOC Database

Information about the MOOC is stored in a relational database, where tables represent modules, concepts and chapters. The schema of this database is shown by Figure 4.2.

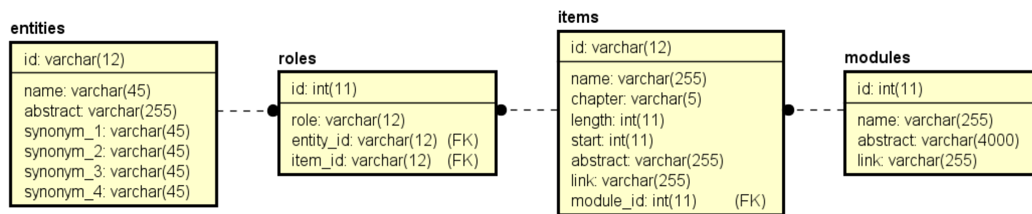

Figure 4.2: MOOC database schema

The choice of a relational database comes from the nature of the different parts of a MOOC and how they relate to each other. In fact, modules contain different chapters, each chapter talks about a number of concepts and every notion presented in the course can be explained in different chapters.

Follows a description of the different tables:

- *Modules*: represent the different modules of the MOOC. Each module has an id, a name, a URL and an abstract, that summarizes its content.

- *Items*: represent chapters and sub chapters composing the different modules of the MOOC. Each item is linked to a module through the `module_id` foreign key, it then has an id, a name, a chapter number, a length that specifies the duration of the corresponding video in seconds, a starting time which tells when the item starts in a video (useful in cases when videos contain different items), a URL and a short abstract explaining its content.

- *Entities*: correspond to the different concepts introduced and discussed during the course. Each entity has an id, a name, an abstract which gives a very short definition and four official synonyms which are part of the teacher's vocabulary used for the course, that can be seen as a subset of the students' vocabulary, which in turn contains all the possible ways in which students can refer to the entities.

- *Roles*: represent the many-to-many relationship between entities and items. They give information on the concepts explained by an item and where entities can be found inside the items, adding a role to each correspondence. Roles are the different types of occurrence that an entity inside an item can have, for example: "definition", "introduction", "example", "picture" and a special role called "main" that defines the main entity described by an item, even when this entity is not referenced directly.

4.2.2 MongoDB Databases

Other databases are used to store and retrieve information about the organization of the course, interactions between the VTA and students, and the different user. These types of data are stored in MongoDB databases, which are portrayed in the following subsections.

4.2.2.1 Administrative Database

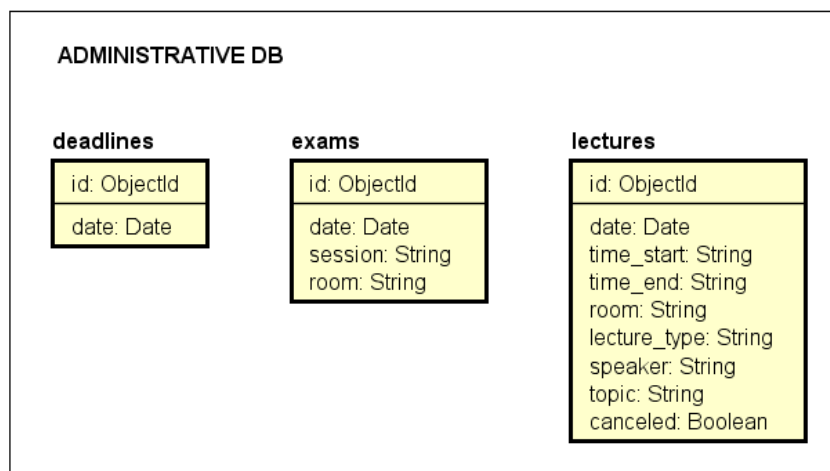

Figure 4.3: Administrative database collections

Figure 4.3 shows the so-called administrative database and its collections, which represent the main events in the organization of the course. The description of their attributes is the following:

- *Lectures*: have a unique id, a date, a starting and a finishing time (`time_start` and `time_end`), an established room where the lecture is held, a `lecture_type` field that says whether the lecture is theoretical or practical, a speaker that could be the teacher or a teaching assistant, a topic field that represents the program for that lecture, and a canceled status.

- *Exams*: have a unique id, a date, a room where the exam takes place and a session field indicating the exam session.

- *Deadlines*: have a unique id and a date.

4.2.2.2 Conversation History Database

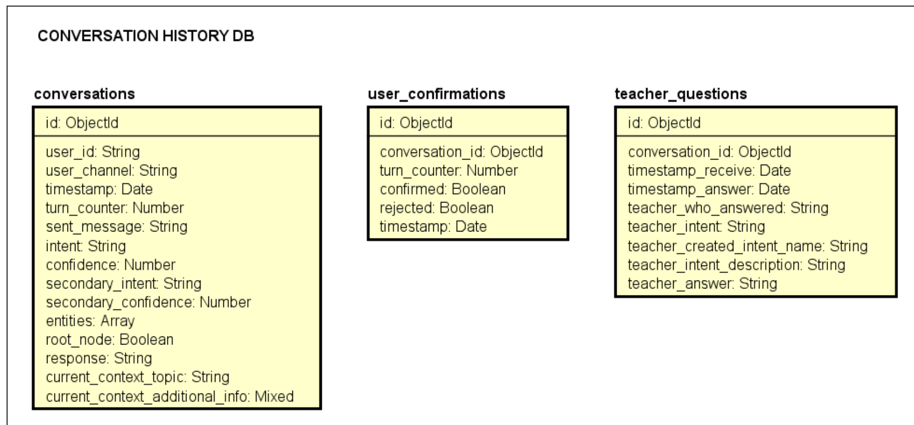

Figure 4.4: Conversation history database collections

Figure 4.4 shows the conversation history database and its collections, that represent the different types of interactions between users, both students and teachers, and the VTA. A document stored in the conversations collection represents a single turn in the conversation between a user and the VTA, documents in *user_-confirmations* reproduce the interactions between users and confirmation questions that are sent out as interactive messages, while documents in *teacher_questions* are used to store the expert opinion of teachers when asked to classify or respond to a user message that is of difficult understanding for Assistant. Their attributes are listed below:

- *Conversations*: have a unique id, Slack's id and channel of the user that sent the message, a timestamp of the event, the number of the current turn of the conversation in `turn_counter`, the text of the message that has been sent from the user in `sent_message`, the intent that Assistant recognized with its confidence, the intent with the second highest confidence in `secondary_intent` and `secondary_confidence`, the array of identified entities, a `root_node` field that indicates whether the intent recognition was useful for the answer selection if its value is set to true, the response sent to the user, the current topic of the conversation in `current_context_topic` and some additional information regarding the current context in `current_context_additional_info`.

- *User confirmations*: have a unique id, the id of the conversation's turn that

originated the confirmation question, the turn counter of such turn, a timestamp, and confirmed and rejected fields that prove the decision of the user.

- *Teacher questions*: have a unique id, the identifier of the conversation's turn from which the need to ask the teachers' opinion originated, the timestamp of the received message in `timestamp_receive`, the timestamp of the answer coming from one of the teachers in `timestamp_answer`, Slack's id of the teacher who answered. Depending on how the teacher decides to manage the request, the selected intent is saved in `teacher_intent`; the created intent, its description and the response that Assistant should give when detecting that intent are saved respectively in `teacher_created_intent_name`, `teacher_intent_description` and `teacher_answer`. If a teacher decides to reply directly to the user without making changes to intents, only `teacher_answer` is used.

4.2.2.3 User Database

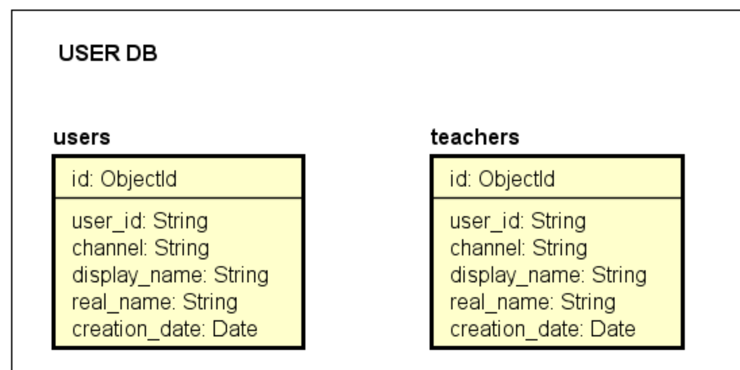

Figure 4.5: User database collections

Figure 4.5 shows the database of users and its collections, that represent the users that have interacted with the VTA and the teachers that should be contacted in case the VTA is not able to understand an input message. Their attributes are the same: a unique id, a creation date and other information that comes from Slack: `user_id`, `channel`, `display_name` and `real_name`.

4.2.2.4 Question Database

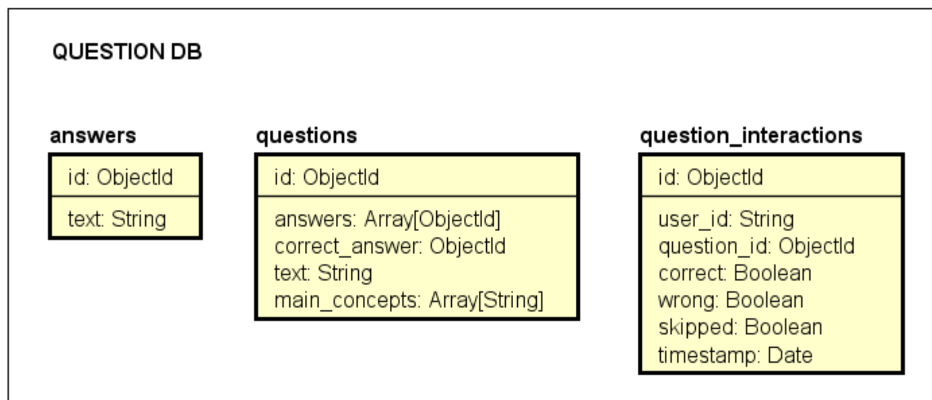

Figure 4.6: Question database collections

The question database, shown in Figure 4.6, is used to store multiple-choice questions about the course, their answers, and the responses that users have selected. The fields of those three collections are presented below:

- *Answers*: have a unique id and a text field that consists of the answer itself.
- *Questions*: have a unique id, an array of answers that represents the available choices, the id of the correct answer, the text of the question, and an array of the main entities that appear in the question, saved in main_concepts.
- *Question interactions*: have a unique id, Slack's id of the user who answered the question corresponding to `question_id`, a timestamp of the interaction, and three fields representing the result obtained by the student, correct, wrong and skipped.

4.3 Watson Assistant Workspace

In this section, the design choices for the main components of the Assistant workspace used for the VTA are presented, starting from the training data. Subsection 4.3.1 contains the list of defined intents, Subsection 4.3.2 is about the entities and Subsection 4.3.3 is about the dialog tree.

Some use cases, that show how some example messages are recognized and answered, are listed in Appendix B.

4.3.1 Intents

The intents defined for the VTA can be divided in different categories, based on the subject they are related to. This categorization is also noticeable from the names adopted for the different intents. A table is presented for each category, with the names of the intents and their description.
Table 4.1 presents intents related to the *Recommender Systems* course.
Table 4.2 presents intents related to the *Recommender Systems* challenge.
Table 4.3 presents intents related to the exam calls of *Recommender Systems*.
Table 4.4 presents intents related to the lectures of *Recommender Systems*.
Table 4.5 presents intents related to the notions explained in *Recommender Systems*.
Table 4.6 presents intents related to the teacher and teaching assistants of *Recommender Systems*.
Table 4.7 presents intents that need contextual information to find a proper answer. *#Definition*, *#Synonyms*, *#Content_References*, *#Item_Module_Description* are other intents that need contextual information, but only when no entities of the course are recognized.
Table 4.8 presents intents related to general questions that can be asked to a virtual assistant.
Table 4.9 presents intents related to general speech acts.

Table 4.1: Course intents

Name	Description
#Course_-Program	A user can ask what can be learned from this course
#Course_Topics	Users can ask a detailed list of the topics discussed during the course
#Course_Website	Users can ask if there is a website for the course

Table 4.2: Challenge intents

Name	Description
#Challenge_Allowed_-Languages	Users can ask what programming languages can be used for the challenge
#Challenge_Allowed_-Libraries	Users can ask whether they can use a library for the challenge or not
#Challenge_Baseline	Users might ask what the baselines of the challenge are
#Challenge_Deadline_Date	Users can request the date of a specific deadline, by date or by number
#Challenge_Duration	Users can ask the period of time in which the challenge is running
#Challenge_End	Users can ask the ending date for the challenge
#Challenge_Enrolling	Users can ask how to enroll to the competition
#Challenge_Evaluation	A user might ask how the grade is calculated for the challenge project
#Challenge_Find_Partner	Users might be searching for a partner for the challenge
#Challenge_List_Deadline	Users ask when all the challenge deadlines are
#Challenge_Metric	Users can ask information about the evaluation metric used to score algorithms for the challenge
#Challenge_Modality	Users can ask how the challenge works
#Challenge_Next_Deadline	The user wants to find out when the nearest future deadline is
#Challenge_Presentation	Users can ask if there is a presentation at the end of the challenge
#Challenge_Results	Users can ask the standing points for the deadlines of the challenge
#Challenge_Rules	Users can ask the rules of the challenge
#Challenge_Sharing_Code	Users can ask if they can share code with other teams
#Challenge_Start	Users can ask the starting date of the challenge
#Challenge_Website	A user can ask the website for the challenge
#Challenge_Win	Users can ask how to win the challenge

Table 4.3: Exam intents

Name	Description
#Exam_Date	Users can ask when the exams take place (with a number, a date or a session)
#Exam_Duration	Users might ask how long the exam lasts
#Exam_List	Users can ask the list of scheduled exams
#Exam_Location	Users can ask where an exam will take place (with a number, a date or a session)
#Exam_Location_-nearest	Users might be interested in knowing where the next exam will take place
#Exam_Modality	Users can ask how the exam works and how it is structured
#Exam_Next	Users want to know the date of the nearest exam call
#Exam_Topics	A user might ask the list of topics that can be part of the exam

Table 4.4: Lecture intents

Name	Description
#Lecture_Calendar	Users can ask the full schedule for the course
#Lecture_Canceled	Users may ask if a certain lecture will be canceled
#Lecture_End	Users can ask the date of the last lecture of the course
#Lecture_Info	Users can ask information regarding a specific lecture
#Lecture_Info_Nearest_-Future	Users can ask information regarding the next lecture
#Lecture_Info_Nearest_-Past	Users can ask some information regarding the last lecture
#Lecture_Nearest_-Canceled	The user wants to know if the next lecture is canceled
#Lecture_Schedule	Users can ask the schedule of the lectures
#Lecture_Start	Users can ask the starting date of the course

Table 4.5: Course content intents

Name	Description
#Content_References	Users can ask where they can find an entity of the course (e.g. URM) inside the modules
#Definition	Users can ask the definition of an entity of the course
#Item_Module_-Description	Users can ask a description (abstract) of a specific module or a specific chapter of a module or even a video of a module
#Send_To_Discovery	Users can ask questions that can be better managed by discovery
#Synonyms	Users can ask the synonyms of a particular entity or concept of the course
#Test_Knowledge	Users can ask the assistant for a question, to test their knowledge

Table 4.6: Teacher intents

Name	Description
#Teacher_Contact	Users might ask for the teacher's e-mail
#Teacher_Info	Users might ask who the teacher is
#Teaching_-Assistants_Info	Users can ask who the teaching assistants are

Table 4.7: Generic questions intents

Name	Description
#Duration	Users can ask the duration of an administrative entity of the course, e.g. an exam, the challenge, a lecture...
#End	Users can ask the ending date/time of an entity of the course, for example lectures, the challenge...
#List	Users can ask a list of entities of the course, for example a list of the exams, the deadlines or lectures
#Location	Users can ask the location of an entity of the course, e.g. a lecture, an exam...
#Modality	Users can ask how something works, e.g. how the challenge works, how the exam is done...
#Next	Users can ask when is the next event of some sort, e.g. exams, lectures, deadlines...
#Start	Users can ask the ending date/time of an entity of the course, for example lectures, the challenge...
#Time	Users can ask the date or time of an event, that can be a deadline, a lecture or an exam

Table 4.8: General intents

Name	Description
#General_About_You	Generic personal information of the chatbot
#General_Chatbot_-Capabilities	A user can ask what the chatbot can do
#General_Human_or_Bot	Ask if speaking to a human or a bot
#General_Jokes	Request a joke
#General_Negative_-Feedback	Express unfavorable feedback
#General_Security_-Assurance	Express concerns about the security of the bot

Table 4.9: General speech acts intents

Name	Description
#Conversation_Topic_-Change	Users can switch the topic of the conversation by saying "let's talk about exams now"
#Goodbye	When the user wants to leave
#Greetings	Greetings from the user
#Okay	Users can demonstrate satisfaction on a response from the bot.
#Really	Users can ask if the bot is sure about what it said
#RelatedTopics	Users can ask something related to the course, but that is not something the chatbot knows
#Thanks	Users might want to thank

4.3.2 Entities

In Table 4.10 the list of entities used to better identify the context of the input messages is presented.

Table 4.10: Entities

Name	Description
@administrative_-entity	It is used to identify if the user wants to talk about a specific entity of the course, such as the course itself, lectures, exams, or the challenge
@cardinality	It is used to detect cardinal numbers in the input messages
@chapter	It represents the different chapter numbers
@course_entity	It represents all the main entities introduced and explained in the course
@exam_session	It identifies the different exam sessions
@module	It represents the different modules, by using their number and name
@reference_role	It represents the roles that an entity can have inside a chapter
@response_types	It is used to identify yes, no and maybe in the input messages
@video	It represents the different video identifiers
@sys-date	A system entity provided by IBM, used to recognize dates inside text

In particular, *@course_entity* is very important since it defines how students can refer to concepts of the course. It can be called a vocabulary of the students, thus it should contain the official names of the concepts, as defined in the vocabulary of the course, as well as more informal synonyms that can be used by students in their messages.

4.3.3 Dialog Tree

The dialog tree has several branches, this section focuses on the guidelines followed in the design phase, rather than the specifications of each dialog node. The complete dialog tree is placed in Appendix A.

4.3.3.1 Dialog Nodes and Branches

There are two types of dialog nodes in the dialog tree:

- Dialog nodes with predefined responses: if a node is triggered, the answer sent to the user is the one defined for that node.
- Dialog nodes requiring external information: if one of these nodes is triggered, a request for some specific information is sent to the application server. Then, the server can send the final answer to the user or it can return the requested information to Assistant, which is able to capture it in a child node and proceed with the current branch.

Nodes with predefined responses are added to the dialog tree to handle requests that don't require access to a knowledge base. For example, nodes that are triggered when a general intent is identified all have a static response. Defining more responses for the same node allows to send a different answer to the same question in different occasions, making the bot more interesting to converse with.

Nodes that need external information can ask the server for data regarding the organization of the course (lecture dates, deadlines, exam dates) or its content (definitions, occurrences in the modules, modules and chapters). Another type of information that can be requested is related to the context of the conversation. In fact, intents in Table 4.7 need to know what the current topic of the conversation is, in order to provide an adequate answer. Context management is covered in Subsection 4.4.2.8.

For every root node that does not trigger for general questions or speech acts, there is a twin node with higher priority whose task is to acknowledge the intent of the user's message. When Assistant recognizes an intent with a low confidence level, it asks the user to know if it got the right meaning. These twin nodes have the same conditions of their siblings, plus a special one related to the confidence level of the intent that has been classified as the most relevant. They have two child nodes, bearing a simple condition, that is to identify a *@response_types* value. When they are executed, a yes-no question is prompted and when the user responds, the conditions of the two child nodes are evaluated. If the user gives a positive answer, the bot acts as if the interpreted intent was the right one, otherwise the bot asks the user to reformulate his request. In the proposed VTA the minimum confidence level used to answer right away is 75%. This choice allows the VTA to respond when it is very confident, during the design phase Assistant was usually able to understand the right meaning even with lower confidence scores, but in order to be more cautious in a real scenario, the threshold was put to 75%. At the other end of the spectrum, when Assistant gave a confidence score near or below 30% it was replying with wrong answers most of the time. Thus, within this

interval of scores, Assistant relies on disambiguation questions, showing what it understood to the user and expecting his feedback.

At the top of the tree, there is a special node that filters messages based on the confidence level of the first recognized intent. If its score is lower than 30%, the bot skips all the other root nodes and jumps to the last one, which condition is *anything_else* and responds that the bot is not able to understand the meaning of the request. In the case that an entity of the course is recognized, but the confidence level for the intent is lower than 30%, the definition of that entity is sent to the user. If a date has been recognized instead, a special node is triggered and contextual information is requested to the server, in order to understand what the date refers to.

Two types of branches are present:

- A branch containing only one leaf node, used to respond to one-shot questions. Its condition consists of an intent that does not require contextual information or additional entities to fully understand the request from the user, thus when it is executed, the response can be found or created in one step. For example, when the *#Greetings* intent is recognized, the bot can reply with its salutations, because it does not need any other kind of information in order to respond, and then wait for any kind of request the user might ask.

- A more complicated kind of branch, with several child nodes, that guides the conversation with the user. Such branches are used to gain information from the user, to perform a final task, and require the submission of some type of message. The bot can make yes-no requests, as well as more open-ended ones. For example, if the user wants a definition, the bot shall ask the entity of the course to look an abstract for. If he responds with its selected entity, the bot can start his search and create the proper answer. This is achieved by creating a child node from the node asking for this piece of information, that recognize the @*course_entity* entity.

In order to give instructions to the application server, nodes populate specific context variables, which are listed in the following subsection.

4.3.3.2 Context Variables and Keywords

There is one main context variable used for communications between Assistant and the application server, which is called *action*. It is a JSON object and the

properties it can have are in the following list:

- `confirmation`: true when asking a disambiguation question to the user. It is used when the confidence level is lower than 75%.
- `ask_teacher`: if true it indicates the need of sending the received message to a teacher. It is used only to force the server to ask for the teacher's help.
- `ask_confirm`: true in nodes that have an intent in the condition. It instructs the server to ask the user if the bot's interpretation is right, in a confirmation question.
- `no_ask_confirm`: is used in nodes that, after their execution, have a jump to another node which contains ask_confirm. Its task is to nullify the effect of `ask_confirm`.
- `lookup`: instructs the server on the information to look for in the databases. Admitted values are: "exam_retrieval", "challenge_retrieval", "lecture_retrieval", "definition", "synonyms", "occurrences", "module", "chapter", "video", "joke".
- `check_context`: true in nodes that need contextual information to select the answer for a question.
- `send_entities`: true in nodes that ask the user to select one of the notions of the course, to complete their task. When the server detects this property, it sends a list of the available concepts.
- `send_question`: true when the bot is required to ask a multi-choice question to the user.
- `discovery_search`: true when the bot detects the intent *#Send_To_Discovery* and executes the corresponding node.
- `input_text`: contains the message sent from the user. It is only used in the particular case in which the input message has to be analyzed from the server and there has been a confirmation question.
- `append_response`: if true it indicates that the server is required to fill the answer and send it to the user without responding to Assistant.
- `append_list`: is the same as `append_response`, but for lists of content.
- `exam_session`: contains the recognized entity *@exam_session*.

- `exam_date`: contains the recognized entity `@sys-date` when the conversation is about exams.
- `room_only`: true when the user wants to know the location of an event.
- `exam_number`: contains the recognized entity @*cardinality* when the conversation is about exams.
- `next_exam`: true when the user wants information about the next exam.
- `next_from_date`: true when the user wants information about the next event, starting from a specific date.
- `date`: used to specify the date associated to `next_from_date`.
- `list_of_exams`: when true it signals that the user has requested all the exam dates.
- `next_deadline`: true when the user wants information about the next deadline of the challenge.
- `deadline_date`: contains the recognized entity @*sys-date* when the conversation is about challenge deadlines.
- `deadline_number`: contains the recognized entity @*cardinality* when the conversation is about challenge deadlines.
- `lecture_date`: contains the recognized entity @*sys-date* when the conversation is about lectures.
- `canceled`: is set to true when the user wants to know if a lecture is canceled.
- `next_lecture`: true when the user wants information about the next lecture.
- `last_lecture`: true when the user wants information about the last lecture.
- `course_entity`: contains the recognized value of @*course_entity*.
- `role`: contains the recognized value of @*role*.
- `module`: contains the recognized value of @*module*.

- `chapter`: contains the recognized value of @*chapter*.
- `video`: it contains the recognized value of @*video*.
- `ask_definition`: it is set to true when the server has to perform a lookup and then ask the user if he wants to know a definition of a concept of the course.
- `ask_occurrence`: it is set to true when the server has to perform a lookup and then ask the user if he wants to know where to find a concept of the course inside the modules.
- `intent`: it indicates the top classified intent. It is used when an acknowledgment question is sent to the user and contextual information is needed.
- `input_date`: it contains the recognized @*sys-date*, when contextual information is needed to understand what type of event the date might refer to.

Another context variable, called `topic`, is used to classify the current topic of the conversation and inform the server of it. The available topics are: "exam", "lecture", "challenge", "challenge_deadline", "course", "content", "items", "generic", "greetings", "confirmation", "joke".

Other context variables are used to maintain a sort of state in branches of the dialog tree, in which the leaf nodes need some information that has been identified in the root or parent nodes, in order to complete their task. Those are: *temp_module*, *temp_chapter*, *temp_video*, *temp_entity*, *temp_role*, *temp_input*, *temp_date*, *temp_session*, *temp_number*. Their task is to store the value of the corresponding entity that has been recognized, to be used in a dialog node that is at a lower level in a branch.

4.4 Node.js Application

4.4.1 Dependencies

Node.js applications can leverage external resources to perform common tasks, which have already been addressed by different communities. With npm, which is "the package manager for JavaScript and the world's largest software registry" [18], one can import free and open source libraries as local dependencies of a Node.js application, enabling reuse of public code. The main packages that have

been installed in the application server are Botkit, Watson Developer Cloud, Mongoose and mysql.

Botkit is a "developer tool for building chat bots, apps and custom integrations for major messaging platforms" [3]. It is used as a middleware between the application and the Slack front end, to receive text message and interactive message events from Slack, and to send responses to users.

Watson Developer Cloud allows to call Watson™ 's APIs from Node.js applications [10]. It is leveraged to communicate with the Assistant workspace and with Discovery.

Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js [15]. It is used to query, insert and update documents contained in the MongoDB server.

mysql is a Node.js driver for MySQL [17]. It is used to connect to the MySQL server.

The process manager for the deployed version of the VTA is pm2 [20], it does automatic load balancing and monitoring of the application server.

4.4.2 Components

The first part of this section presents the classes used to model information relevant for the application. The second part focuses on the most important part of the application server, the modules. They use external libraries and the basic classes to integrate all the different components present in the architecture of the VTA. Every basic class has a module that exports the class itself. Those modules are required to let other modules import and perform tasks involving the use of those classes. For each category of information there is a manager module (a sort of domain expert), which is used to abstract the connection to the related database and the actions needed to store and retrieve the information requested by other modules. Higher-level modules are only required to handle the conversation between the users and the VTA, leaving the details of how information is retrieved to the corresponding managers.

4.4.2.1 Classes

Objects stored in MOOC, administrative, user and question databases have a corresponding class in the Node.js application. In the figures below the classes and their attributes are shown, their methods are omitted for brevity since only getters

and setters are present. Classes related to the MOOC database are shown in Figure 4.7.

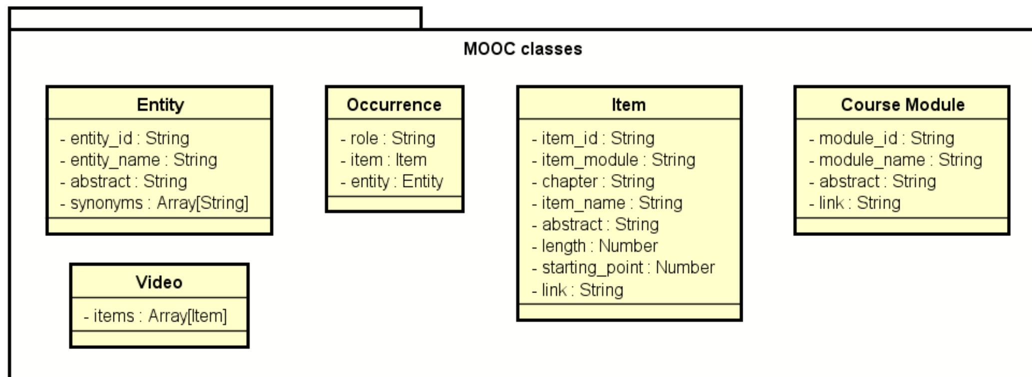

Figure 4.7: Classes related to the MOOC database

The correspondence between classes and the database tables is straightforward, except for Occurrence that corresponds to the *roles* table and *Video* that is just a container of items. Ideally each item should have its own video, but this addition is helpful in cases where short items are collected in the same video.

Classes corresponding to the collections of the administrative database are shown in Figure 4.8.

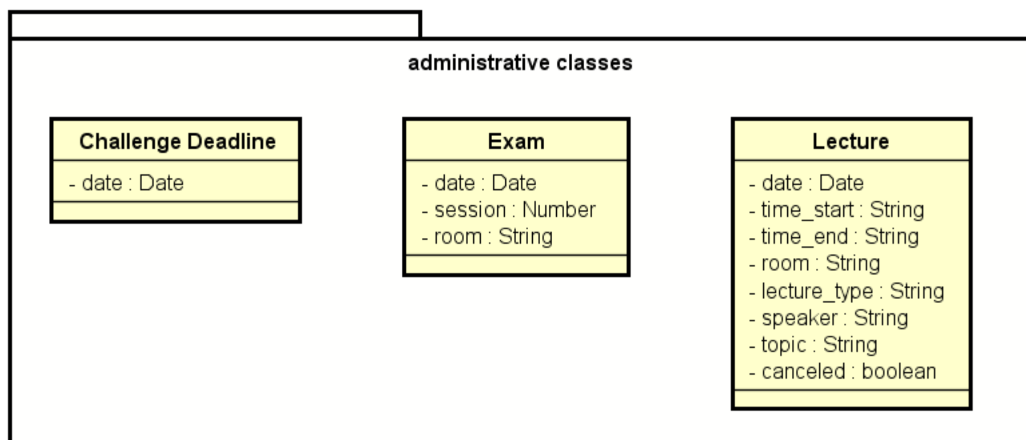

Figure 4.8: Classes related to the Administrative database

Classes created to handle questions are shown in Figure 4.9.

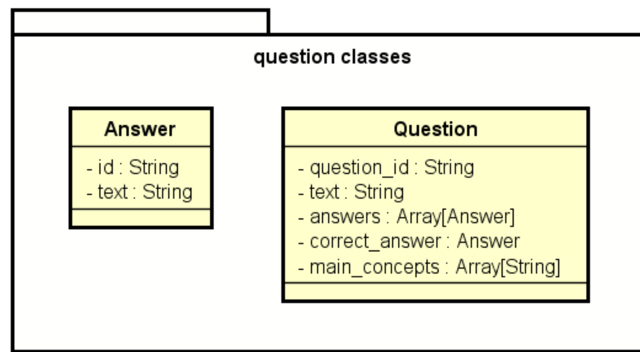

Figure 4.9: Classes related to the Question database

A single class, called *User*, is present to manage both teachers and students. It is presented in Figure 4.10.

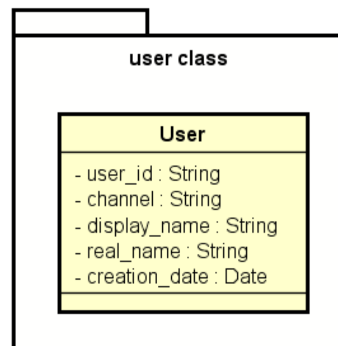

Figure 4.10: Class related to the User database

To represent the context of a conversation and its properties, another class is used. It has a `topic` attribute which corresponds to the `topic` context variable set in dialog nodes, a timestamp and the turn counter of the last time it has been used, a `current` attribute that says if an instance of Context is the currently used one in the conversation. The `additional_info` attribute stores all the useful information that can be accessed when the current context determines the answer to give to a user. It can contain an instance of *Exam*, *Challenge Deadline*, *Lecture*, *Entity*, *Item*, *Module* and *Video*. The methods of this class are all getters and setters, so they are omitted for brevity. This class is shown in Figure 4.11.

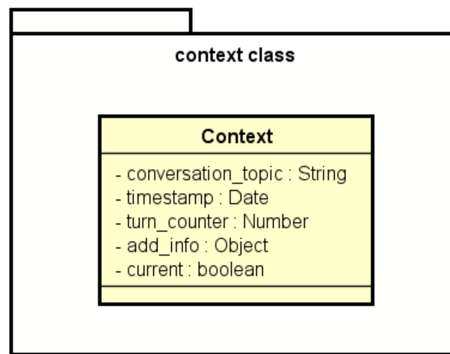

Figure 4.11: Context class

A last class is *spellChecker*, shown in Figure 4.12. It is the one used to perform spelling correction of input messages. It follows the spelling correction technique explained in Section 2.5.

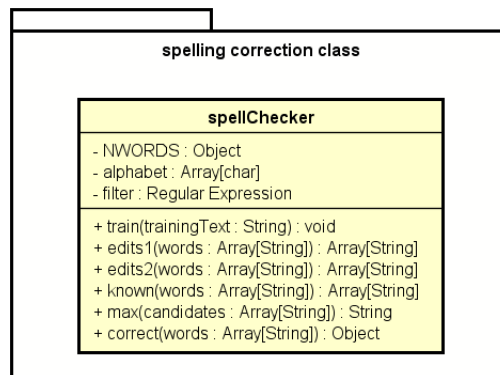

Figure 4.12: Class used as spell checker

Its methods are:

- `train`: extracts words from a training corpus, using a regular expression to find them. It stores in NWORDS the number of times that each word compares in the corpus of text.
- `edits1`: finds all the words that are reachable from the input words, with one edit. The modifications that are applied to the input words contain splits, deletions, transpositions, replacements and insertions.
- `edits2`: finds all the words that are reachable from the input words, applying `edits1` two times.
- known: selects the words that are known (the ones appearing in the training corpus) from the edited words.

- `max`: selects the best candidate word from the list returned from the edits.
- `correct`: returns an object that indicates the candidate correction for each input word.

4.4.2.2 MOOC Modules

Figure 4.13 shows the modules that handle MOOC related information, and their dependencies.

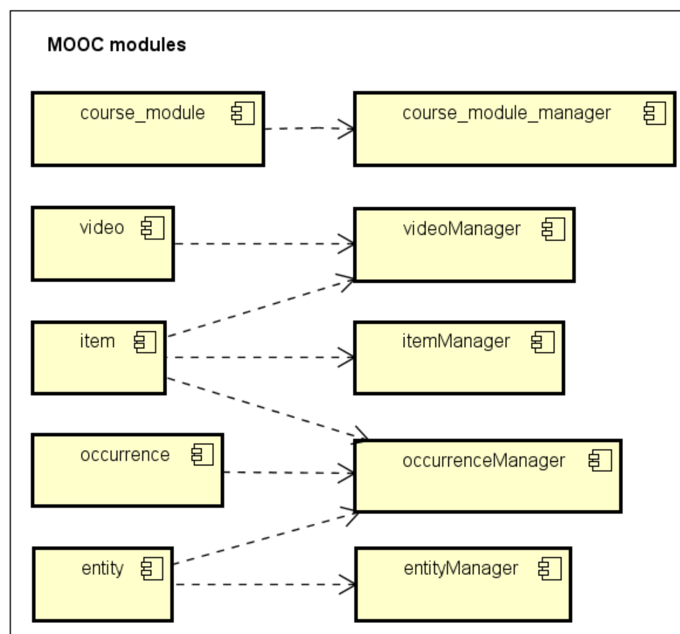

Figure 4.13: MOOC modules

- *course_module_manager*: exposes the modules of the MOOC stored in the MOOC database through two functions:
  - `getAllModules(callback)`: retrieves the list of all available modules of the course and makes them available to the callback function.
  - `getModuleByID(id, callback)`: retrieves the module of the course corresponding to `id` and makes it available to the callback function.
- *videoManager*: exports a function used to obtain information regarding a video of the MOOC. It is defined as `getVideoByModuleAndVideoID(module_id, video_id, callback)` and after looking for the video inside the MOOC database it makes it available for the callback.

- *itemManager*: can be used to retrieve information about all or specific items of the MOOC. The functions exported by this module are:
  - `getAllItems(callback)`: used to obtain the list of all the items.
  - `getAllChapters(callback)`: used to obtain the list of items that are chapters and not sub chapters.
  - `getItemByID(id, callback)`: used to obtain the item corresponding to a specific identifier.
  - `getItemByModuleAndChapter(module_id, chapter, callback)`: used to obtain the item of a specific module and chapter.
- *occurrenceManager*: exports methods that find the occurrence of an entity inside items from the MOOC. Its functions are:
  - `getOccurrencesByRole(entity_id, role, callback)`: finds the items that contain the entity with `entity_id` as identifier, with the specified role.
  - `getAllOccurrences(entity_id, callback)`: finds all the items that contain the entity with `entity_id` as identifier.
- *entityManager*: exports methods that retrieve information regarding entities of the course from the MOOC database. Its functions are:
  - `getAllEntities(callback)`: retrieves all the entities contained in the database.
  - `getEntityByID(id, callback)`: retrieves the entity that has *id* as identifier.
  - `getEntityByName(name, callback)`: retrieves the entity corresponding to the specified entity name.

4.4.2.3 Administrative Modules

There are three modules used to handle information regarding the administration of the course (exams, lectures and challenge deadlines). They are represented in Figure 4.14, with their dependencies.

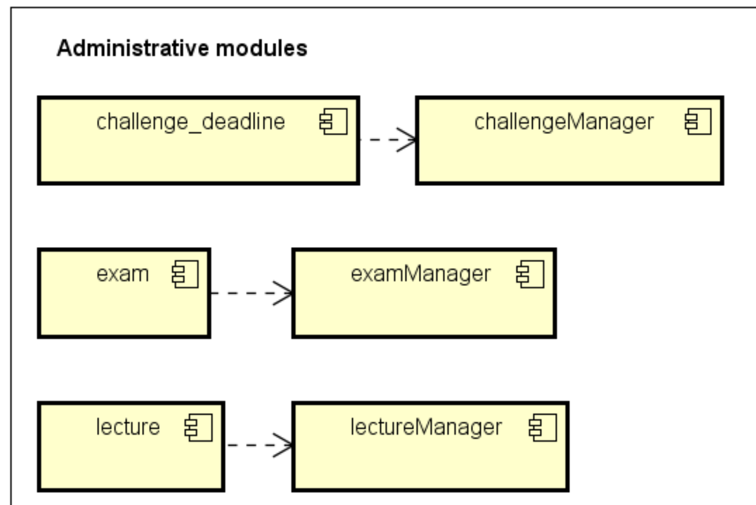

Figure 4.14: Administrative modules

- *challengeManager*: performs queries on the administrative database to find information regarding the challenge.
  - `getAllDeadlines(callback)`: retrieves the list of challenge deadlines.
  - `getDeadlineByDate(date, callback)`: retrieves the deadline that is the nearest to the input date.
  - `getDeadlineByNumber(number, callback)`: retrieves the deadline number `number`.
  - `getNextDeadline(callback)`: retrieves the nearest future deadline.
  - `getNextDeadlineByDate(date, callback)`: retrieves the nearest future deadline starting from the specified date.
- *examManager*: performs queries on the administrative database to find exam related information.
  - `getAllExams(callback)`: retrieves the list of scheduled exams.
  - `getExamByNumber(examNumber, callback)`: retrieves one exam, chosen by number of exam call.
  - `getExamByDate(date, callback)`: retrieves the exam closest to the specified date.

  - `getExamBySession(examSession, callback)`: retrieves the exams scheduled for a specified session.
  - `getNearestExam(callback)`: retrieves the nearest future exam.
  - `getNearestExamByDate(date, callback)`: retrieves the nearest future exam, starting from the input date.

- *lectureManager*:
  - `getLectureByDate(date, callback)`: finds the lecture that is the nearest to the input date.
  - `getNearestLecture(callback)`: finds the nearest future lecture.
  - `getLastLecture(callback)`: finds the nearest past lecture.
  - `getNearestLectureByDate(date, callback)`: finds the nearest future lecture, from the input date.

4.4.2.4 Question Modules

The module that handles multi-choice questions and the answers coming from students is *questionManager*. Its dependencies are shown in Figure 4.15.

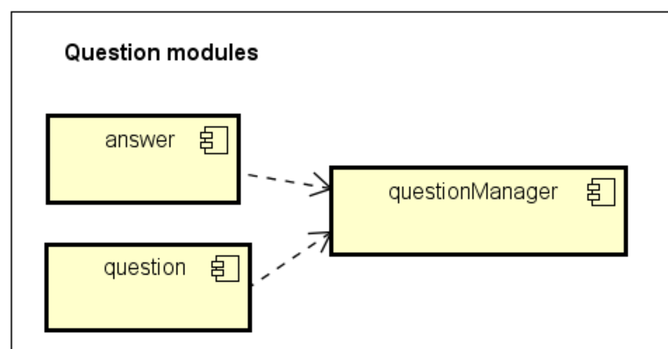

Figure 4.15: Question module

The functions exported by this module are:

- `getQuestion(user_id, callback)`: finds a question that was never presented to a user.
- `getQuestionByID(question_id, callback)`: finds the question corresponding to the input id.

- `getQuestionsSolvedByUser(user_id, callback)`: retrieves all the questions that the specified user has answered.

- `checkAnswer(question_id, answer, user_id, callback)`: it determines whether the user has got the right answer for a question.

- `saveInteraction(user_id, question_id, correct, wrong, skipped)`: saves the result that the user got with its answer to a question.

4.4.2.5 User Modules

Users are managed by two different modules: *userManager* and *teacherManager*, as shown in Figure 4.16.

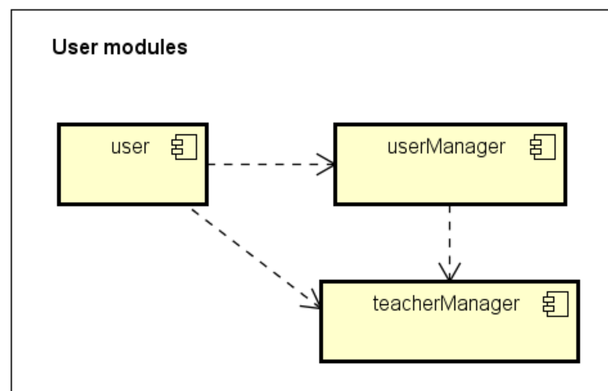

Figure 4.16: User modules

- *userManager* can query and insert users in the user database. The functions it exports are:
  - `getAllUsers(callback)`: used to obtain the list of users who interacted with the VTA.
  - `getUserByID(id, callback)`: finds the user with *id* as identifier.
  - `getChannelByID(id, callback)`: finds the Slack channel of user with identifier *id*.
  - `createUser(id, channel, display_name, real_name, callback)`: saves the new user inside the user database.

- *teacherManager* can be used to retrieve information regarding teachers and send them custom messages when required. The functions it exports are

the same as the ones from *userManager*, in addition to `sendToTeachers(user_conversation, bot)` which is used to ask for the expert opinion of all the teachers about how to manage a conversation. This function sends an interactive message using the input bot, which reports the message that the bot was not able to interpret with a high confidence. Interactions between teachers and these interactive messages are handled in the *chatbot* module.

4.4.2.6 Spelling Correction Modules

Figure 4.17 shows the *spellingManager* module, which handles the correction of input sentences. It exploits the methods of the *spellChecker* class to repair misspellings. It trains an instance of such class with a corpus of text that consists of the transcription of the modules of the MOOC. It exposes a `checkSentence` method that takes as input a message and, if necessary, corrects the single words contained in it.

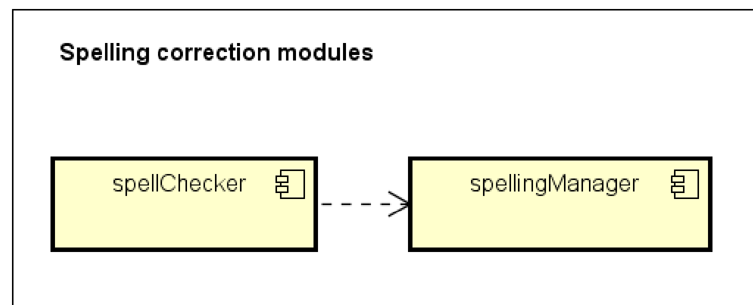

Figure 4.17: Spelling correction modules

4.4.2.7 Watson Modules

To connect to the instances of Watson™ Assistant and Discovery two distinct modules are employed, as shown by Figure 4.18.

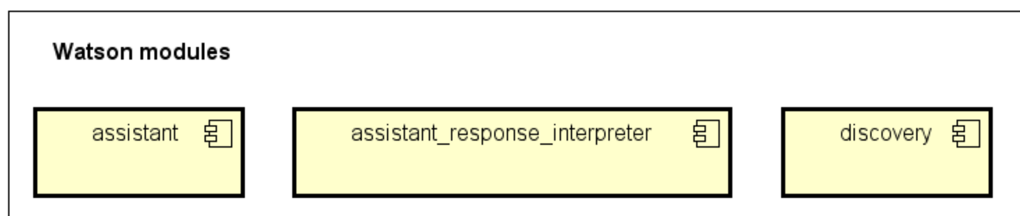

Figure 4.18: Watson modules

- *Assistant*: connects to the Watson™ Assistant workspace and exposes methods to communicate with it through the watson developer cloud APIs. Those methods are:
  - `sendMessage(text, context)`: sends a user message to Assistant, alongside the context. In this scenario, the latter variable refers to the situation in which the current conversation lives. It stores information regarding the visited dialog nodes, the turn counter, context variables and the identifier of the conversation. It is a very important parameter because it allows to continue conversations from where the user left. When Assistant has finished classifying intents, entities and has found the dialog node to execute, it sends back a response, which is interpreted using the *assistant_response_interpreter* module since it is a complex object.
  - `createExample(intent, text)`: adds `text` as a training sample for `intent` in the Assistant workspace.
  - `listIntents()`: is used to retrieve the list of intents and their training samples.
  - `createIntentWithOneExample(intent, description, example)`: creates a new intent with the input name, description and example.
  - `createDialogNode(name, conditions, output, title)`: creates a dialog node with the specified title, name, conditions and output.
  - `getIntent(name)`: retrieves the intent with the input name.
  - `deleteExample(intent, example)`: deletes the training sample corresponding to `example` from `intent`.
- *assistant_response_interpreter*: is used to mask the structural details of Assistant's responses. It has methods to get recognized intents, entities and context variables from them.
- *Discovery*: connects to the Discovery instance and exposes the method used to query the documents of the course. Such function is `query(text)`, and it sends a natural language query to be executed on the document collection that has been uploaded on Discovery. The service responds with a list of or-

dered results, corresponding to the most relevant documents for the specific query.

4.4.2.8 Context Manager

When Assistant recognizes an intent that needs contextual information in order to provide an answer, the application logic detects it through assistant response interpreter and asks *contextManager* to retrieve the needed data. As shown in Figure 4.19, *contextManager* leverages other modules to fulfill the requests he receives.

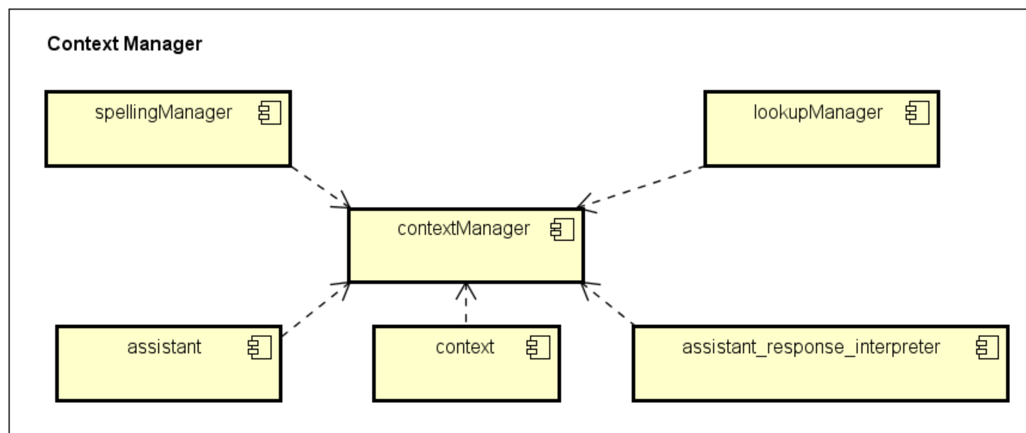

Figure 4.19: Context Manager module

Contextual information is stored for every user and it keeps track of the topics (which correspond to the `topic` context variable defined in Assistant) that appear in the conversation with the VTA. It is represented as an array of *Context* instances, where each component stands for a different topic. The `add_info` property helps to define what the user is interested in, since it can contain complex objects such as instances of the classes that were previously described.

When a request to look for context to disambiguate the meaning of an input message is received by the application server, *contextManager* looks inside an array of rules with the `manageContext` method, which receives as input the response from Assistant, the array of contexts for the user that sent the original message and a callback function to invoke when the search of the context is finished. For each of the intents contained in Table 4.7 a set of rules is defined, each of those lists the kinds of contexts that are allowed for the specific intent. For example, a temporal intent makes sense only with certain conversational topics, such as exams, lectures and deadlines. Thus, if Assistant recognized *#Time* as primary intent, a context with one of those topics is looked for inside the array of contexts.

In such array, only one of the contexts is set as current, and has a greater priority with respect to the others. If the current context is not of the type the manager is looking for, the other components of the array are examined, giving a higher priority to the youngest ones, that can be detected by their timestamp or turn counter. In this search, only contexts that stay below an age threshold (in terms of time and turn counter with respect to the current time and turn counter of the conversation) are taken into account. The current context has to be updated when it does not fall inside the allowed context topics of an intent. If another context is found eligible to become the current one, they have to be swapped. If a context is found, a proper answer is created, otherwise the VTA will respond to the user that it needs more information in order to reply. Continuing the example, if the user was talking about exams with the VTA and asks for the nearest one in the future using *#Next* as intent, *contextManager* simulates a request to *lookupManager* as if the user asked for “the next exam” and then responds to the user. The answer creation may be more complex, depending on the `add_info` of the context found. In fact, if after receiving an exam date the user writes “and the next?” falling again in the *#Next* intent, `add_info` already contains an instance of the class *Exam*, and this drives towards a different request to *lookupManager*, since the user wants to know the nearest exam after a certain date. In this simple example, after the context lookup, the response is sent to the user directly. In other cases, the VTA can ask a disambiguation question to be sure of the meaning of the message. Thus, the assistant module is leveraged to understand the answer of the user, which tells if the bot’s interpretation of the original message with the aid of contextual information is correct or wrong, and proceed with the conversation.

This simple representation of context in conversations could benefit from external information, such as the history of interactions between users and content of the MOOC. This kind of data could be used to better understand what the users should have studied and exclude information from the remaining parts of the course in the search for definitions and references.

4.4.2.9 Lookup Manager

When Assistant requests information that is stored in one of the databases, the application logic understands it with the help of assistant response interpreter and asks *lookupManager* to perform the queries needed to form complete answers. Figure 4.20 shows the modules that *lookupManager* leverages in order to get data regarding the organization of the course and its content.

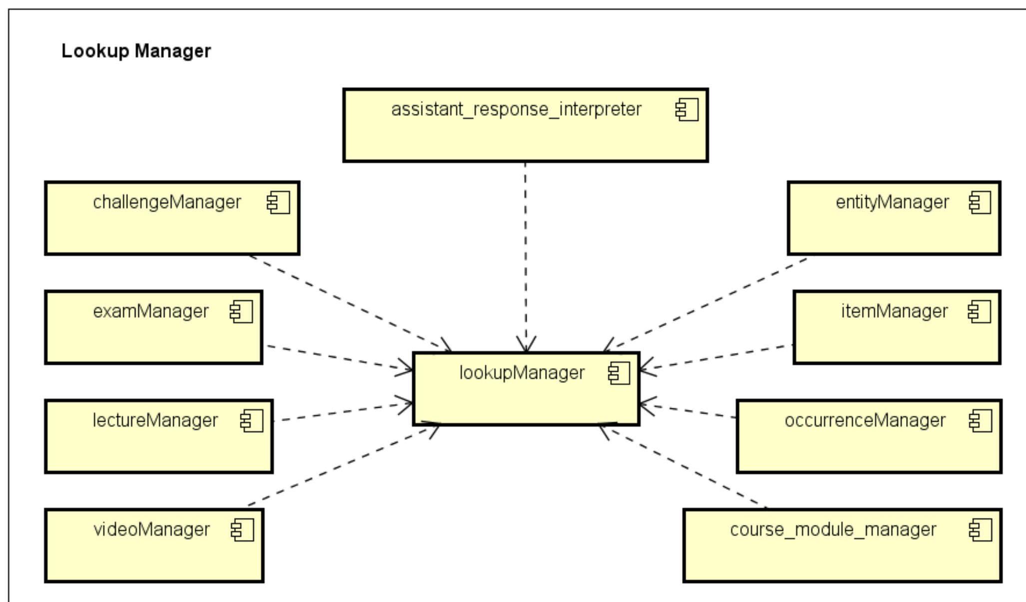

Figure 4.20: Lookup Manager module

There is one main function that handles lookups, and it receives as input the response from Assistant, the array of contexts and a callback, such as *contextManager*. When a lookup is requested, assistant response interpreter drives the decision of the query to perform, thanks to the keywords of the *action* context variable defined in dialog nodes. Depending on the type of information needed, *lookupManager* calls functions from the managers contained in its dependencies. When it receives the data from them, it can either compose an answer and invoke the callback or it can send information back to Assistant and let the dialog nodes create the final answer for the user. This behavior is decided by the dialog nodes that request the lookup with the `append_response` keyword in the `action` context variable. In this project, `append_response` is always set to true for two reasons, flexibility in the composition of answers and a significative reduction in the number of API calls to Assistant.

4.4.2.10 Chatbot

Chabot is the core component of the application, it receives user messages, calls Assistant to analyze them, performs the actions required to send a complete answer to user and responds. It handles normal text messages as well as Slack interactive messages and Slack dialogs. It also manages the context of the conversations between users and the VTA. All the dependencies of this module are shown in Figure 4.21.

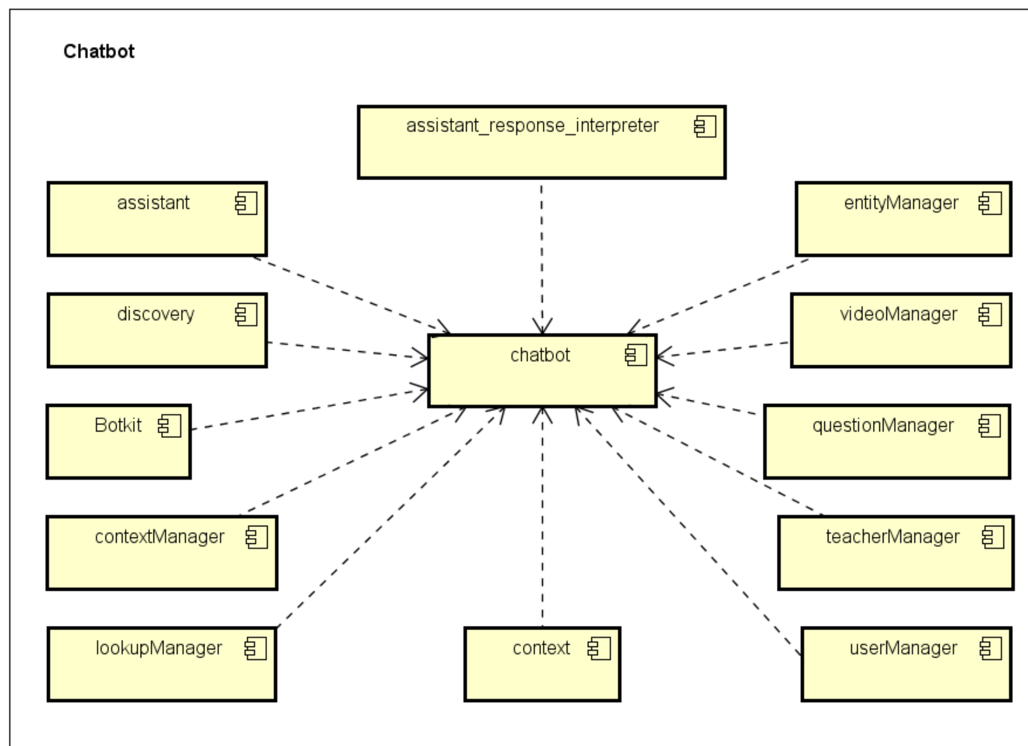

Figure 4.21: Chatbot module

Chatbot leverages Botkit's functionalities to create a slackBot (also called Slack controller), a middleware used to receive and send events to a bot user residing on a Slack workspace. Another task that the slackBot performs is to maintain a state for each user, saving contextual information and data needed from Assistant to continue the conversations from where users left off. This module comprises listeners for direct messages, interactive messages and dialogs coming from Slack, as well as a function that directs input requests towards the other modules that are able to fulfill them. This function is defined as `manageWatsonResponse(message, response, bot, conv_contexts, message_type)` and it is called after Assistant has classified an input message, to decide whether to perform a lookup, check the context or perform other special tasks.

Sequence diagrams in the following section explore in greater detail the passages that a message goes through when it is received by the application server.

4.5 Sequence Diagrams

4.5.1 High-Level Interactions

At a very high level of abstraction, interactions between users and the VTA are shown in Figure 4.22.

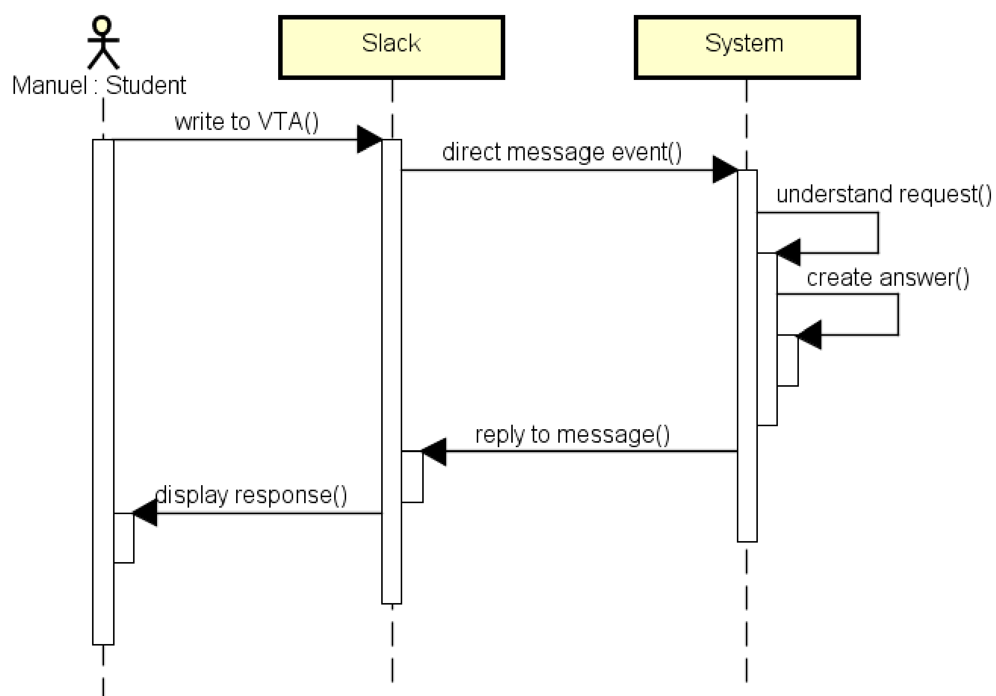

Figure 4.22: High-level view of interactions

When a user sends a direct message to the virtual assistant on Slack, the system receives it, processes it to understand its meaning and creates a proper answer that, when ready, is sent as a reply to the user in the same channel of the original message.

A second sequence diagram unveils some of the details of the system, in Figure 4.23. It shows that the understanding and part of the answer selection is done by the Watson™ Assistant service.

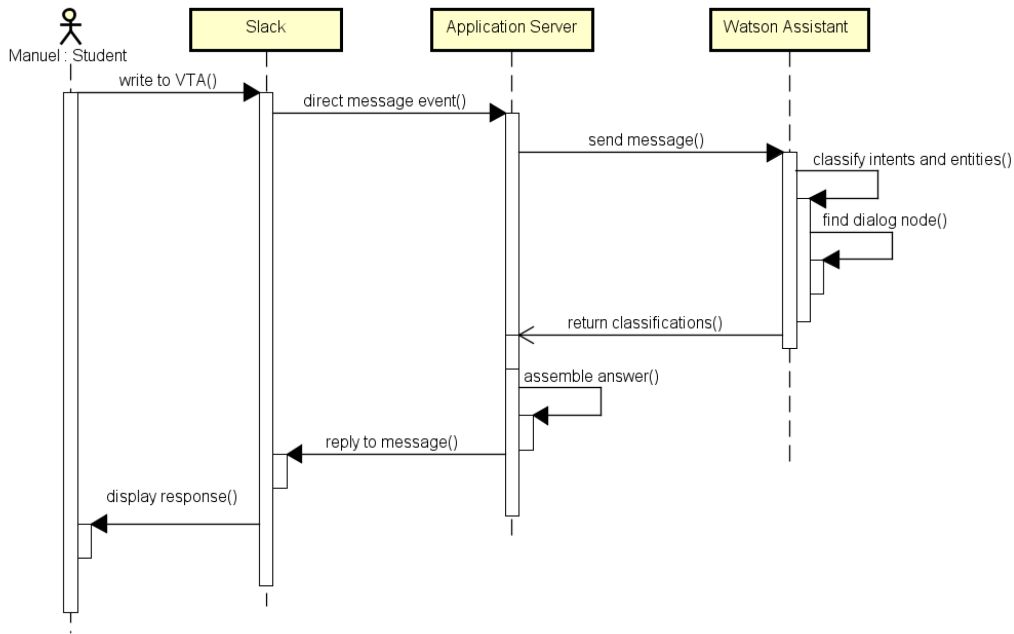

Figure 4.23: High-level view with Assistant

The third sequence diagram, in Figure 4.24, unveils what happens when data stored in a knowledge base is needed in order to complete an answer. The application server sends a particular query to the database server and when the results become available, it finalizes the answer.

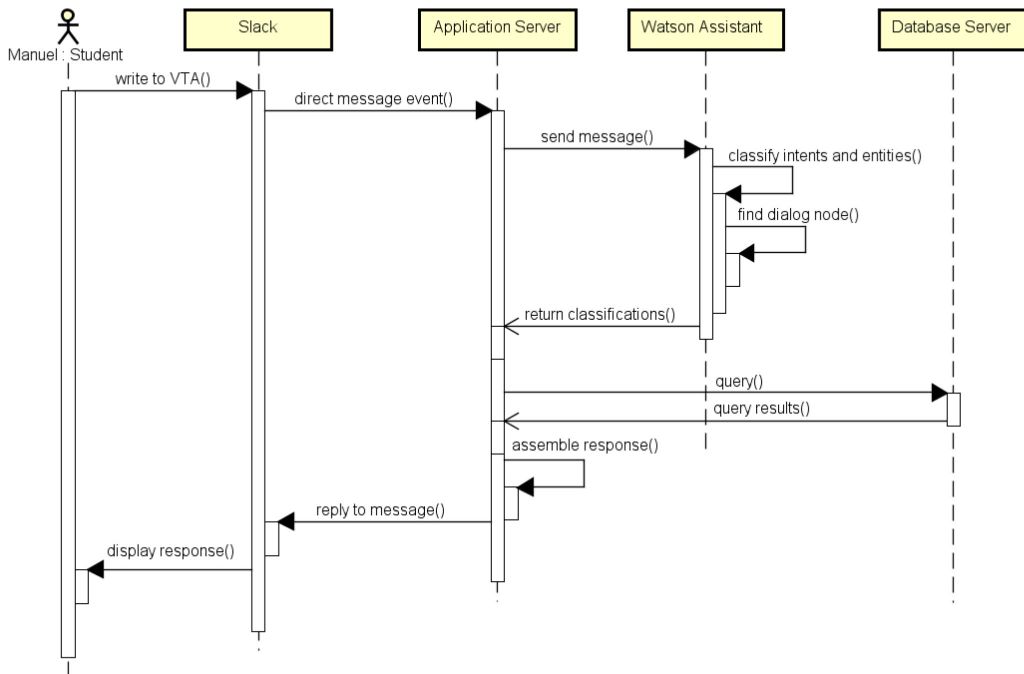

Figure 4.24: High-level view with Assistant and databases

4.5.2 Initialization

The starting phase of the application server is characterized by:

- connection to the different databases, performed by Mongoose for MongoDB and the MySQL driver.
- authentication to Slack as a bot user with the start of an RTM connection leveraging Botkit to call the corresponding Slack APIs.
- training of the spelling corrector, with the words contained in the documents of the MOOC.

4.5.3 Text Messages

Figure 4.25 shows what happens when a direct text message is received by the application server.

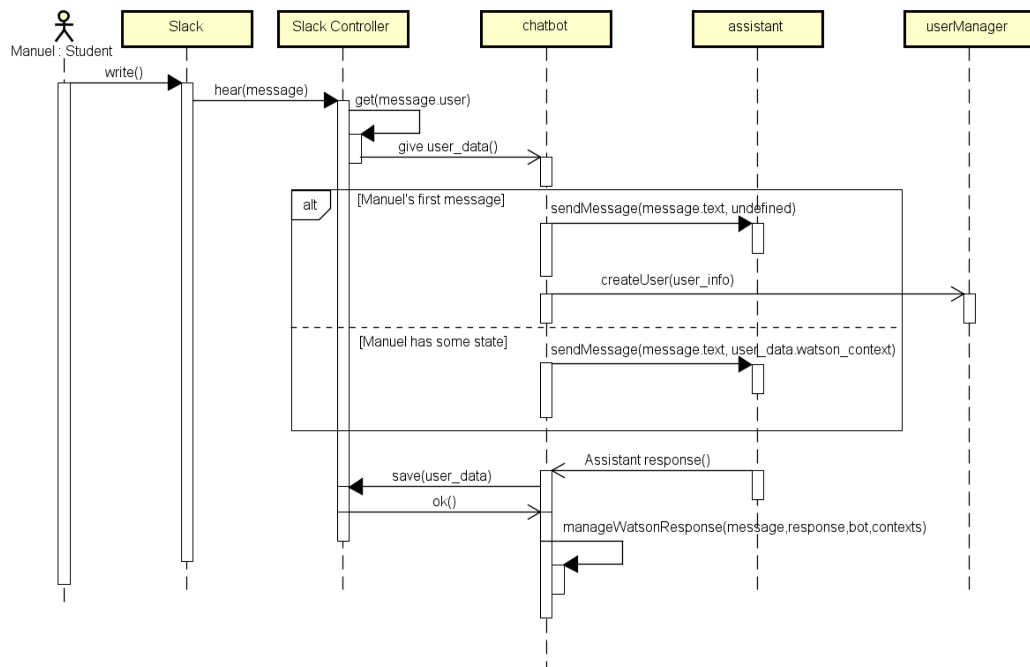

Figure 4.25: Text message sequence diagram number 1

The Slack controller retrieves the state of the user, if it is the first time the user sends a message, he is saved in the user database with the execution of the `createUser` function from *userManager*. Information about the user, such as Slack ID and Slack channel are retrieved from `message` while display and real names are retrieved through Slack's APIs. The state of the user contains his Slack ID

and Slack channel, the context information (`watson_context`) needed by Assistant to continue conversations and the array of Context objects created during the user's interaction with the VTA. This state is updated after the reception of the response object from Assistant, which is used to update the array of contexts and the `watson_context`. *Chatbot* sends the text of the message to Assistant with the `sendMessage` function contained in the assistant module and when the response from the service arrives, it calls the `manageWatsonResponse` function. *Chatbot* can call *contextManager* or *lookupManager* in order to reply to the user, it can call functions from *questionManager* or *entityManager* to fill the text of an interactive message, or even send a query to Discovery to respond with an extract from chapters of the MOOC. All these cases are exemplified in the following sequence diagrams, which pick up from the end of the sequence diagram in Figure 4.25.

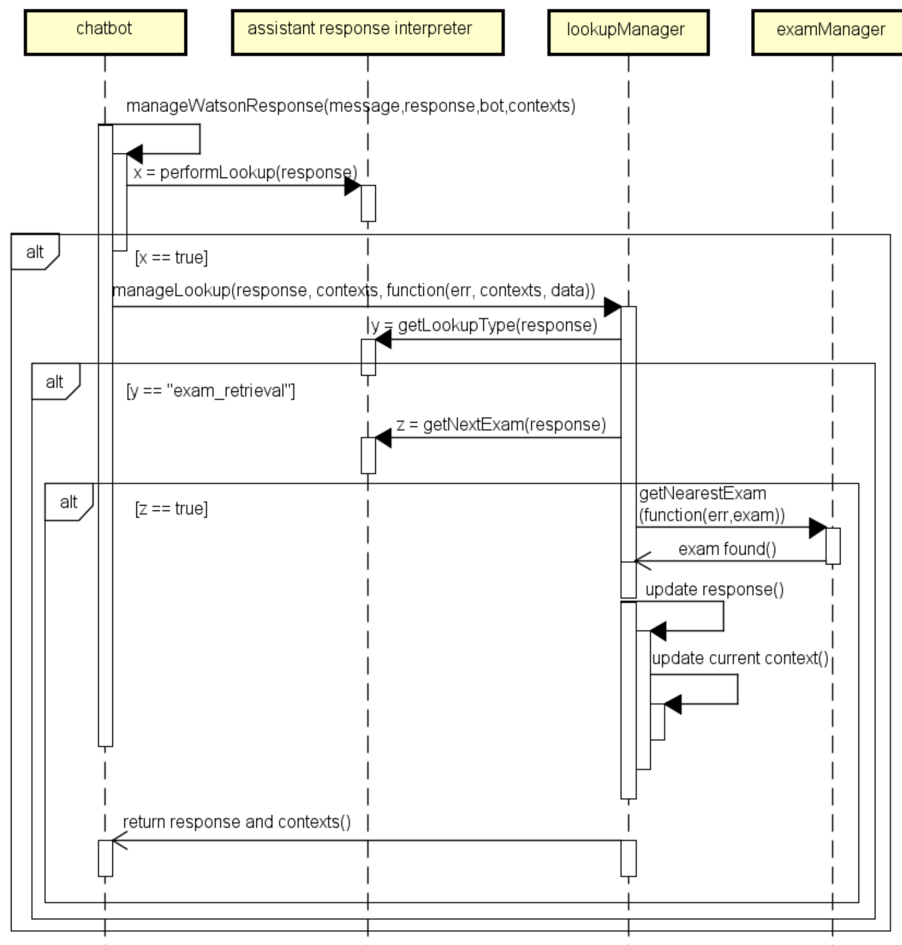

Figure 4.26: Text message sequence diagram number 2

Figure 4.26 shows what happens when Assistant requests a database lookup.

The assistant response interpreter module is used to identify the request and understand what information to look for in the database. *LookupManager* handles the call to *examManager* and the creation of the answer, with predefined sentences which need the insertion of the information related to the retrieved exam. It also updates the current context of the user's conversation, appending the exam object to it. The created answer is appended to the output contained in response, and when the lookup is concluded, the callback function is invoked with the updated response and contexts objects. The result of this type of interaction is illustrated in Figure 4.27.

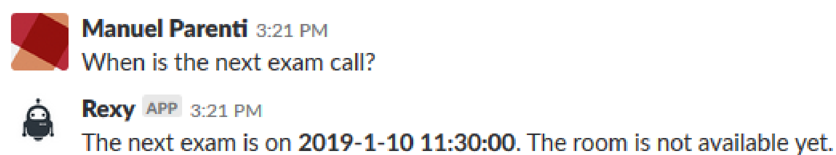

Figure 4.27: Interaction with lookup

Figure 4.28 shows a sequence diagram of the interactions between the main modules when Assistant requests contextual information to understand the meaning of the input message. The *assistant response interpreter* module is used to identify the request and the classified intent. *ContextManager* looks inside the array of contexts to find the current context and check if it makes sense with the recognized intent, as explained in Subsection 4.4.2.8. In the example, the user was talking about exams and he asked a temporal question. If the found context contains an exam it means that during the conversation, the user already mentioned that specific event, thus he might be looking for temporal information about it. If no mention of a specific exam is found, the decision that *contextManager* takes is to retrieve the nearest future exam. It does so by modifying the response object and sending it to *lookupManager*, which will take care of the database lookup and the creation of the answer. Once *lookupManager* has finished, *contextManager* can invoke its callback function with the updated response and contexts objects. The result of this type of interaction is illustrated in Figure 4.29.

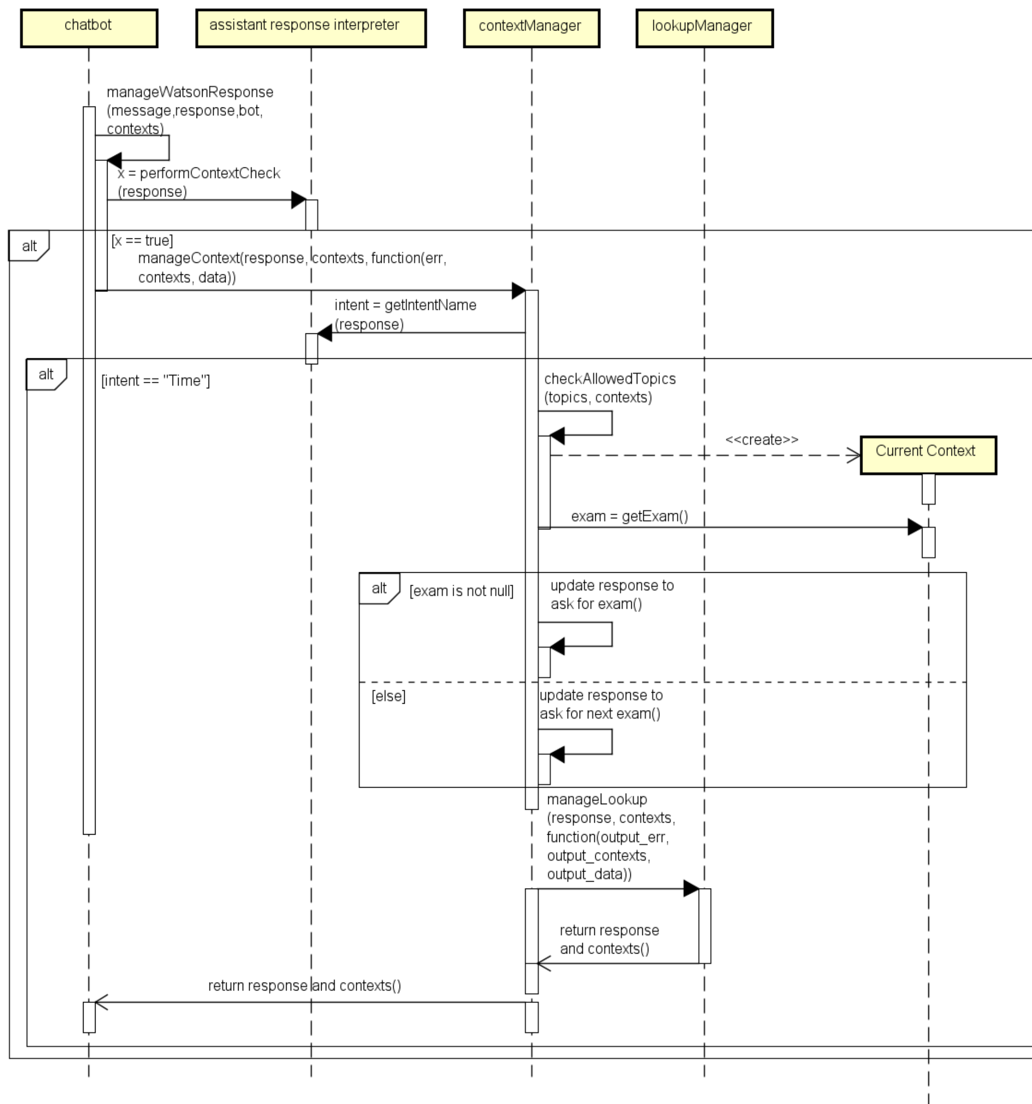

Figure 4.28: Text message sequence diagram number 3

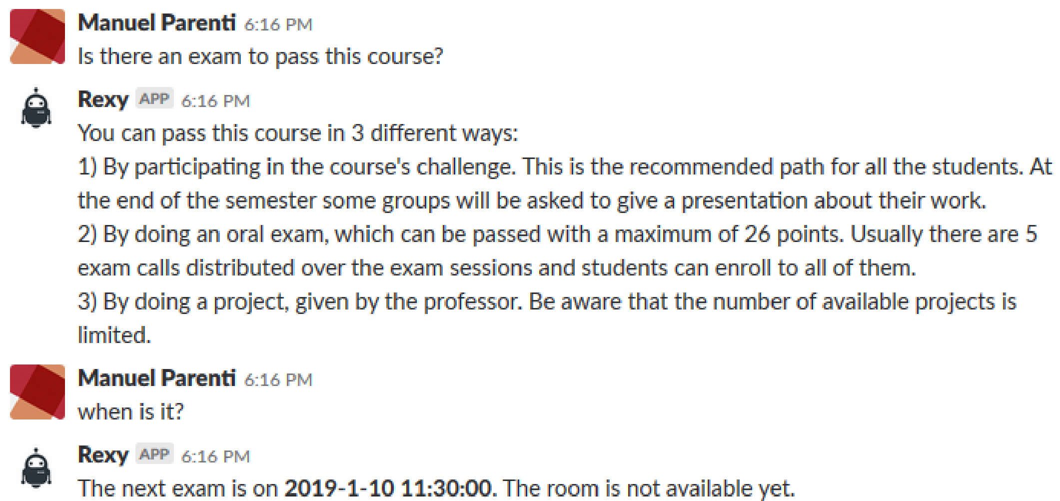

Figure 4.29: Interaction with contextual information

Figure 4.30 shows a sequence diagram of the scenario in which the list of concepts of the course has to be sent to a user.

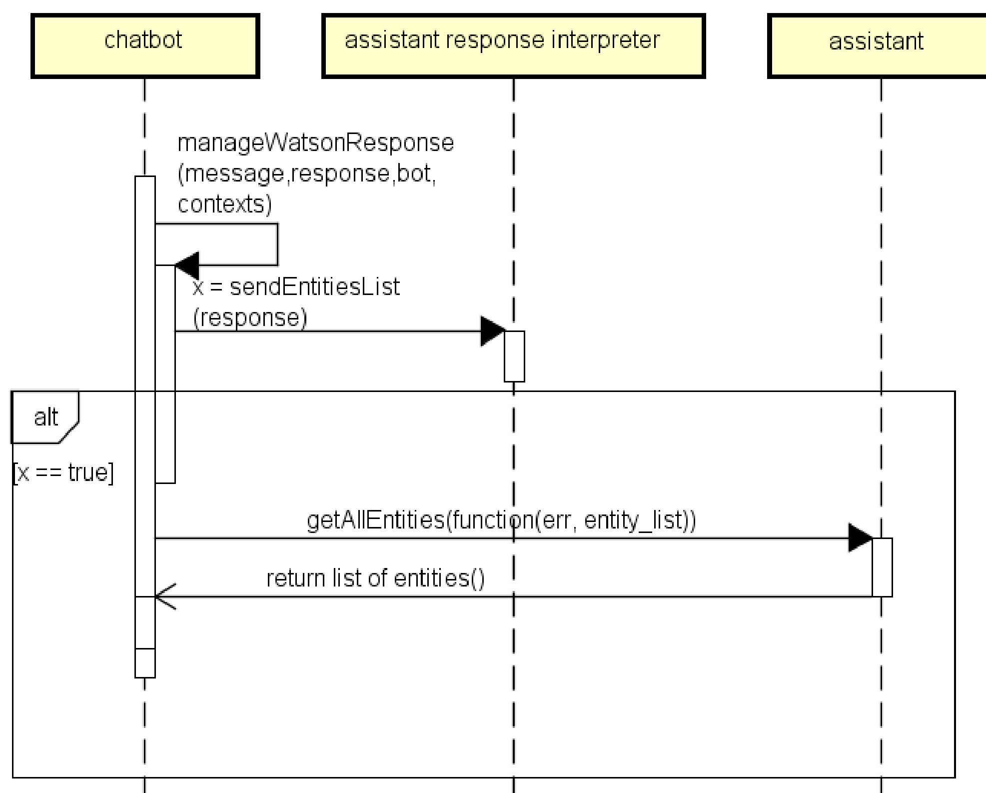

Figure 4.30: Text message sequence diagram number 4

Chatbot requests the list of entities to assistant and when they are returned, it creates an interactive message with list elements.

The result of this type of interaction is illustrated in Figure 4.31.

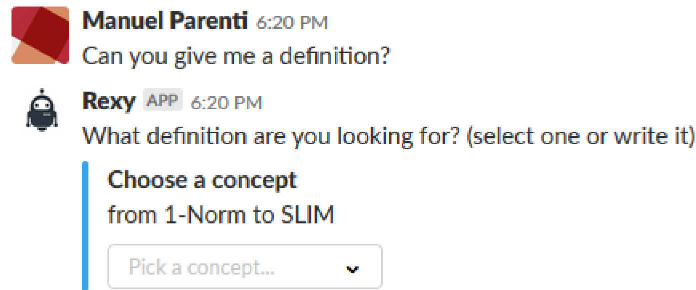

Figure 4.31: Display of the list of concepts

Figure 4.32 shows a sequence diagram of the scenario in which a user wants to practice or to test his understanding of the course.

Chatbot requests a new question for the user to *questionManager*, which looks into the question database to find a question that the user has not previously seen. This module makes a *Question* object available to *chatbot*, to create the interactive message for the user, with the text of the questions and the list of answers.

An example of question sent to a user is illustrated in Figure 4.33.

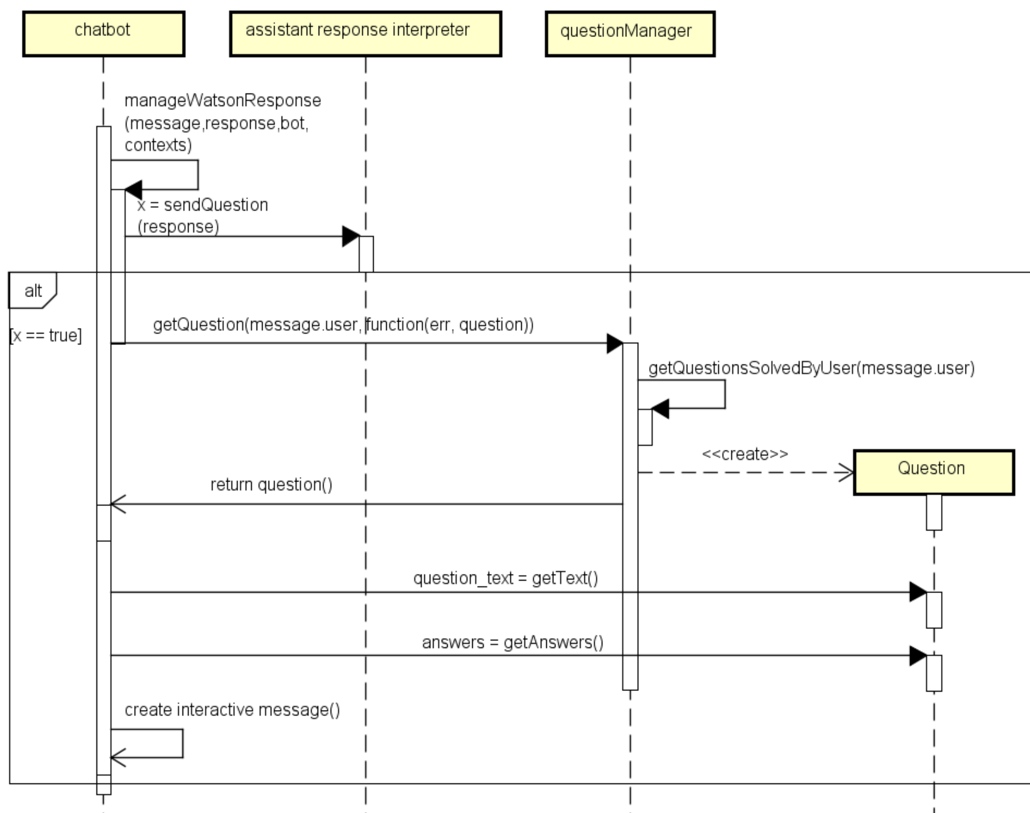

Figure 4.32: Text message sequence diagram number 5

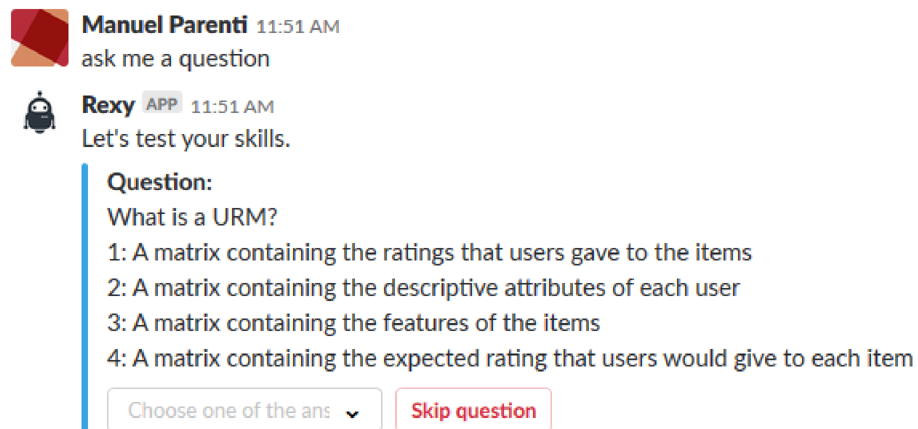

Figure 4.33: Display of a question

Figure 4.34 shows a sequence diagram of the scenario in which the VTA relies on Discovery to respond to a user.

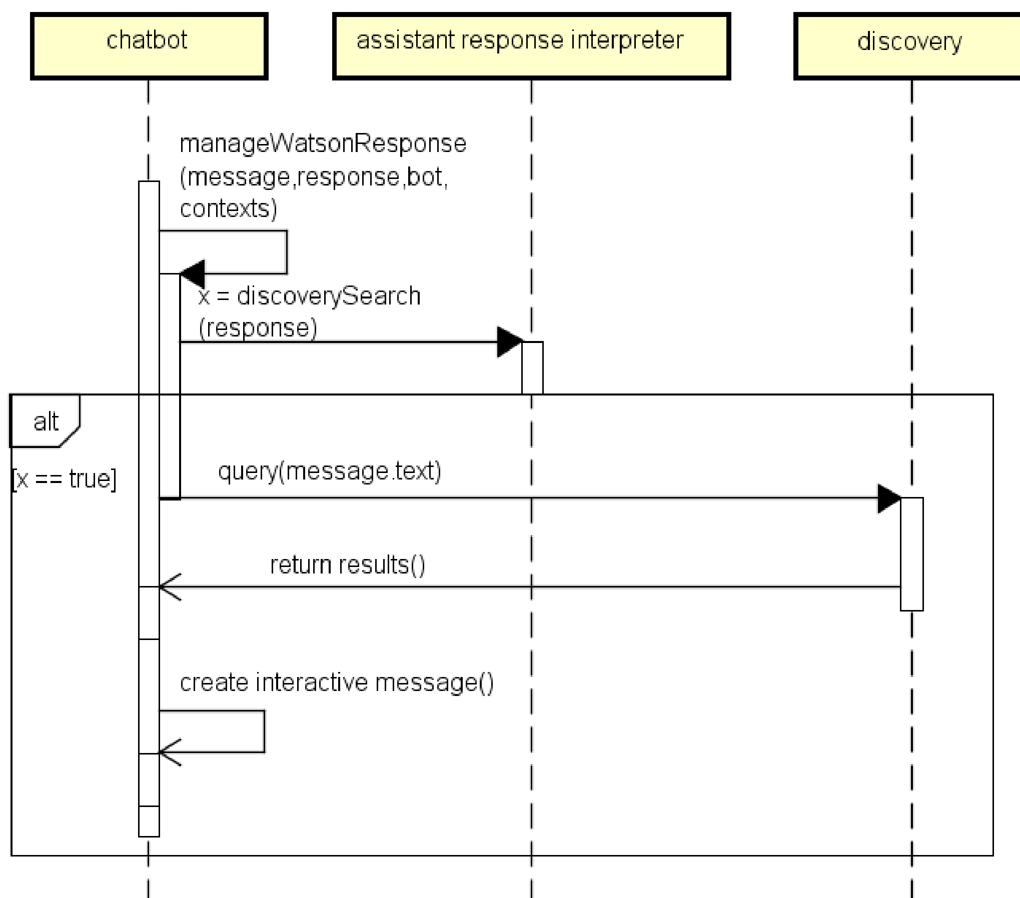

Figure 4.34: Text message sequence diagram number 6

The *chatbot* module detects that it needs to query Discovery through the as-

sistant response interpreter, it sends the text of the input message as a natural language query, and when the results are available, it creates an interactive message to answer the user. An example of interactive message containing the result of a Discovery query is illustrated in Figure 4.35.

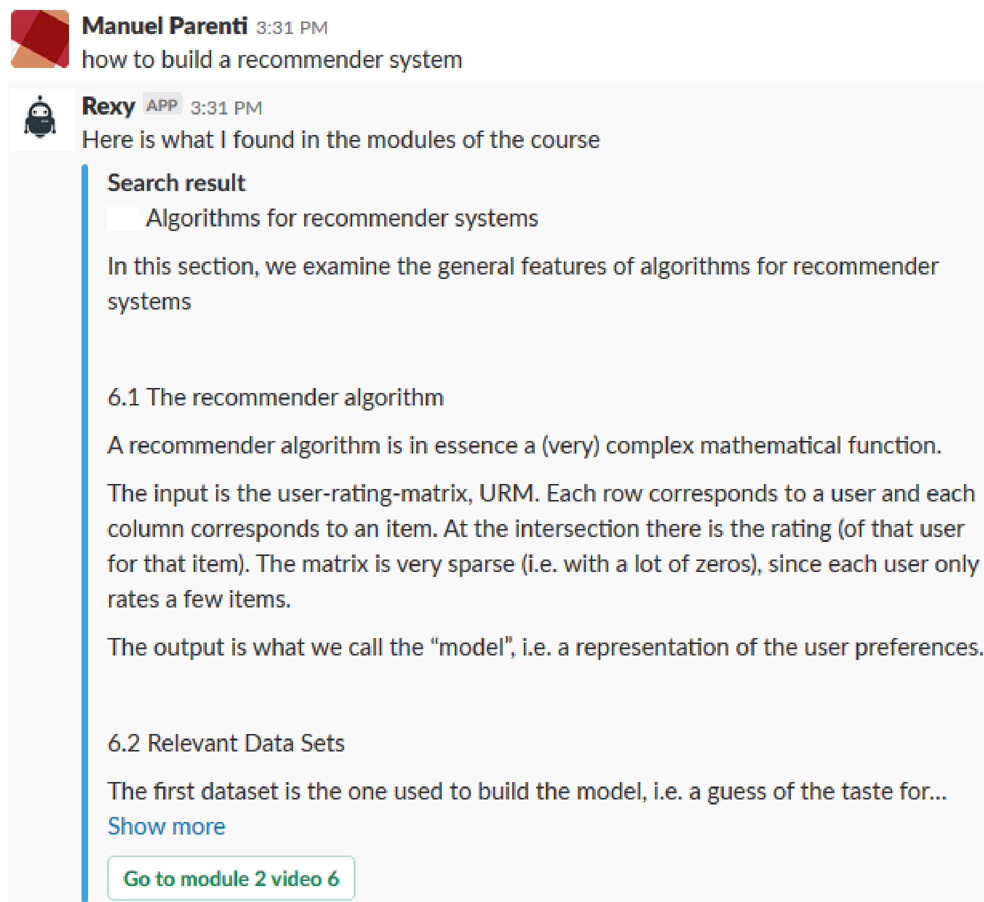

Figure 4.35: Discovery query result

Figure 4.36 shows a sequence diagram of the scenario in which the response found by Assistant is complete and needs no additional information from the server. It also shows the final passages of the `manageWatsonResponse` function, which apply to the previous sequence diagrams as well.

When the prepared answer is ready to be sent to the user, his state is saved, because the contextual information and `watson_context` might have been updated. The turn of the conversation is saved in the conversation history database and the answer is finally received by the user.

Chatbot performs other two checks: if the confidence of the intent recognized by Assistant is below a threshold and the conversation is in a state in which the recognition of the intent is important to determine the node of the dialog tree to

execute, a confirmation question is sent to the user. Similarly, a question containing the input message can be sent to the teachers of the course, in order to handle cases in which Assistant might not be able to understand the meaning correctly. These two types of interactions are saved in the conversation history database.
In Figure 4.37 there is a confirmation question asked to a student, while in Figure 4.38 the related teacher question is shown.

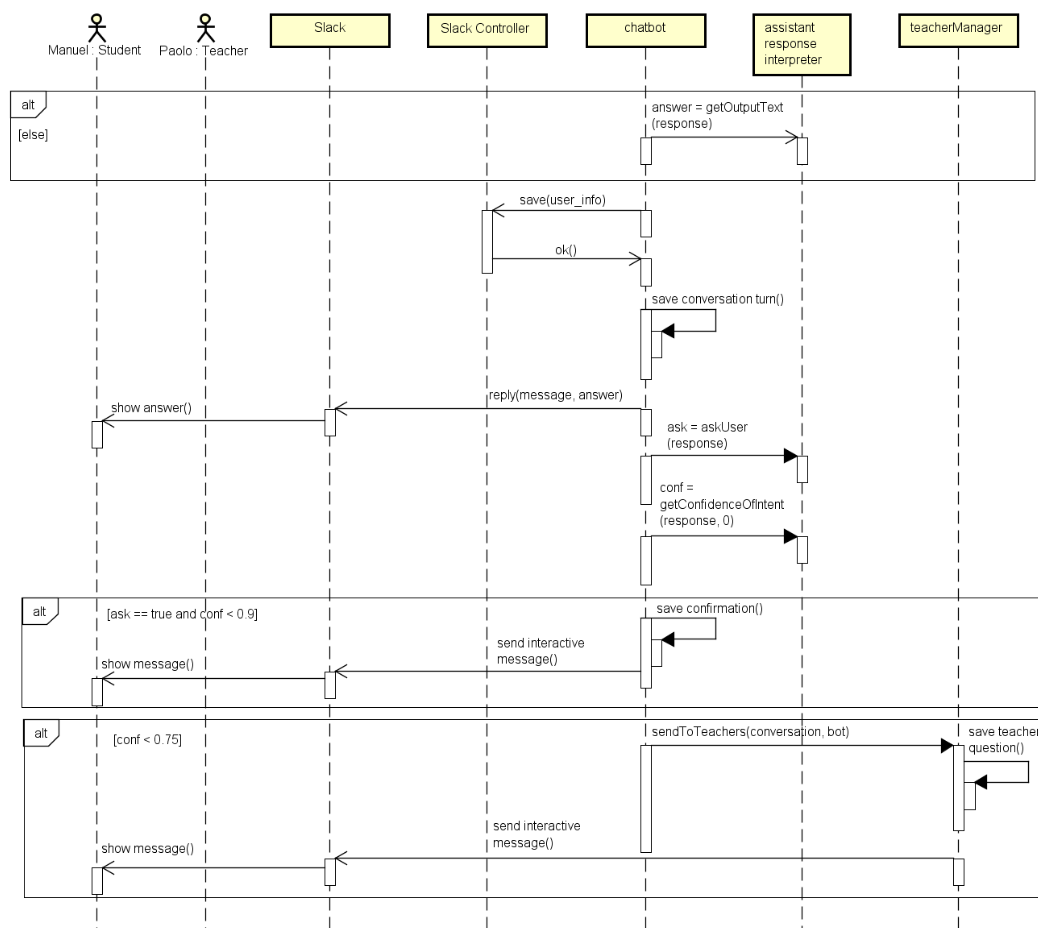

Figure 4.36: Text message sequence diagram number 7

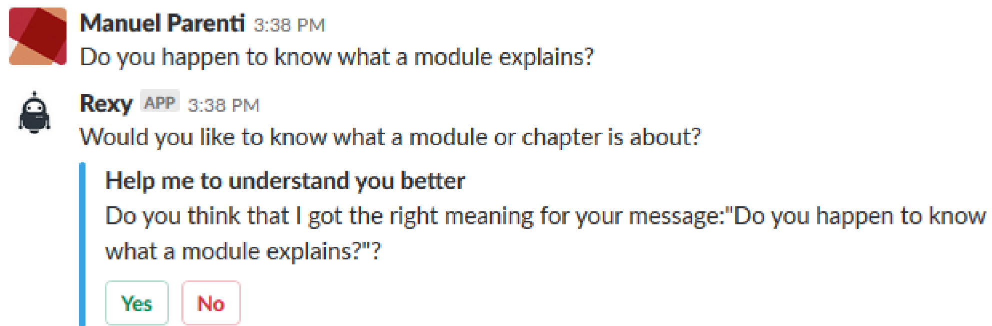

Figure 4.37: Confirmation question

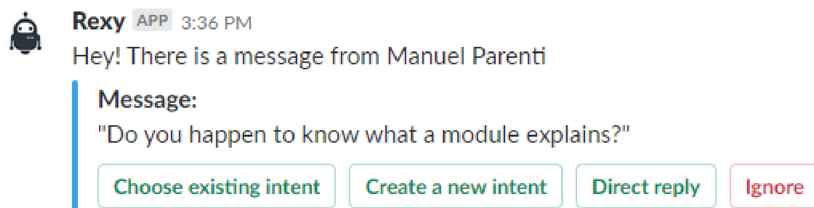

Figure 4.38: Teacher question

4.5.4 Interactive Messages

The sequence diagram in Figure 4.39 shows the interactions between modules and objects of the application server when a user interacts with a confirmation question.
Conversation, *User Confirmation* and *Teacher Question* are mongoose models used to communicate with the corresponding collections stored in the conversation history database. After the event corresponding to the decision of the user, the confirmation stored in the database is updated. If the user finds the interpretation of Assistant correct, thus clicking on the "yes" button, this interaction is used to train the intent classifier of Assistant, but only if no teacher already classified that message. Therefore, the decisions of teachers are given a higher value of trust. If the user thinks that the interpretation was incorrect, the teachers receive a teacher question related to it, in the case they did not receive it already.
The updated version of an interactive message containing a confirmation question, after the user clicked on "yes" is shown in Figure 4.40.

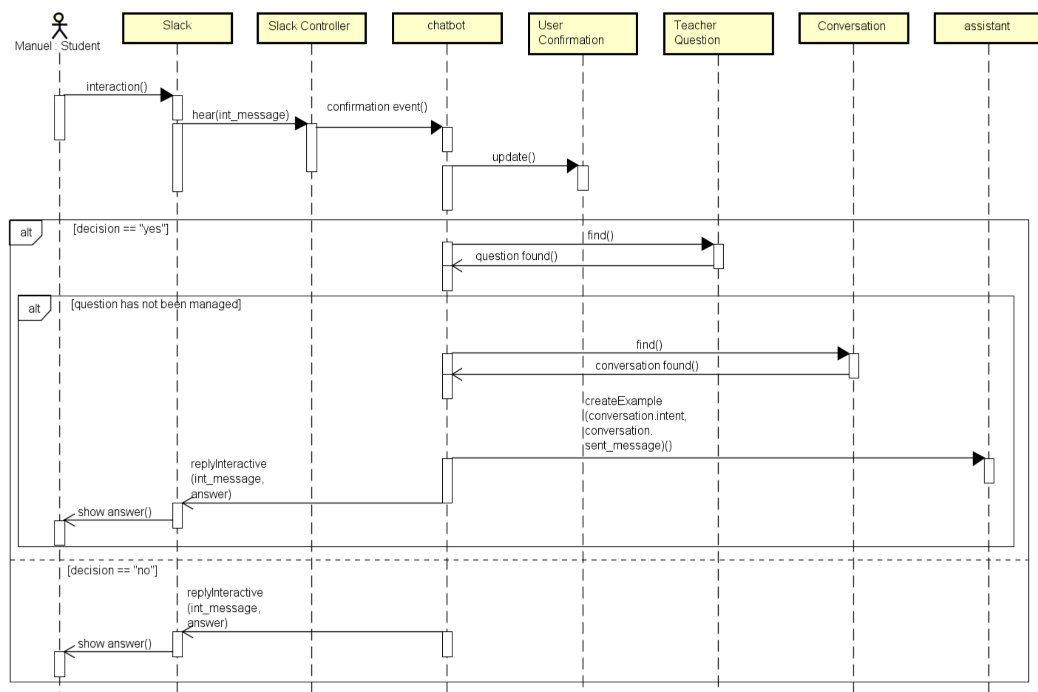

Figure 4.39: Interactive message sequence diagram number 1

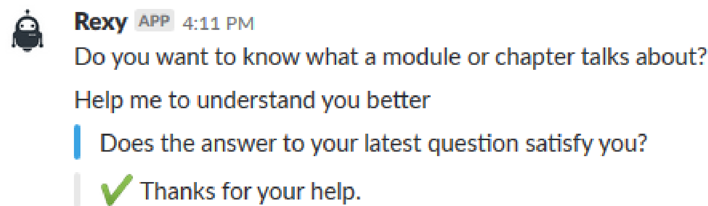

Figure 4.40: Result after a user confirmation

In the second sequence diagram, in Figure 4.41, the user selects a concept of the course from a list element.
A message containing its choice is sent to Assistant, following the same passages of a normal text message. When the final answer is ready, it updates the interactive message, displaying the response from the server, as shown by the example in Figure 4.42.

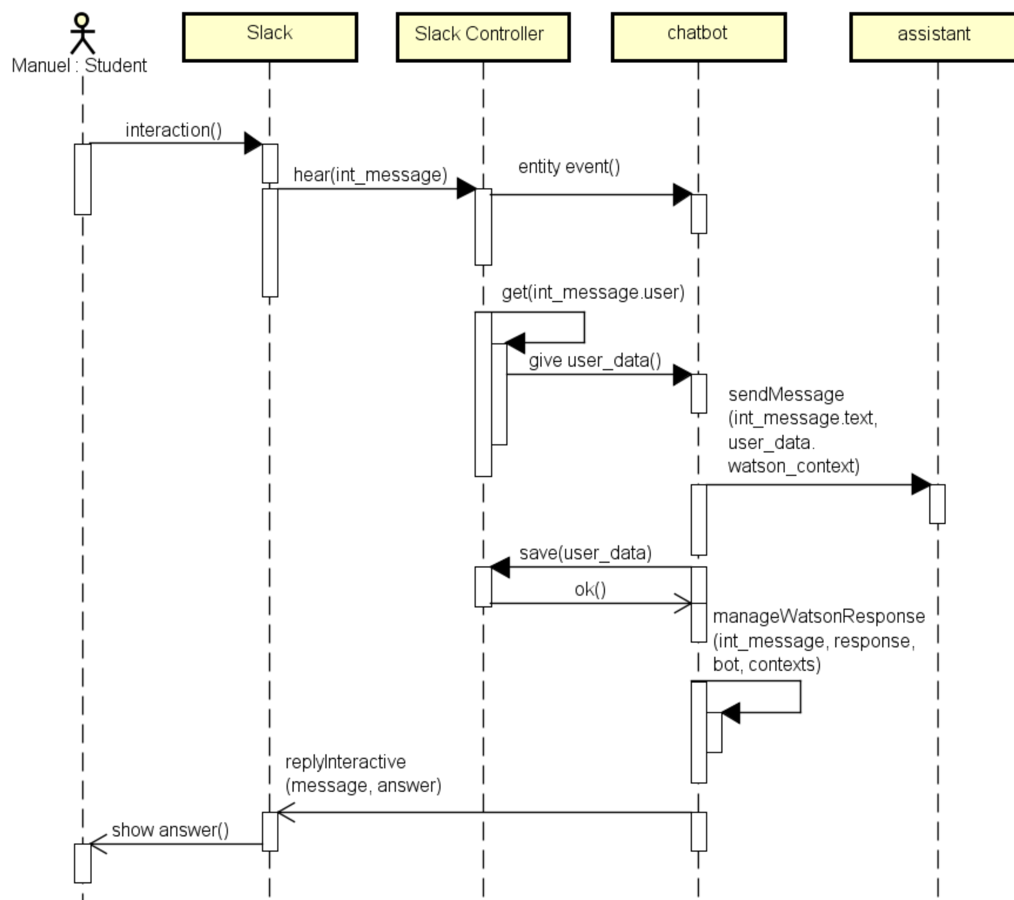

Figure 4.41: Interactive message sequence diagram number 2

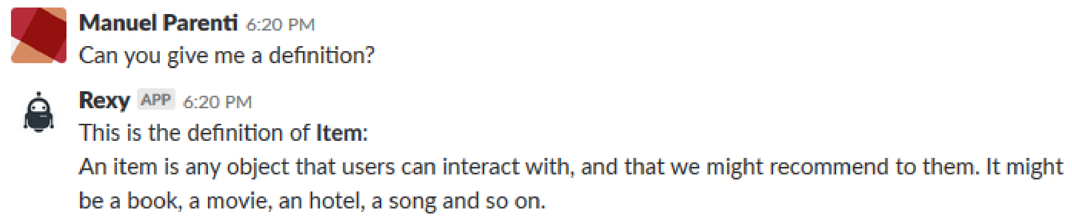

Figure 4.42: Result after a entity selection

In the third sequence diagram, in Figure 4.43, a teacher selects one of the options of an interactive message related to a teacher question.

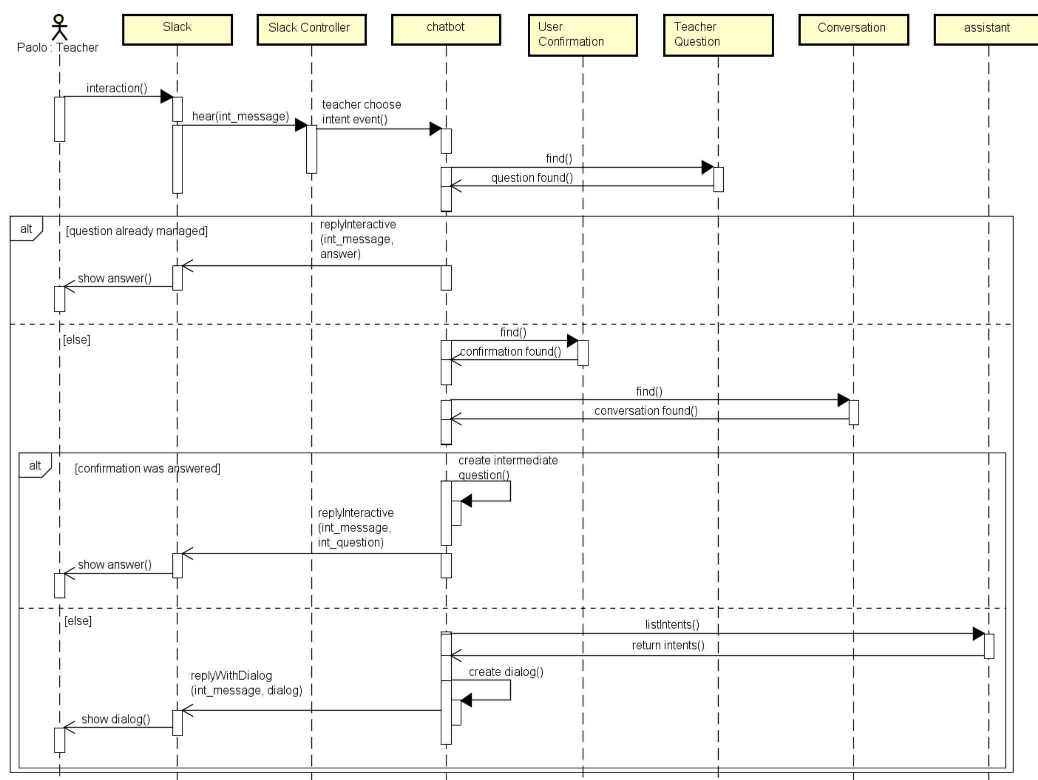

Figure 4.43: Interactive message sequence diagram number 3

The available choices produce different end results, but they follow the same passages. *Chatbot* retrieves the teacher question from the database and if it was handled by another teacher, the currently active teacher is notified. If a user confirmation is present for the message that has to be handled, and it has been resolved by a user, a special message is sent to the teacher. This contains the decision of the user and allows the teacher to approve or reject it. In the latter case and in the case that no filled confirmation questions are present in the database, *chatbot* creates a menu (Slack dialog) for the teacher to fill.

Continuing the example of Figure 4.40, if a teacher tries to manage a teacher question related to the user's message, he receives the interactive message in the left image of Figure 4.44. If he rejects the decision of the user, one of the Slack dialogs presented in Figures 4.46, 4.45 and 4.47 is displayed, depending on the teacher's choice.

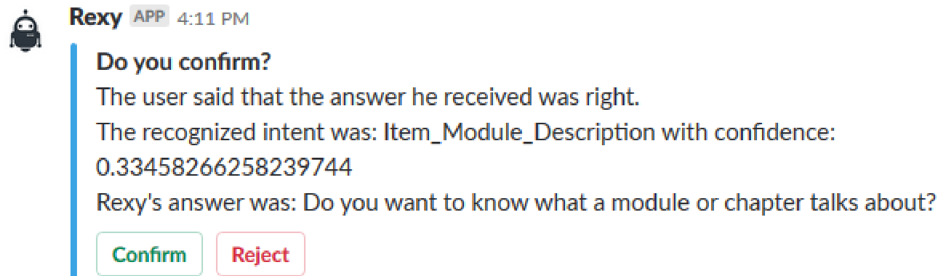

Figure 4.44: Managing a message that has been confirmed

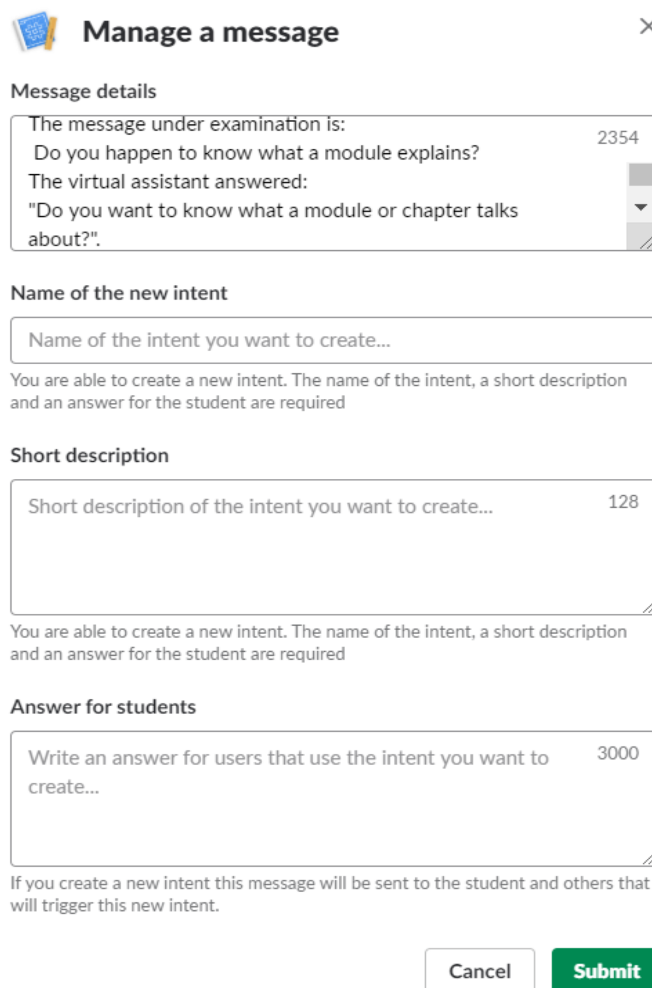

Figure 4.45: Dialog with intent creation

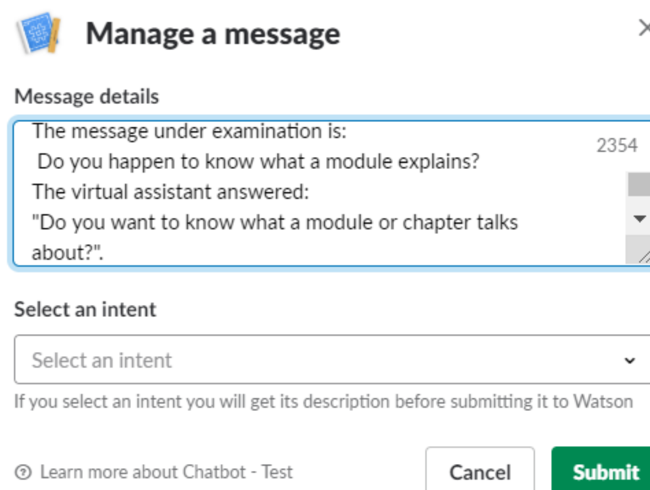

Figure 4.46: Dialog with intent selection

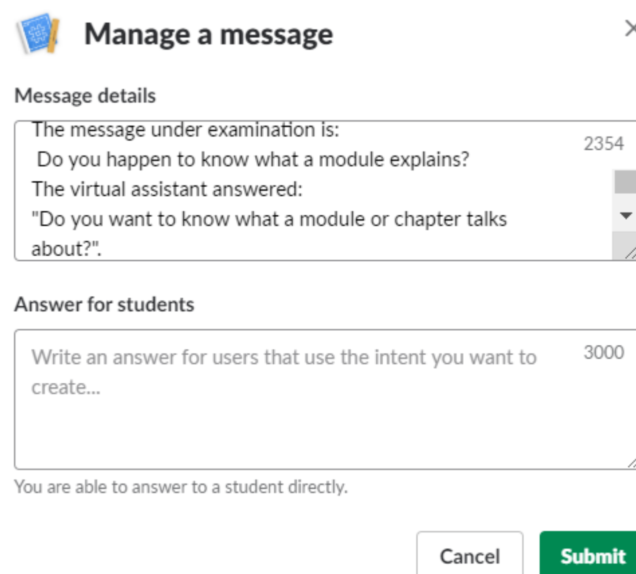

Figure 4.47: Dialog with direct reply

When a teacher submits a Slack dialog for the selection of an intent, the server retrieves information regarding that intent to ask the teacher if he is sure of its choice. The sequence diagram shown in Figure 4.48 illustrates the interactions of the different modules after a teacher has decided to accept or reject his selection. If the decision is accepted, an example is created for the selected intent, otherwise the teacher can make another choice.

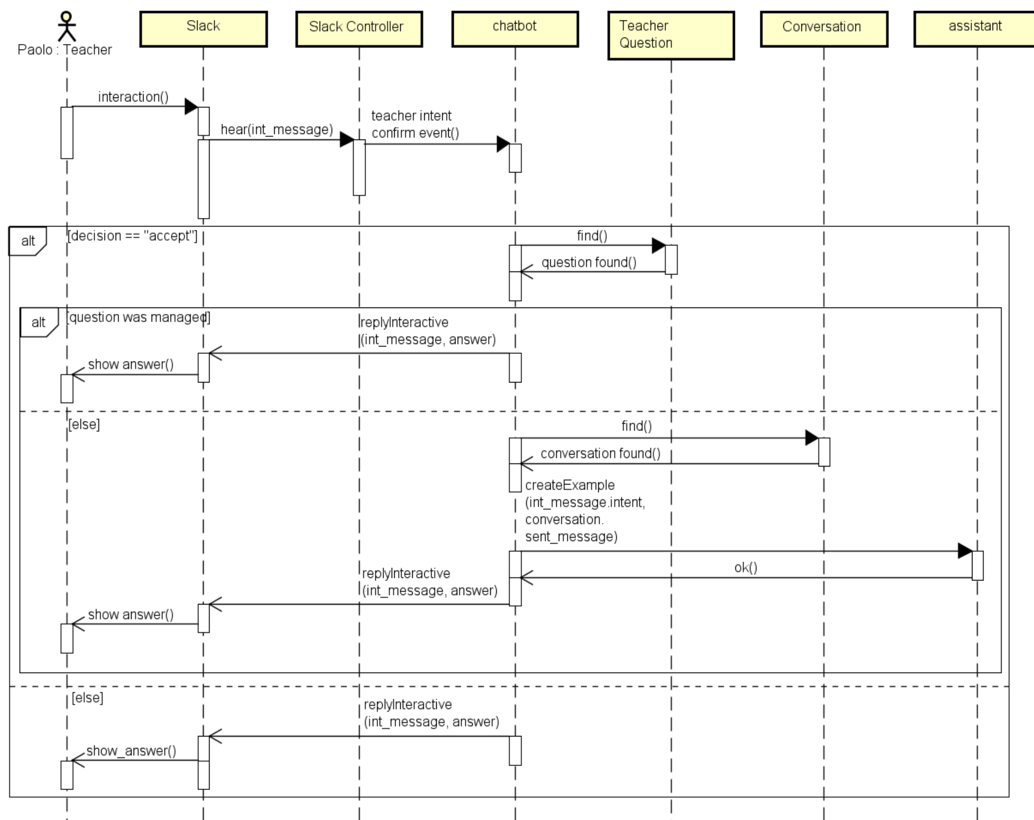

Figure 4.48: Interactive message sequence diagram number 4

The interactive message that a teacher receives to confirm his selection of the intent is shown in Figure 4.49.

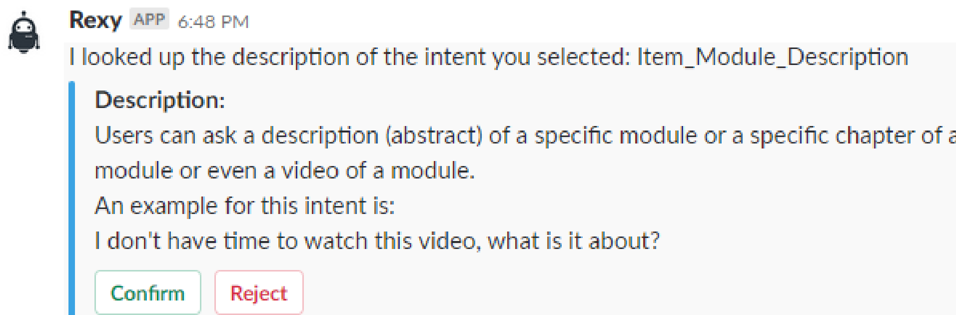

Figure 4.49: Interactive message with intent description

The sequence diagram shown in Figure 4.50 illustrates the interactions of the different modules after a teacher has decided to accept or reject a user's answer to a confirmation question. If the teacher rejects the user's input, the example that was created on Assistant is deleted and the teacher can continue with his task of handling the user message with a Slack dialog.

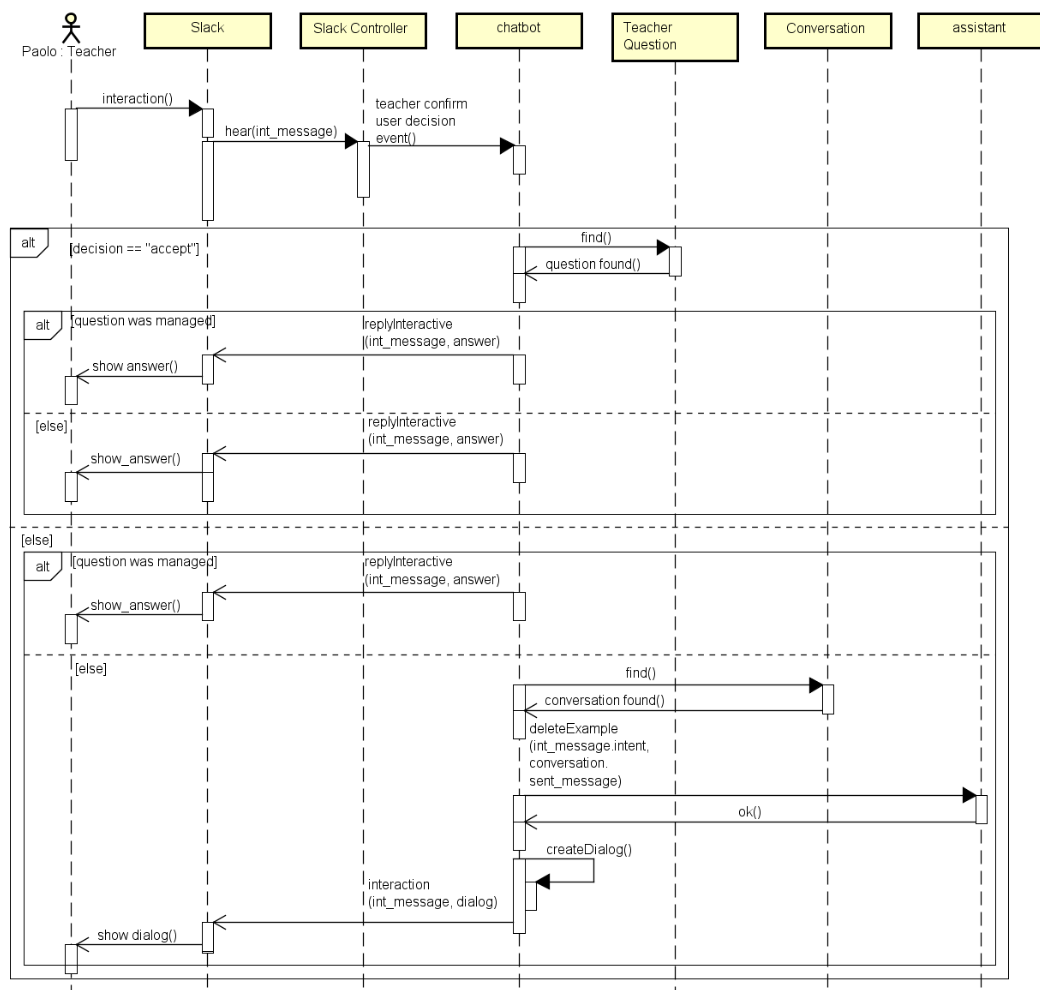

Figure 4.50: Interactive message sequence diagram number 5

The sequence diagram contained in Figure 4.51 shows the operations that the *chatbot* module performs when a user answers to a question regarding the course. It checks the correctness of the answer and if the user got the right one, he receives a positive message, otherwise the correct answer is displayed to him. If the user decides to skip a question, a new question is retrieved with the use of *question-Manager*. All those interactions are saved in the question database.
An example of interactive message after the correct answer has been given is in Figure 4.52.
After the results of a selected answer are shown to the user, he can decide to pass to a new question. The passages involved in the retrieval of such question are shown in Figure 4.53.

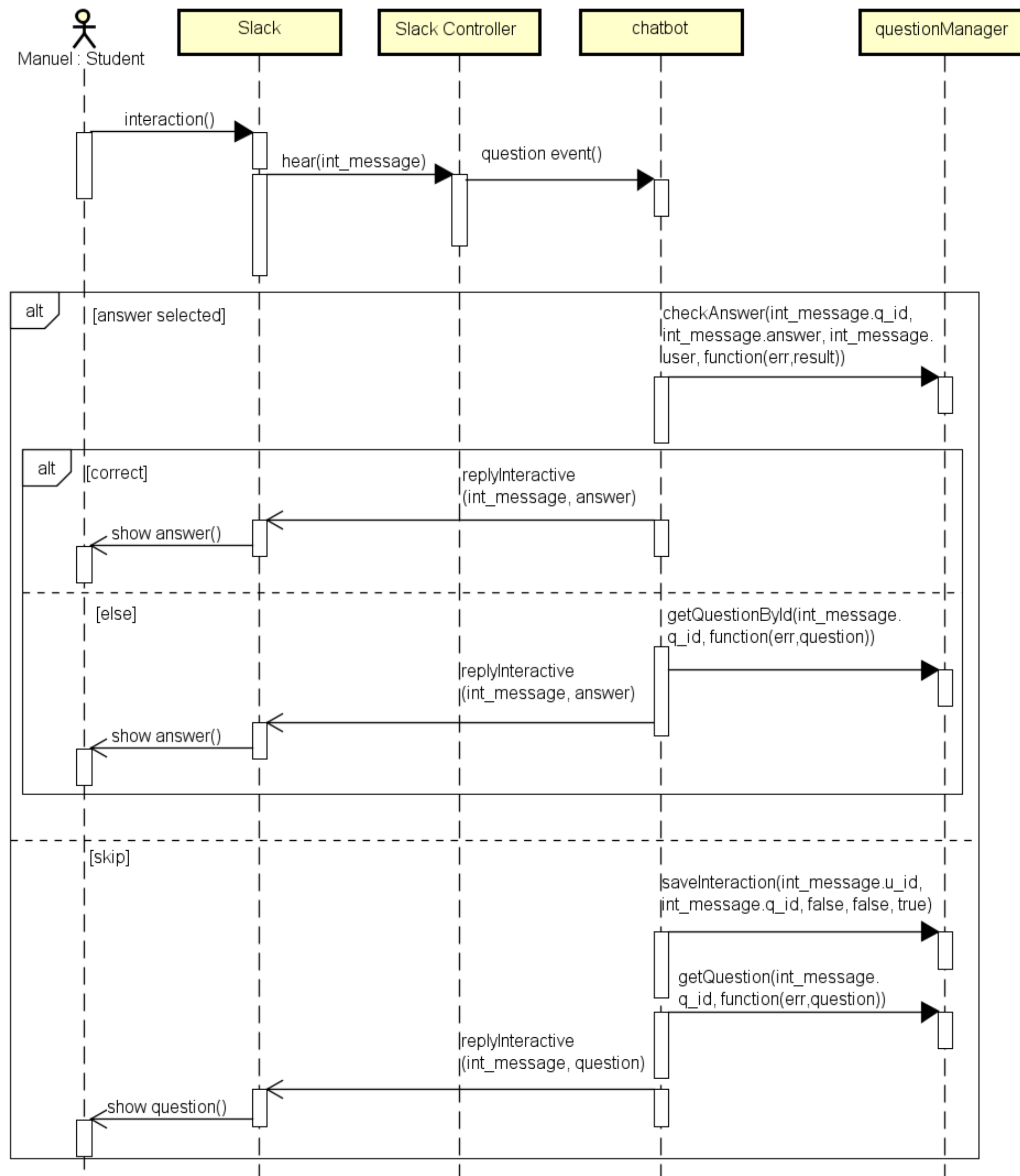

Figure 4.51: Interactive message sequence diagram number 6

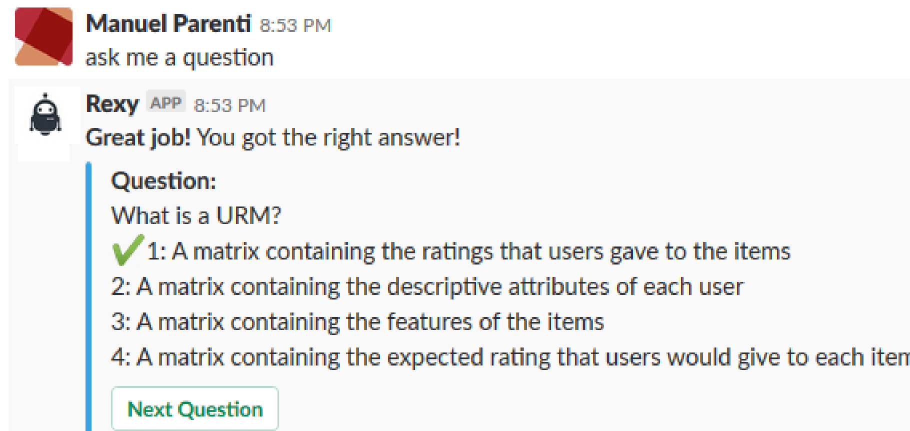

Figure 4.52: Interactive message with correct answer

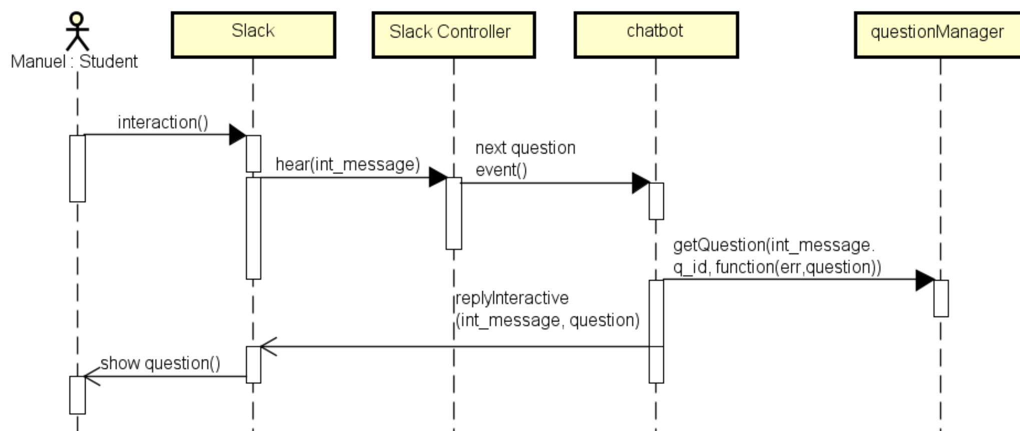

Figure 4.53: Interactive message sequence diagram number 7

4.5.5 Slack Dialogs

The sequence diagram shown in Figure 4.54 illustrates the passages that are followed by the server when a teacher submits a filled dialog.
Slack Controller validates the input and if some fields have to be modified by the teacher, it notifies him. When a correctly filled dialog submission is received by *chatbot*, it checks that the problem has not been solved by another teacher. In the case the dialog's goal was to select an existing intent, the message from the user is added to the training set of the selected intent. If the teacher wants to create a new intent, the assistant module saves it and creates a dialog node to execute when Assistant recognizes that intent. Then, the answer defined by the teacher is sent to the user who sent the original message. This also happens in the case

when the teacher only wants to send a direct reply to the user.
An example of a message received by a student after a dialog submission is shown in Figure 4.55.

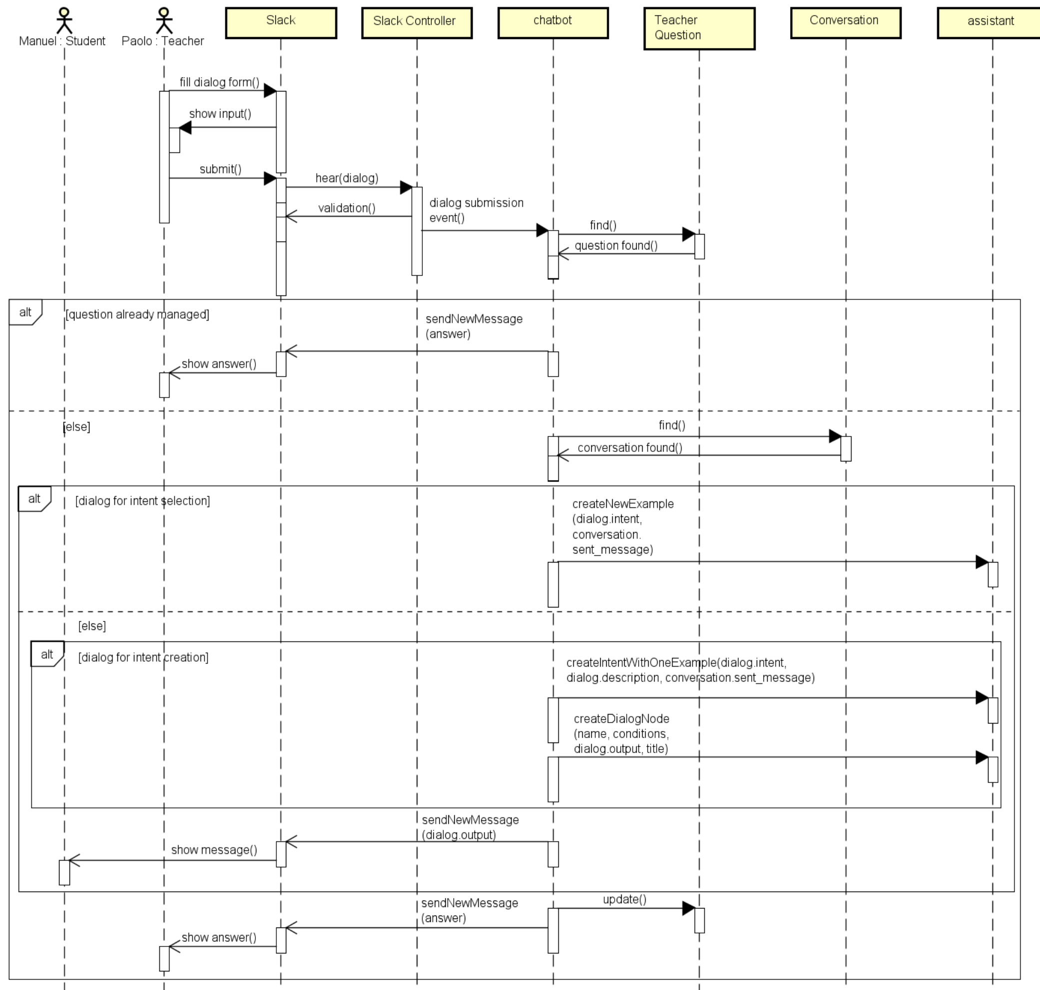

Figure 4.54: Dialog submission sequence diagram

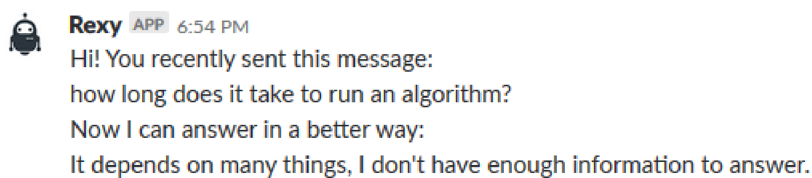

Figure 4.55: The message received by a student after a teacher has managed one of his messages.

Chapter 5

Results

This chapter presents the results of Rexy's activity in the *Recommender Systems* course, the tasks needed to update and adapt Rexy to other courses, the amount of concurrent requests it is able to handle, and a preliminary analysis of student clustering.

5.1 Interactions with Rexy

At the time of writing, Rexy has been deployed for 40 days, therefore the numbers presented in this section show a very early insight of how it worked.

20 students interacted with it, reaching a total of 239 conversation turns, thus having an average of 11,95 messages per student. The messages that required the classification of an intent were 158, the remaining ones contained the entities needed to continue some branches of the dialog tree. The manual inspection of its activity showed that Rexy detected the right intent 103 times out of 158 and the messages it was not able to understand properly are divided in 26 requests that should have been handled and 29 requests that were not relevant (random characters or texts that the assistant should not be able to understand). Entities were recognized correctly every time.

At the beginning Rexy was lacking some intents because they were not foreseen during the creation phase but were added later for future users. The 26 requests that Rexy should have understood were not unique, there were groups of 3 or 4 messages bringing the same intent with different formulations; this happened because students tried to make Rexy understand their question with multiple attempts even though it was not able to answer. Every group of such requests was mapped to a new intent in order to be able to respond to them in future occasions.

Moreover, the students that sent them received an answer from one of the teaching assistants.

In their interactions with Rexy, students showed that the challenge is their main interest since most of the time they wrote Rexy asking for information, rules and deadlines regarding it. The second most requested topic was the content of the course, mostly in definitions of concepts, and the third consisted in the lectures of the course.

The students used a conversational and idiomatic writing style most of the times, they wrote complete and articulate sentences, as if they were writing to a human assistant. Only three students wrote to Rexy command-like messages and very short sentences. The VTA was able to understand both types of requests, but this qualitative result shows that most of the students expect digital assistants to be able to interpret longer sentences and might prefer to use natural language instead of learning new commands.

The intents that were touched by the requests of the students were only a fraction of all the intents that had been defined, leaving a good part of the dialog tree unexplored. This was expected, because Rexy was deployed a month after the start of the course. This limited the impact that it could have had, since the organization of the course is a topic that is usually discussed in the first days. Also, trying to change the habits of the students to push them to interact with a VTA is not trivial and needs further investigations in order to understand if such VTA could be successful in different areas.

What Rexy was really good at was replying to questions related to the syllabus of *Recommender Systems* and navigating through the content of the course, while it struggled on questions regarding very specific aspects of the challenge. For instance, some of the messages it received were about computational time of algorithms and how to translate a concept into code; such requests were not expected because the information needed to answer cannot be found inside the material of the online course. Additionally some of those questions cannot be answered easily, with just a short text or few turns of a conversation; for instance, to forecast the computational time required by some algorithm, one should know how it is written, the programming language, the size of the involved dataset, the hardware resources that are available.

5.2 Maintenance and Portability

Every semester, a course might undergo changes in its organization and its content. Therefore, the information stored in the knowledge base of the VTA must be updated, as well as some of its predefined responses. This task is necessary to maintain consistency between the real organization of the course and what the VTA can say to students. It is not very time consuming, since it consists of an update of part of the pre-existing data, namely the administrative database and the response defined for the dialog nodes in the Assistant workspace. In detail, on the database side, the update involves the lectures, exams and deadlines collections, and on the dialog tree side it involves the responses containing information subject to change, such as the rules of the competition or the topics covered by the course.

Instructing the VTA for a new course requires more effort: its knowledge base (lectures, exams and questions databases) shall be filled and some of the static responses, defined for specific requests, must be modified in order to fit the new scenario. However, tagging and creating abstracts of the different modules of the course to fill the corresponding database is definitely the most time consuming task. The tagging activity consist in recognizing the concepts of the course in the transcripts of the video lessons and giving each occurrence a role (e.g. definition or introduction of a concept). The effort required depends on the dimension of the content of the MOOC; tagging the material of an already existing course implies that the annotator must go through it completely, possibly even multiple times in order to understand the content and its structure. In the case a new course is being prepared, this task might be easier and less time consuming since the different chapters can be tagged during their creation. Moreover, the writer is an expert of the course, thus the tags and abstracts he produces should be reliable.

The already existing intents used by Assistant to understand the meaning of messages are quite general and their input samples are not course-specific, thus they do not require any type of adaptation. Entities and their values, instead, have to be changed since they are related to a single course. The rules defined for the context management should also be revised, since for *Recommender Systems* the challenge and its deadlines were taken into account, but those can be reused in courses that have some side projects or assignments. Furthermore, because courses are not equal to each other, new intents and branches of the dialog tree shall be created for the requests that students of a particular course can make. This is necessary since two courses are not equal and teachers know what types of questions their students usually ask.

In the case of an online course, only @*course_entity*, @*reference-role*, @*module*, @*chapter*, @*video* entities have to be changed, while most of the intents related to the organization of the course can be substituted. Thus, the only databases to be filled are the ones containing information about the video lessons and the multi-choice questions for students.

5.3 Scalability

Before Rexy's deployment, load tests were run to ensure that the application was able to answer to students in an adequate amount of time. The results showed that the VTA's performance was more than acceptable even in situations that far exceeded the expected traffic of messages coming from students.

5.3.1 Testing Scenario

The tests were run on a dummy application, which emulated all the functionalities of the real application, except for the interactive messages, and the reception/dispatch of messages from and to Slack, because of the limits in the rate of messages that can be sent by bot users [25] and because the emulation of students writing many messages at the same time was not possible. In order to mimic student interactions, POST requests were sent to the application, with the text of the message and information about the sender inside the body of the requests. When the application was ready to answer, it responded to the corresponding POST request. The tests were run on the same virtual machine used for the deployment of the real application server, presented in Subsection 4.1.5.

5.3.2 Test Description

The tests were carried out with Artillery, an open source load testing toolkit [2]. It enables testing of complex user behavior with the definition of scenarios, which specify the type and order of actions performed by the virtual users. It also allows to test different loads in dedicated phases, which consist of an arrival rate, a duration and, optionally, a maximum number of requests per second that can be sent. The tests consisted of sending messages to the application server in different phases and with different loads, measuring the time elapsed until the reception of the response. For every message sent, a random user was picked from 50 fake users, to simulate the management of contextual information. The text of each

message was taken randomly from the training samples of all the intents stored in Assistant. After the reception of all the responses, Artillery calculates statistics about the performance of the system, such as the minimum (min), maximum (max), median, 95th (p95) and 99th (p99) percentile values of the request latency.

5.3.3 Results

In a first test, 300 requests were sent in a 60 seconds long window (e.g. 5 requests/second or RPS). In a second test, after a warm up phase of 30 seconds in which the RPS went from 5 to 50 in a linear fashion, a second phase started with a constant load of 50 RPS for other 30 seconds. The time required to respond increased, as shown in Figure 5.1. When ramping up the RPS to 100 and to 200 in such scenario, the resulting response times were longer and not acceptable.

During the activity of Rexy as a VTA for *Recommender Systems*, students wrote in different times and in a very distributed fashion, so bursts of messages like the latter tests were not representative of the real environment. In a scenario in which the end users are numerous and are expected to interact very frequently with the VTA, it is advisable to allocate more resources to the application server or to distribute the load on more threads.

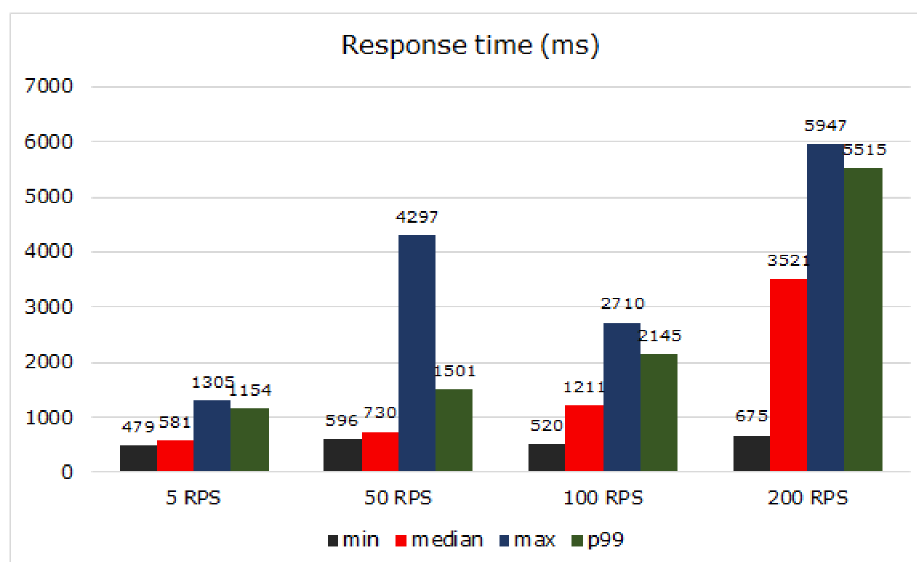

Figure 5.1: Load test results

5.4 Clustering of Users

Since a VTA could be able to personalize the experience of its users, one of the first steps that can be taken in that direction is to analyze their behavior to find profiles that represent groups of users. Once they have been discovered, the VTA could bring new functionalities targeted at them or at the whole community of users. In this section a preliminary study aimed at finding those groups is presented. It is based on the clustering algorithms introduced in Section 2.4, and it is run on simulated user interactions because of the lack of a consistent number of real user interactions.

5.4.1 Simulation

The simulated groups of users are as follows: 20 users interested to exams ("exam users"), 30 users interested to the challenge ("challenge users"), 20 users interested to lectures ("lecture users") and 30 users interested to the content of the course ("content users"). In a second step, other 40 users, interested in all the different topics in equal manner ("mixed users"), are added. For every user, a random number of interactions between 20 and 40 is generated taking text messages from the training samples of the intents, which are divided in sets for the different topics: exams, challenge, lectures, content, common phrases and general intents. Said sets can be partially overlapping in case some intent can be used in questions about more than one topic. To simulate a user interested in one of the available topics of conversation at the time of picking a message, the probability associated to samples of the intents contained in the corresponding group is 0.5, and the probability of picking a message from each of the other groups is 0.1. All the said interactions are saved in the *conversation history database* with their intent and recognized entities, and, as the *id* of the user, the group which he belongs to.

5.4.2 User Model

The users are modeled with the main information that is currently gathered by the VTA, which are the intents recognized in the messages they sent to the assistant. Thus, the user model is a vector containing the absolute frequency of messages that were classified as an intent, for all the available intents. This choice comes from the design of the intents and also from the objective of the partitioning activity, since most of the defined intents don't need additional information to represent the interests of a user. In other cases, intents could be less informative

and need contextual information, such as entities, to represent the meaning of a message. Also, for tasks that need to differentiate users based on specific information that is captured by entities, the user model could benefit from this additional information. In this direction, the same clustering algorithms are run on another environment in which the intents alone are not sufficient to tell the meaning of a message. For instance, the same intent (*#Time*) is used to find temporal information about an exam, a lecture and a deadline of the challenge, and the VTA is able to discriminate the cases with the recognition of an exam, lecture or deadline entity. In this scenario, the proposed user model consists of a vector in which the dimensions are not only the intents, but the result of the cartesian product between intents and entities, which is filtered to retain only couples (intent, entity) that can actually occur during a conversation. The structure of this second simulation is the same as the previous one, but the sets of intents for the different conversational topics were much more overlapping and, in order to mimic the user behavior, the samples of the intents must be divided into coherent groups. For instance, when picking random samples for a user interested in lectures, if the ambiguous *#Time* intent is selected from the lecture group of intents, only samples that talk about lectures are admitted.

5.4.3 Techniques

The clustering experiments on the simulated data are implemented in Python with public libraries: in particular, K-Medoids is present in the "pyclustering" library [21] while K-Means and Hierarchical Agglomerative clustering algorithms, as well as the clustering evaluation metrics, are provided by the "scikit-learn" library [23].

K-Medoids clustering is run with a square Euclidean distance metric and 50 random initializations. On the same data, this method with a Manhattan distance was not as performant. At the end of the clustering process, the partition with the highest silhouette score is taken as result.

K-Means clustering is also run with 50 initializations and the resulting partitioning is the one which minimizes the Within-Clusters-Sum-of-Squares.

Hierarchical Agglomerative clustering is run with a Euclidean distance and Ward linkage. This configuration beat others with Manhattan distance or with average linkage on the same data.

The clustering algorithms are executed to find the four clusters of "exam", "lecture", "challenge" and "content" users, with and without the interactions of "mixed"

users. During the evaluation of the partitions, the "mixed" users are excluded from the ground truth and from the predicted clusters, in order to measure the ability of the algorithms to divide the groups of users that showed a specific interest. Different variations of the dataset and the user model have been explored, as described in the following subsection.

5.4.4 Results

5.4.4.1 Scenario A - only intents

On the simulated interactions containing the intents described in Subsection 4.3.1, the different clustering algorithms were able to find the clusters contained in the ground truth, obtaining the results listed in the following charts. Each figure shows the average performance of 10 runs of the different techniques with respect to one quality measure and for each of the algorithms, they show several variants which are applied on the dataset and on the user model. In fact, there are results for both the application of the algorithms on the datasets with and without the "mixed" users, and orthogonally to the dataset types, there are the different user models: one containing all the intents and one containing only part of the intents. The discarded ones are: *#Goodbye*, *#Greetings*, *#Thanks*, *#General_About_You*, *#General_-Chatbot_Capabilities*, *#General_Human_or_Bot*, *#General_Jokes*, *#General_Negative_-Feedback*, *#General_Security_Assurance*, *#RelatedTopics*, *#Teacher_Contact*, *#Teacher_-Info*, *#Teaching_Assistants_Info*, *#Okay*, *#Really*. They have been filtered out of the model since they are expected to bring less information than the others, for the goal of separating the 4 clusters specified in the ground truth.

The results show that with the simpler user model caused by the filtering of intents, K-Medoids was able to better divide the users in the groups they belong to, while the other algorithms were not affected by this change, apart from the Agglomerative clustering in the dataset filled with "mixed" users. In the scenario in which the "mixed" users are added to the dataset, the performance of every algorithm decreases, probably due to the noise added by those new users. In order to improve the partitioning, it is possible to find more clusters and reassemble them later. In this sense, the best result was found by instructing the Agglomerative algorithm to find 6 clusters instead of 4, reaching a homogeneity of 1.0, meaning that each cluster contained only users of a single group, but those groups are spread out in more than one predicted cluster. From the results shown in the several charts, the clustering algorithm that proved to be the most reliable for this specific experiment is K-Means, since it was able to perform equally or better than the

others in every situation.

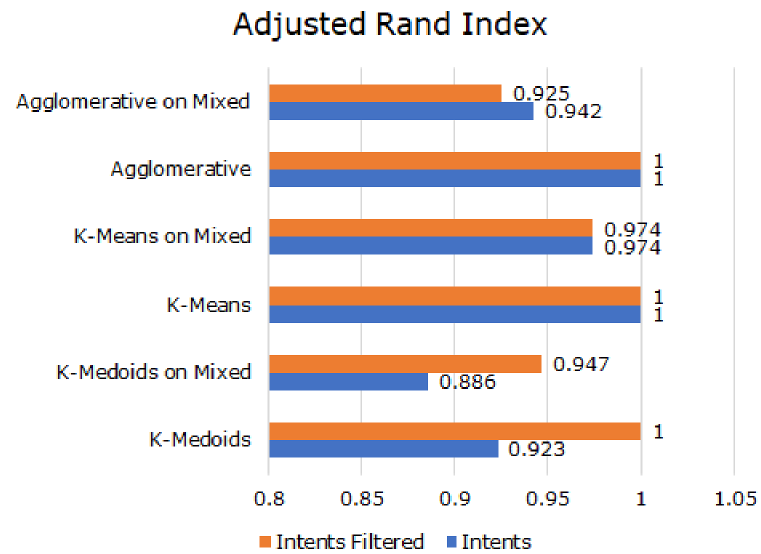

Figure 5.2: Adjusted Rand Index in Scenario A

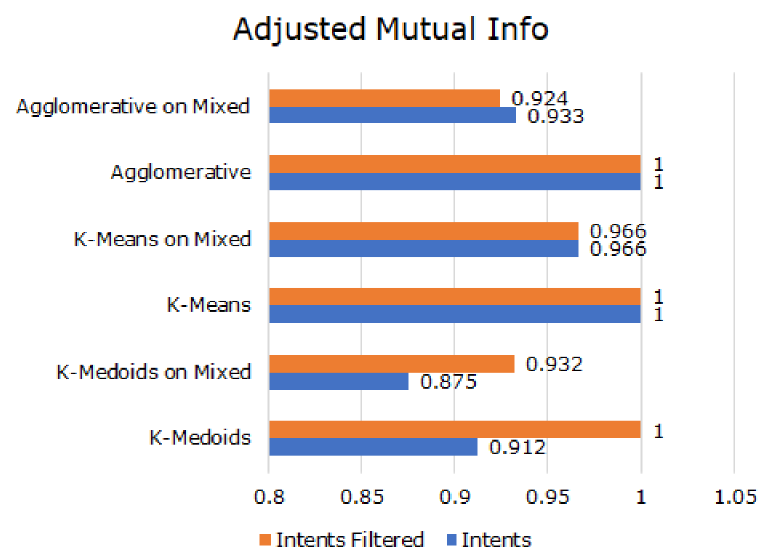

Figure 5.3: Adjusted Mutual Info in Scenario A

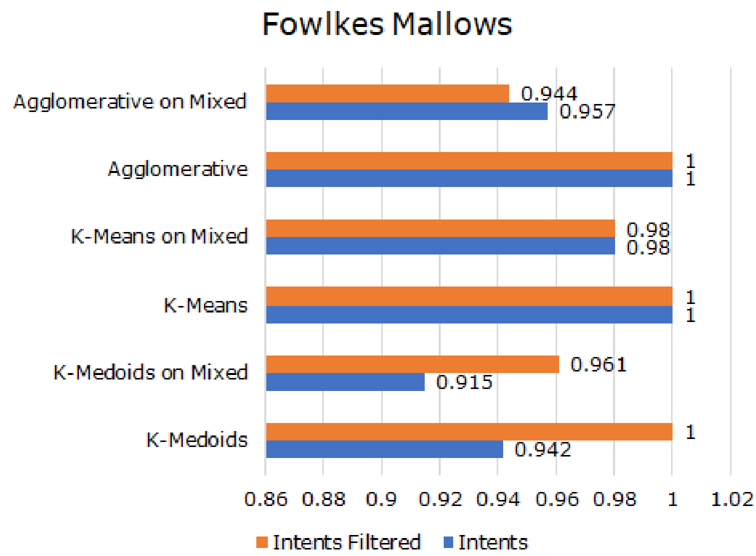

Figure 5.4: Fowlkes-Mallows measure in Scenario A

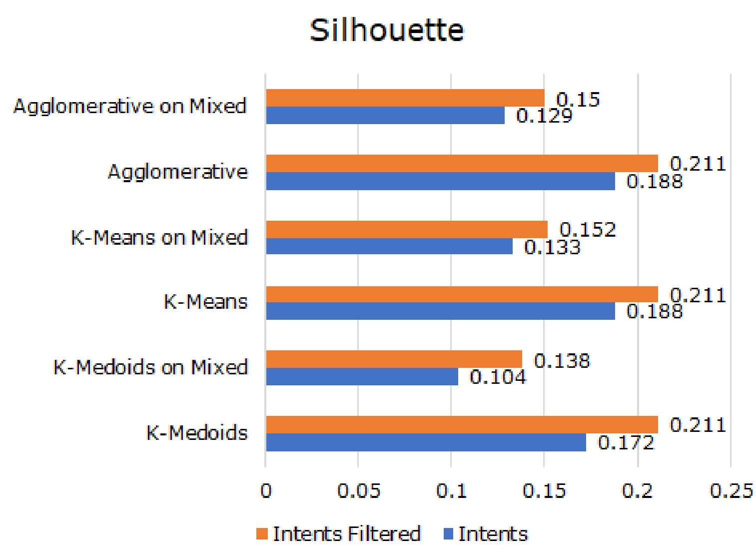

Figure 5.5: Silhouette coefficient in Scenario A

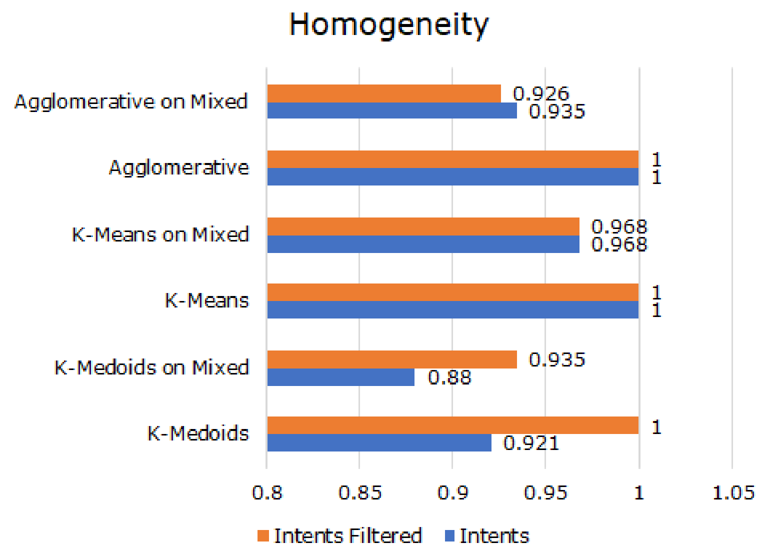

Figure 5.6: Homogeneity in Scenario A

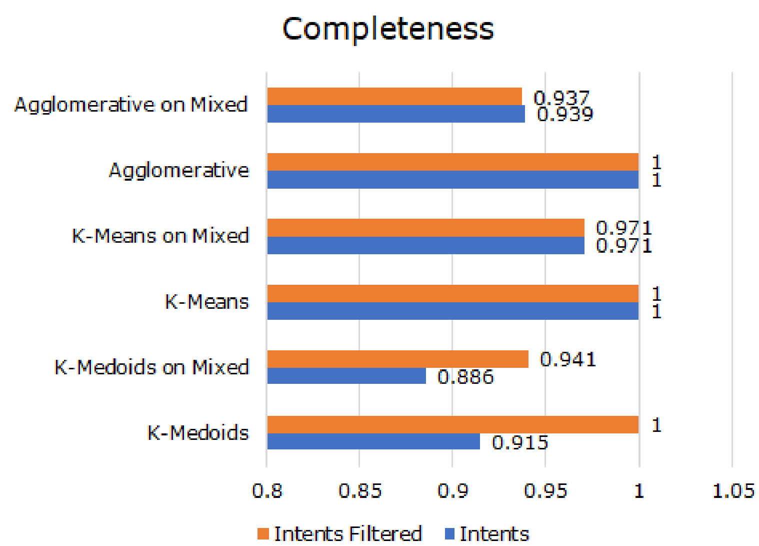

Figure 5.7: Completeness in Scenario A

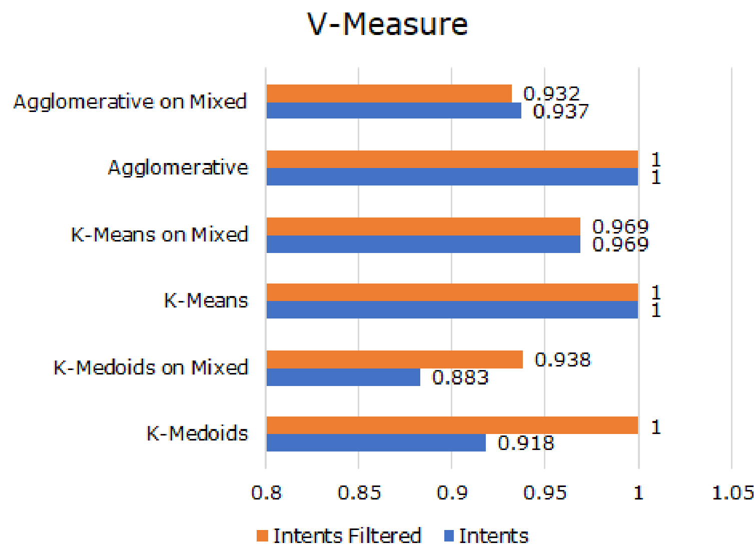

Figure 5.8: V-measure in Scenario A

5.4.4.2 Scenario B - intents and entities

In this section the evaluation of the same clustering algorithms is presented for the case in which the simulated interactions contain a new set of intents and entities, which are used to recognize the same questions that users can ask to Rexy, but in a different way. In this scenario, the role of the entities is crucial in understanding questions related to the organization of the course, which share the same basic intents and rely on the recognition of entities that represent exams, lectures and deadlines. The intents to be discarded are the same as the previous scenario. In this case, the user model can comprise a combination of intents and entities, which brings an improvement over the simpler user model made out of intents only. This result is shown in the following charts, that follow the same guidelines of the ones presented in the previous subsection, while also including the new type of user model.

When the "mixed" users are included in the dataset, the performance of the algorithms decreases, except for K-Means. In this scenario, deleting the less informative intents from the user model, brings to an improvement of the clustering only with the K-Medoids algorithm when the "mixed" users are added to the dataset. It also improves the silhouette score of all the algorithms. As in the previous scenario, the most reliable clustering algorithm for this specific experiment is K-Means, as shown by the following figures.

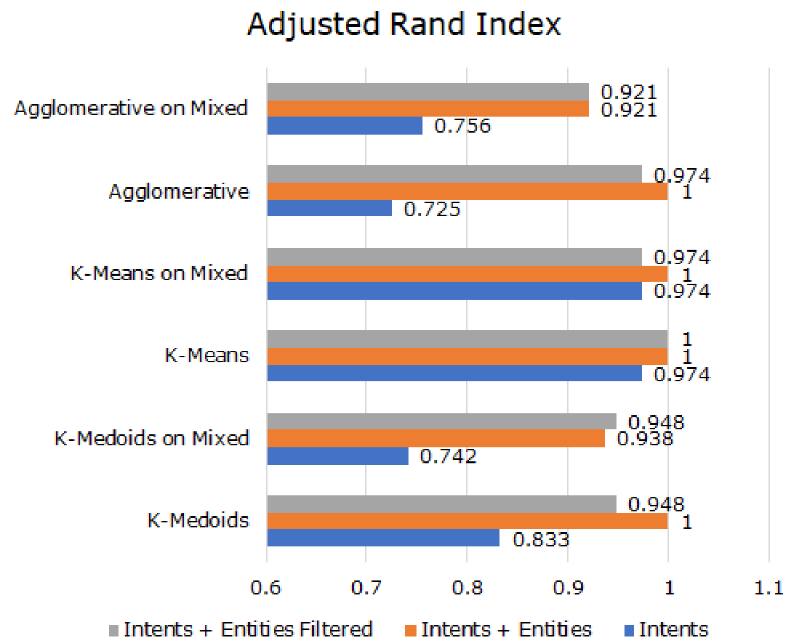

Figure 5.9: Adjusted Rand Index in Scenario B

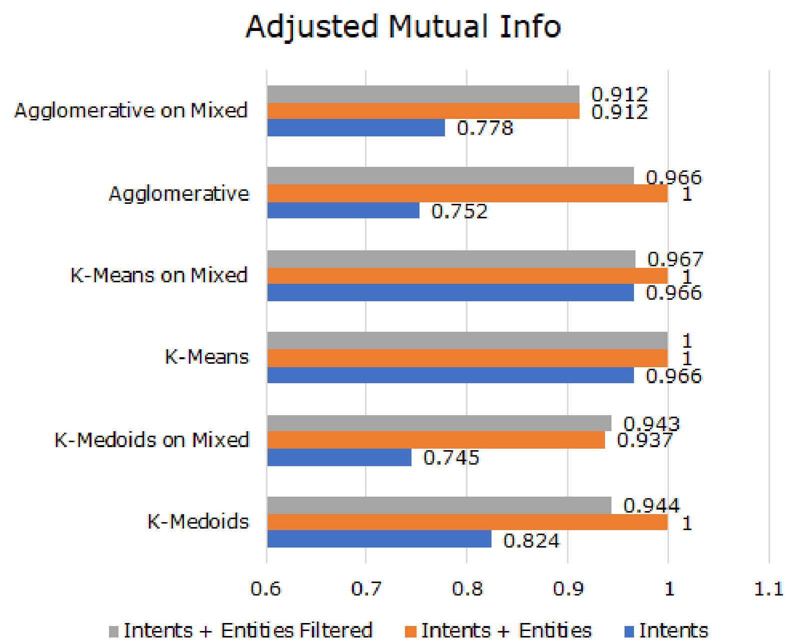

Figure 5.10: Adjusted Mutual Info in Scenario B

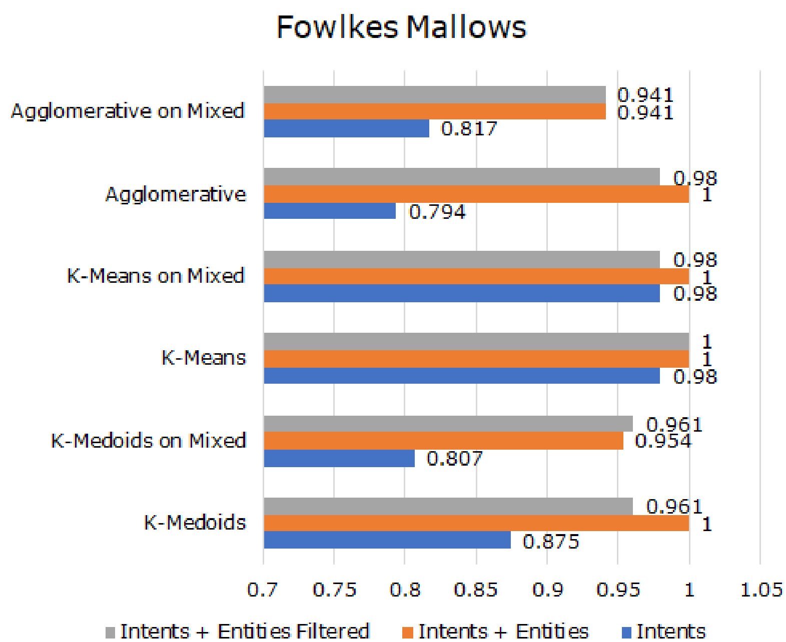

Figure 5.11: Fowlkes-Mallows measure in Scenario B

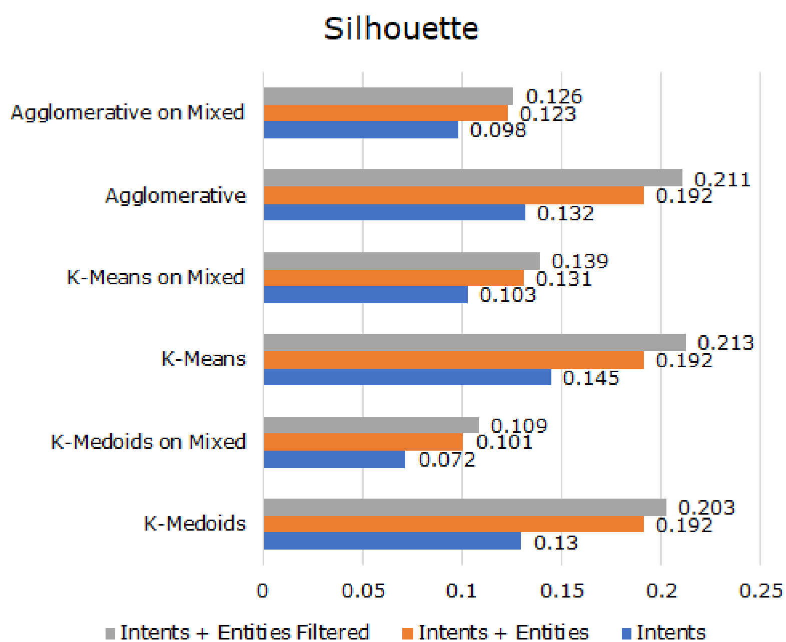

Figure 5.12: Silhouette coefficient in Scenario B

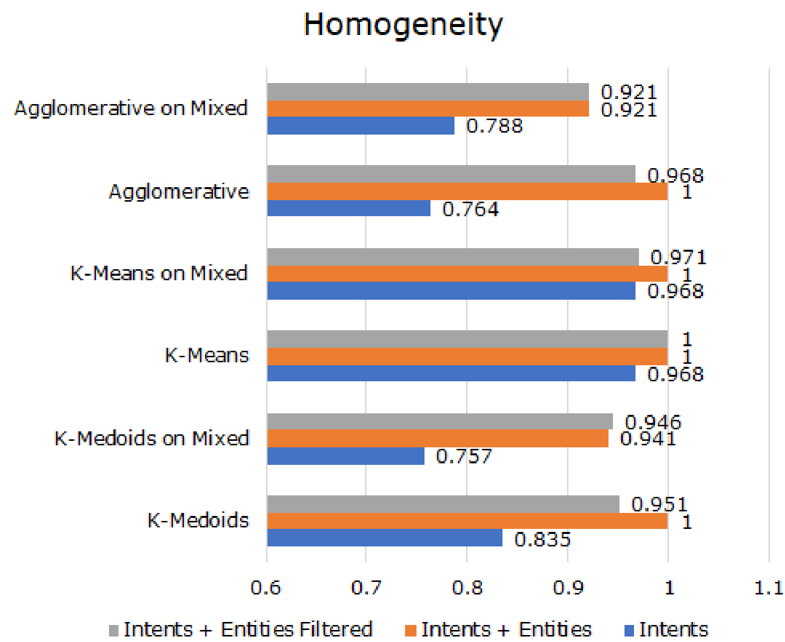

Figure 5.13: Homogeneity in Scenario B

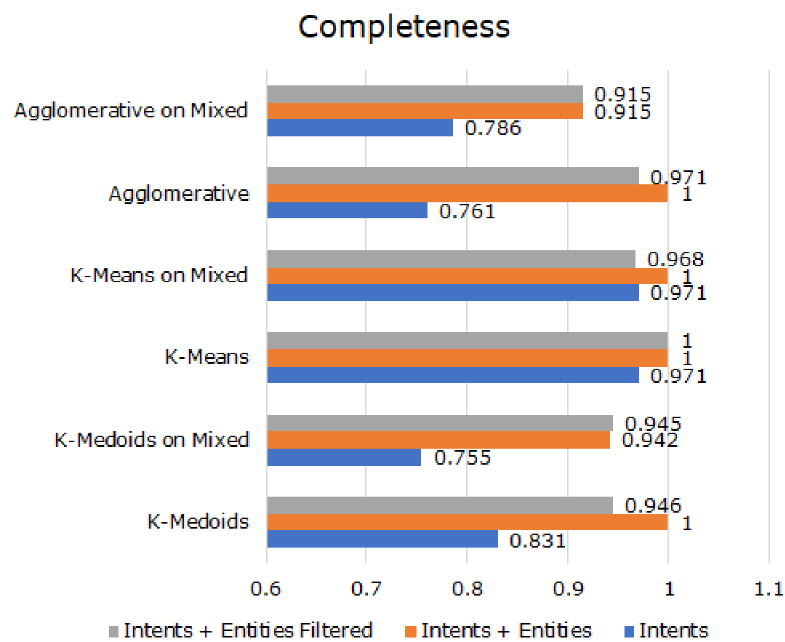

Figure 5.14: Completeness in Scenario B

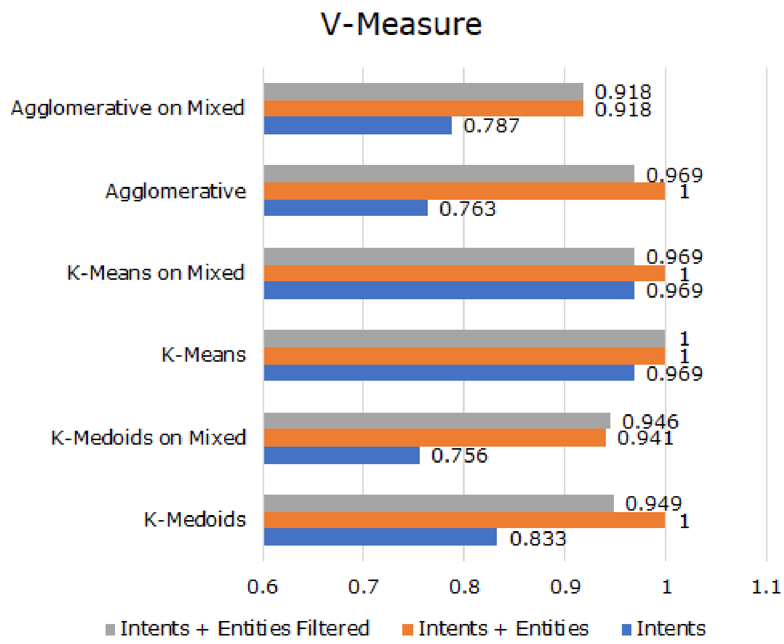

Figure 5.15: V-measure in Scenario B

Conclusions

This work shows the design of Rexy, a VTA for an on-site university level course, as a solution for the automatic handling of FAQs regarding the organization and the content of the course. While being able to reply efficiently to expected questions, it is also capable of engaging students with multi-choice questions and guarantee some degree of control to the human TAs on the conversations that students have with the VTA.

The application that is presented in this thesis has a database tier containing information about the organization of the course (exams, lectures, deadlines) and about its content (modules, entities, chapters).
A Watson™ Assistant workspace is used to analyze the messages coming from students, in particular for running intent classification and entity recognition tasks. This service provides a way to manage conversations using a dialog tree, in which the nodes define how the assistant is going to answer in each situation.
A server application handles messages coming from students and their replies. It allows to retrieve information from the database tier to create the answers when Assistant does not have enough knowledge to do so. It is also used to enrich the results of the intent and entity analysis performed by Assistant with elements coming from the context of the conversation, in order to better identify the right answers. Furthermore, it allows to save the interactions between the VTA and students and to improve its understanding capabilities in a continuous learning scenario, by sending special requests to human TAs when Assistant is not able to handle a student's question.
The front end consists in the integration with Slack, a messaging application that has been used by students and teachers to interact with Rexy. The employment of another popular messaging application or a custom one is possible.

Rexy was deployed in the first semester of the academic year 2018/2019 at Politecnico di Milano for the *Recommender Systems* course, but can be adapted to other on-site and online courses, by substituting parts of the knowledge base of the

VTA, which is composed by the databases containing information about the course and its content, and course dependent answers. During its activity, Rexy was able to reply correctly to the majority of the questions coming from the students, which were mostly related to the organization of the course. Even though some of the messages were not foreseen and thus misunderstood by Rexy, the human TAs were enabled to instruct it on how to respond improving its comprehension capabilities and replying to the students which questions were left unanswered.

The proposed VTA could still benefit from the addition of new functionalities, which could make it more interesting and useful for students and adopters of this solution. Therefore, future work is needed in order to study the effect that the presence of such VTA has in a course, either on-site or online, considering the limited number of interactions that Rexy had with real students it is not possible to state if it is really able to improve teaching, student engagement and reduce the number of requests asked to teachers. Some work can lead to the addition of features aimed at improving student engagement, and more importantly measure their effectiveness on a broad enough number of courses and demography of users. Another type of research could focus on automating the creation of the knowledge base of the VTA, in particular of the database containing information about the content of the course. If satisfactory results can be obtained in this sense, the adaptation of the proposed VTA to other courses becomes straightforward and much less time demanding. The adoption of new, ad hoc solutions for understanding the requests coming from students could bring much more flexibility for the functionalities of the VTA, and the creation of responses from scratch or from official resources like documents, articles and books could improve the quality of its replies, while extending its knowledge automatically. Another line of research could consider the possibilities of implementing personalized learning with a VTA, which should bring a personalization of the interactions with every different student or groups of similar students.

Bibliography

[1] Alexa for Business. URL: https://aws.amazon.com/alexaforbusiness/.

[2] Artillery. URL: https://artillery.io/docs/.

[3] Botkit. URL: https://botkit.ai/.

[4] Chatfuel. URL: https://chatfuel.com/.

[5] DialogFlow. URL: https://dialogflow.com/.

[6] IBM Watson. URL: https://www.ibm.com/watson/.

[7] IBM Watson Assistant. URL: https://www.ibm.com/watson/ai-assistant/.

[8] IBM Watson Assistant APIs documentation. URL: https://console.bluemix.net/apidocs/assistant?language=node.

[9] IBM Watson Assistant documentation. URL: https://console.bluemix.net/docs/services/conversation/index.html.

[10] IBM Watson documentation. URL: https://console.bluemix.net/developer/watson/documentation.

[11] Microsoft Azure Cognitive Services. URL: https://azure.microsoft.com/en-us/services/cognitive-services/.

[12] Microsoft Bot Framework. URL: https://dev.botframework.com/.

[13] Microsoft LUIS. URL: https://www.luis.ai/home.

[14] MongoDB documentation. URL: https://docs.mongodb.com/.

[15] Mongoose. URL: https://mongoosejs.com/.

[16] MySQL documentation. URL: https://dev.mysql.com/doc/.

[17] MySQL driver. URL: https://github.com/mysqljs/mysql.

[18] npm. URL: https://www.npmjs.com/.

[19] Pandorabots. URL: https://home.pandorabots.com/home.html.

[20] pm2. URL: http://pm2.keymetrics.io/.

[21] pyclustering. URL: https://github.com/annoviko/pyclustering.

[22] Recommender System 2018 Challenge Polimi on Kaggle. URL: https://www.kaggle.com/c/recommender-system-2018-challenge-polimi.

[23] scikit-learn. URL: http://scikit-learn.org/stable/modules/clustering.html.

[24] Slack. URL: https://slack.com/.

[25] Slack APIs rate limits. URL: https://api.slack.com/docs/rate-limits.

[26] Slack Bot Users. URL: https://api.slack.com/bot-users.

[27] Wit.ai. URL: https://wit.ai/.

[28] Messaging apps are now bigger than social networks. URL: https://www.businessinsider.com/the-messaging-app-report-2015-11?IR=T, Sept. 2016.

[29] Brandtzaeg, P. B., and Følstad, A. Why people use chatbots. In *International Conference on Internet Science* (2017), Springer, pp. 377–392.

[30] Dale, R. The return of the chatbots. *Natural Language Engineering 22*, 5 (2016), 811–817.

[31] Daniel, J. Making sense of MOOCs: Musings in a maze of myth, paradox and possibility. *Journal of interactive Media in education 2012*, 3 (2012).

[32] du Boulay, B. Artificial Intelligence as an Effective Classroom Assistant. *IEEE Intelligent Systems 31*, 6 (Nov 2016), 76–81.

[33] Ferrucci, D., Brown, E., Chu-Carroll, J., Fan, J., Gondek, D., Kalyanpur, A. A., Lally, A., Murdock, J. W., Nyberg, E., Prager, J., et al. Building Watson: An overview of the DeepQA project. *AI magazine 31*, 3 (2010), 59–79.

[34] Gabbatt, A. IBM computer Watson wins Jeopardy clash. URL: https://www.theguardian.com/technology/2011/feb/17/ibm-computer-watson-wins-jeopardy, Feb. 2011.

[35] Goel, A. K., and Polepeddi, L. Jill Watson: A Virtual Teaching Assistant for Online Education. Tech. rep., Georgia Institute of Technology, 2016.

[36] Hartigan, J. A., and Wong, M. A. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics) 28*, 1 (1979), 100–108.

[37] Jain, A., and Dubes, R. Algorithms for clustering data.

[38] Jia, J. The study of the application of a web-based chatbot system on the teaching of foreign languages. In *Society for Information Technology & Teacher Education International Conference* (2004), Association for the Advancement of Computing in Education (AACE), pp. 1201–1207.

[39] Jurafsky, D., and Martin, J. Dialog Systems and Chatbots. *Speech and language processing* (2017).

[40] Kaufman, L., and Rousseeuw, P. *Clustering by means of medoids.* North-Holland, 1987.

[41] Kernighan, M. D., Church, K. W., and Gale, W. A. A Spelling Correction Program Based on a Noisy Channel Model. In *Proceedings of the 13th Conference on Computational Linguistics - Volume 2* (Stroudsburg, PA, USA, 1990), COLING '90, Association for Computational Linguistics, pp. 205–210.

[42] Kukich, K. Techniques for Automatically Correcting Words in Text. *ACM Comput. Surv. 24*, 4 (Dec. 1992), 377–439.

[43] Norvig, P. How to Write a Spelling Corrector. URL: http://norvig.com/spell-correct.html.

[44] Pirrone, R., Pilato, G., Rizzo, R., and Russo, G. Learning path generation by domain ontology transformation. In *AI* IA 2005: Advances in Artificial Intelligence.* Springer, 2005, pp. 359–369.

[45] Sánchez-Diaz, X., Ayala-Bastidas, G., Fonseca-Ortiz, P., and Garrido, L. A Knowledge-based Methodology for Building a Conversational Chatbot as an Intelligent Tutor.

[46] Shah, D. By The Numbers: MOOCs in 2017. URL: https://www.class-central.com/report/mooc-stats-2017/, Jan. 2018.

[47] Shah, H., Warwick, K., Vallverdú, J., and Wu, D. Can machines talk? Comparison of Eliza with modern dialogue systems. *Computers in Human Behavior 58* (2016), 278–295.

[48] VanLehn, K. The relative effectiveness of human tutoring, intelligent tutoring systems, and other tutoring systems. *Educational Psychologist 46*, 4 (2011), 197–221.

[49] Vinyals, O., and Le, Q. A neural conversational model. *arXiv preprint arXiv:1506.05869* (2015).

[50] Weizenbaum, J. ELIZA - a computer program for the study of natural language communication between man and machine. *Communications of the ACM 9*, 1 (1966), 36–45.

[51] Woo, J. W. Messaging apps exceeded 6 billion combined monthly active users in 2017. URL: https://technology.ihs.com/602537/messaging-apps-exceeded-6-billion-combined-monthly-active-users-in-2017, May 2018.

[52] Yuan, L., and Powell, S. MOOCs and open education: Implications for higher education, 2013.

[53] Zaki, M. J., Meira Jr, W., and Meira, W. *Data mining and analysis: fundamental concepts and algorithms.* Cambridge University Press, 2014, ch. 17, pp. 425–466.

Appendix A

Dialog Tree

In this appendix the complete dialog tree present on the Assistant workspace is reported, with the names of the nodes and the jumps between them. They have a numeric notation that is used to identify the root and leaf nodes of branches. "LC" stands for low confidence and it is used to mark disambiguation nodes.

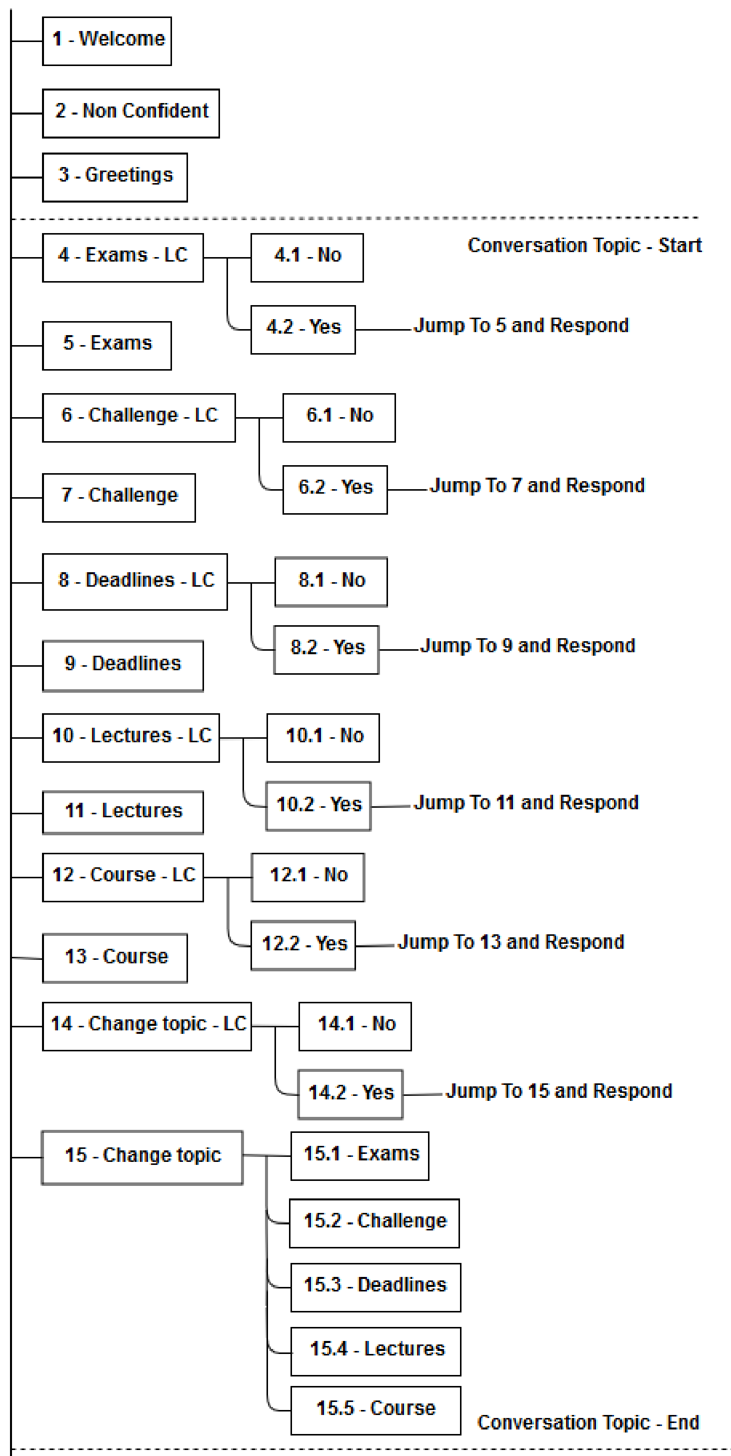

Figure A.1: Dialog tree: part 1

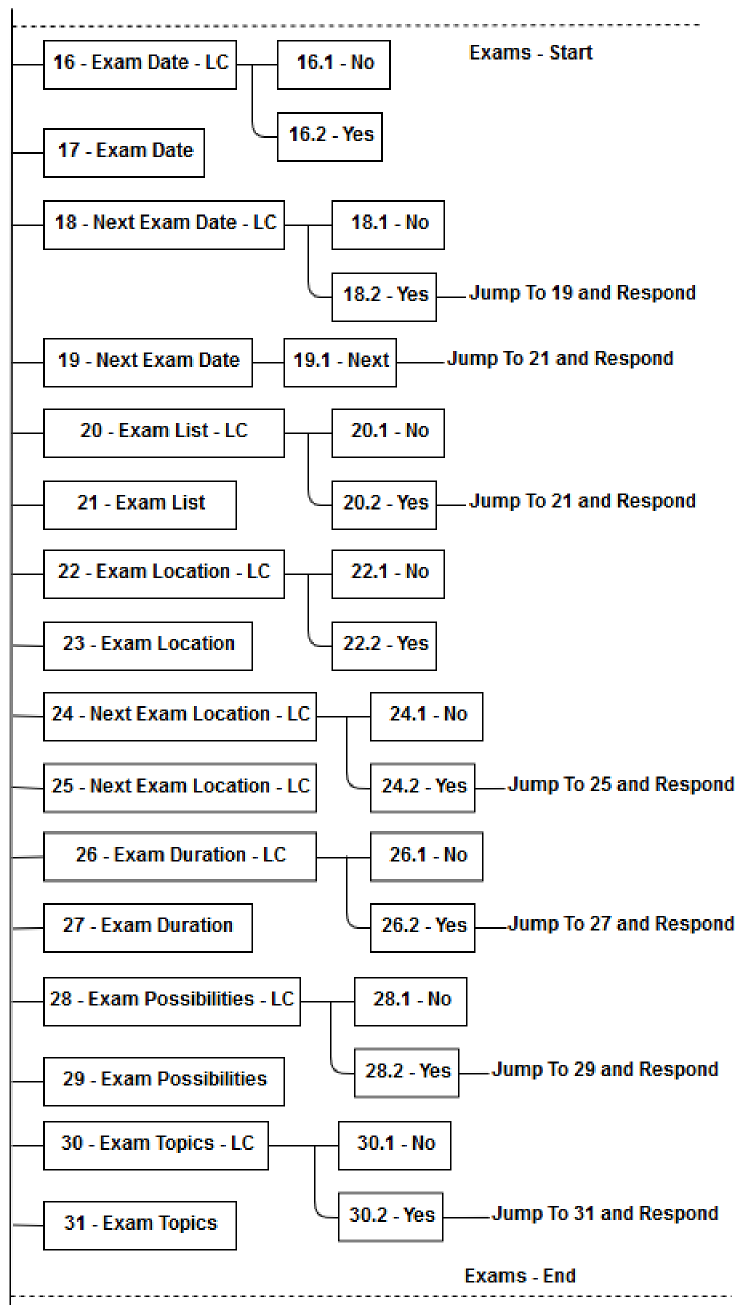

Figure A.2: Dialog tree: part 2

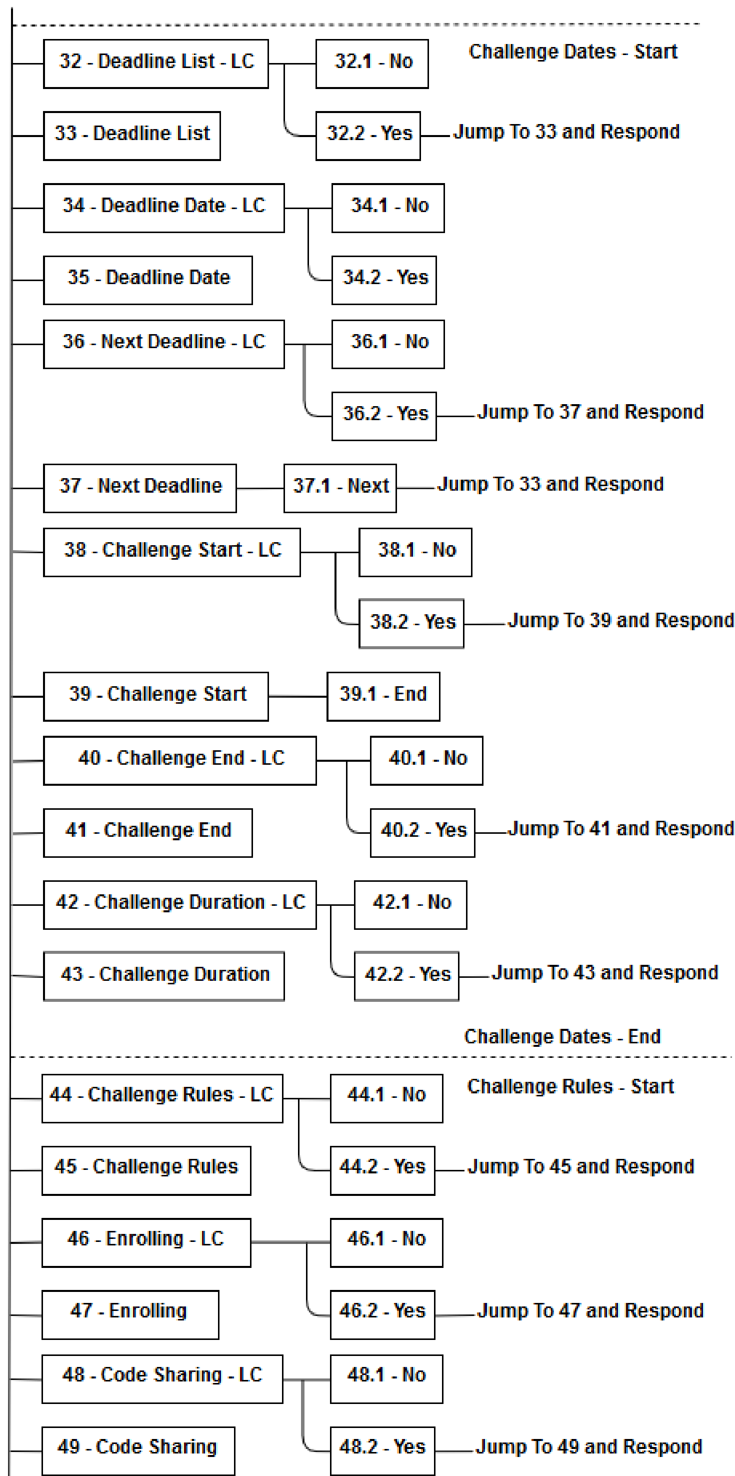

Figure A.3: Dialog tree: part 3

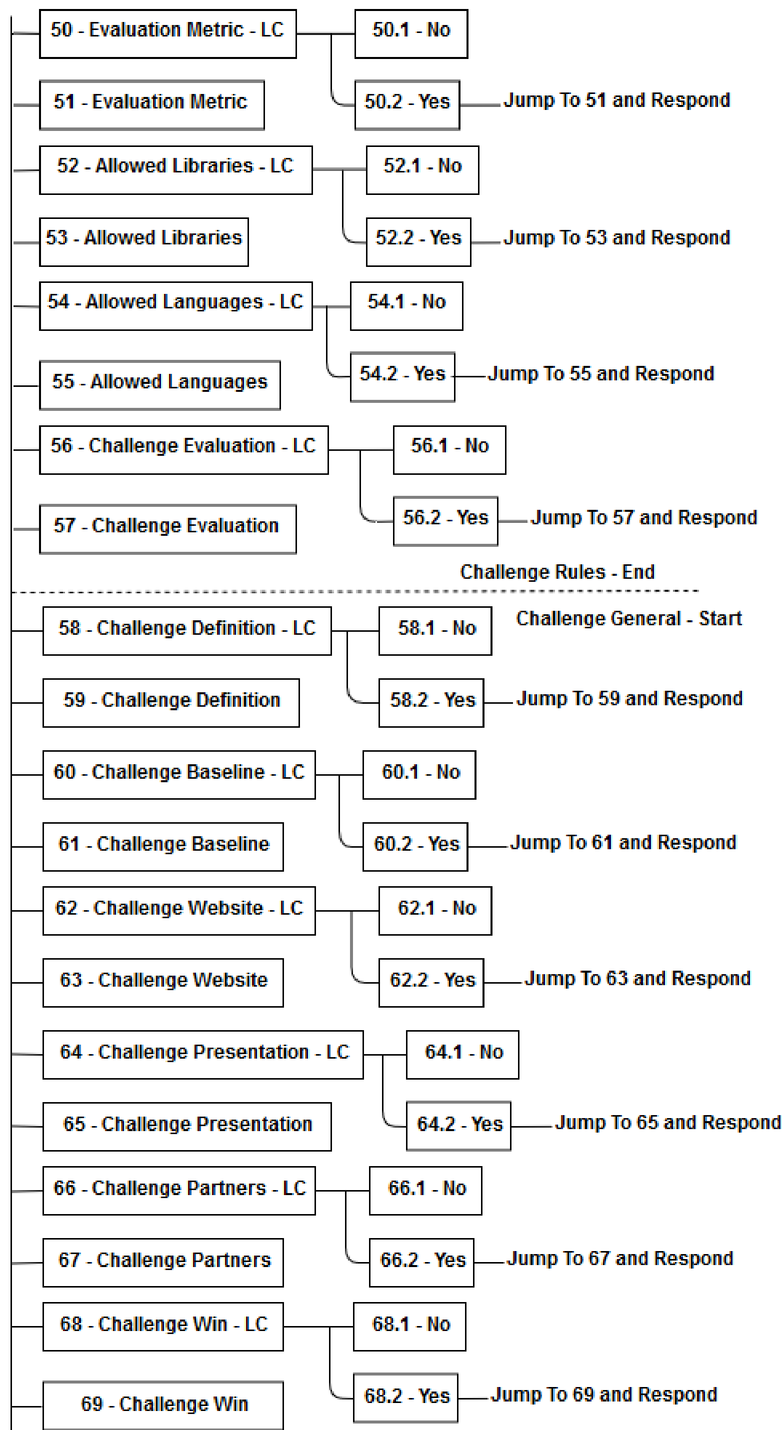

Figure A.4: Dialog tree: part 4

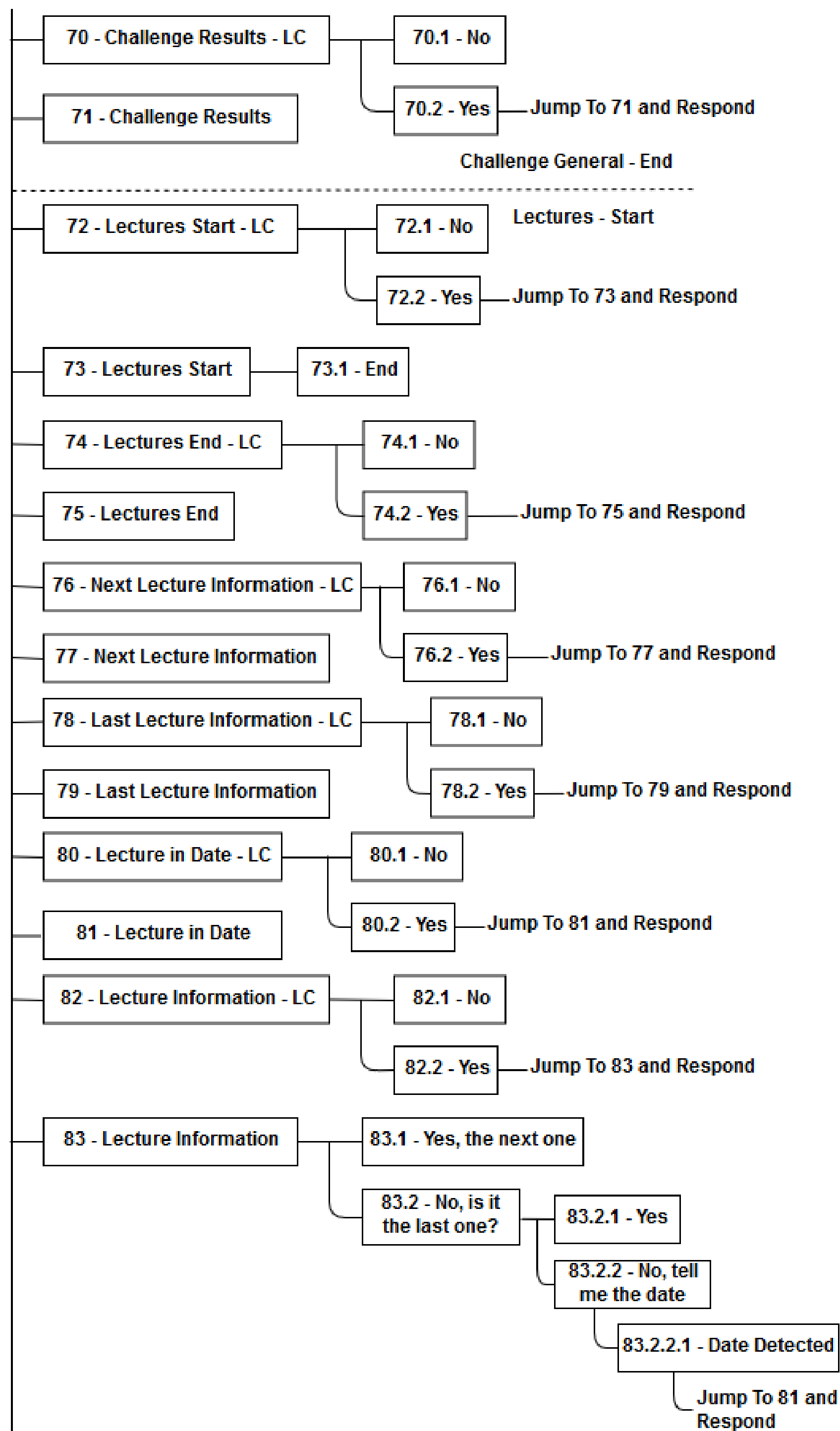

Figure A.5: Dialog tree: part 5

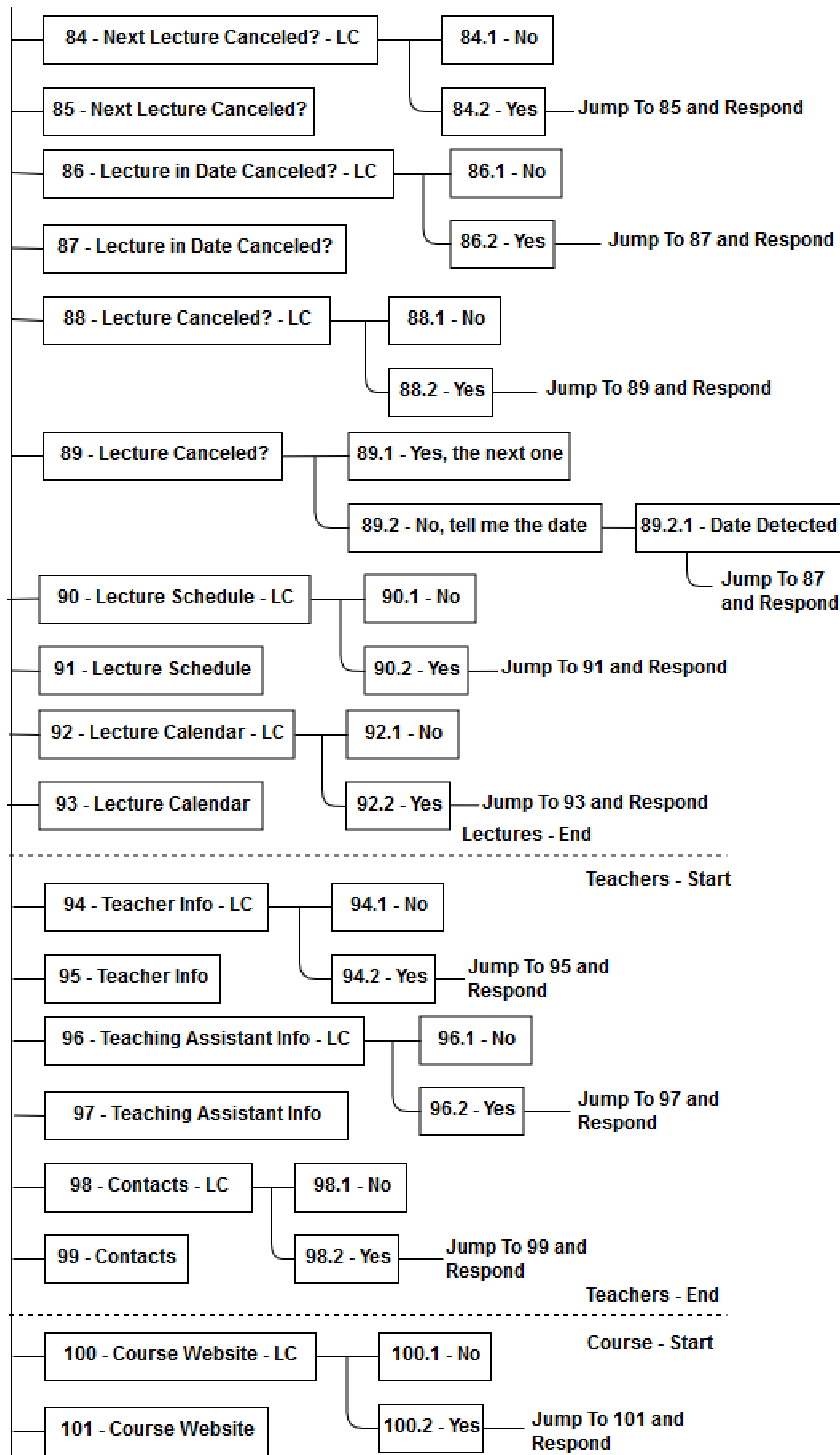

Figure A.6: Dialog tree: part 6

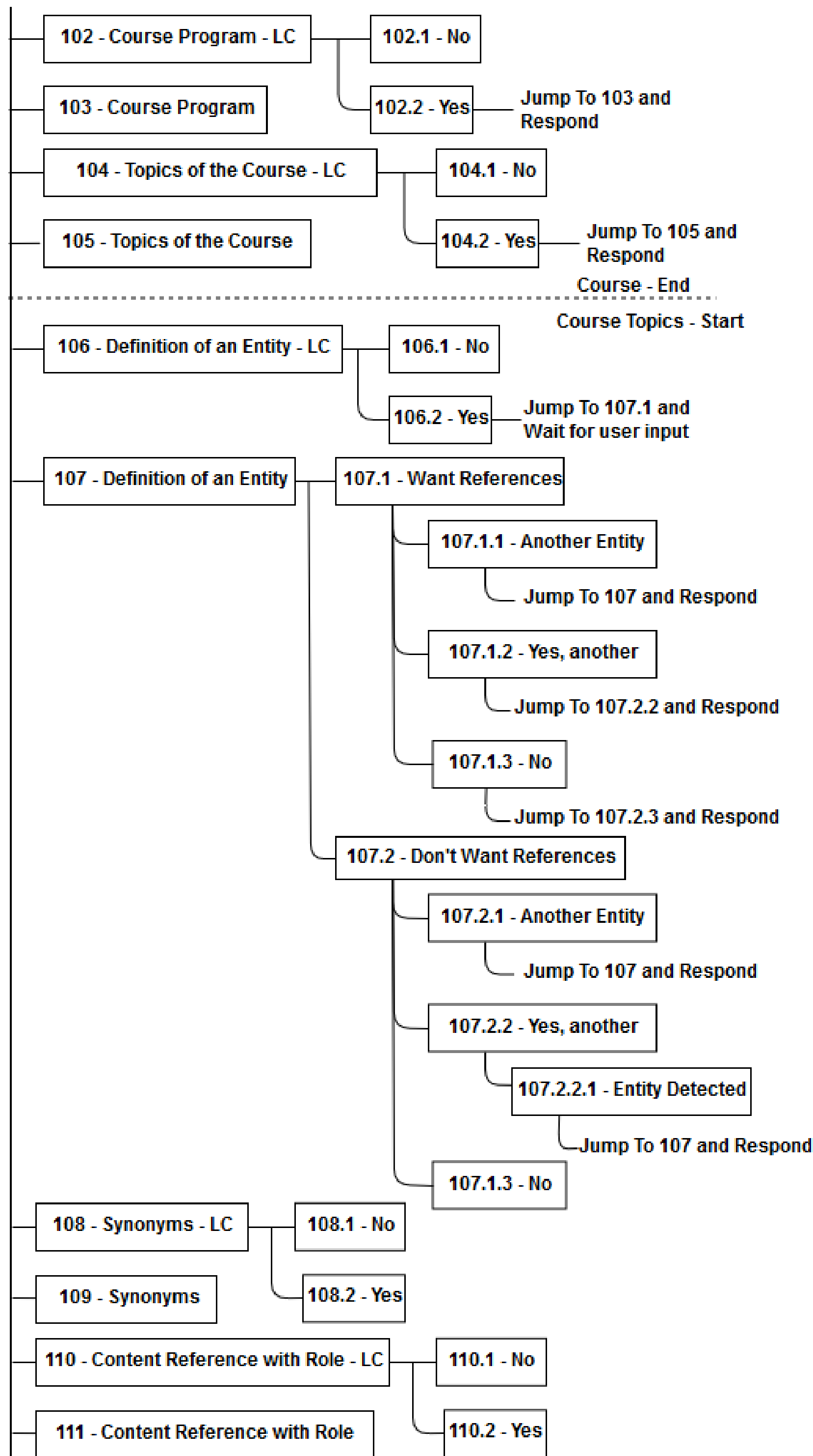

Figure A.7: Dialog tree: part 7

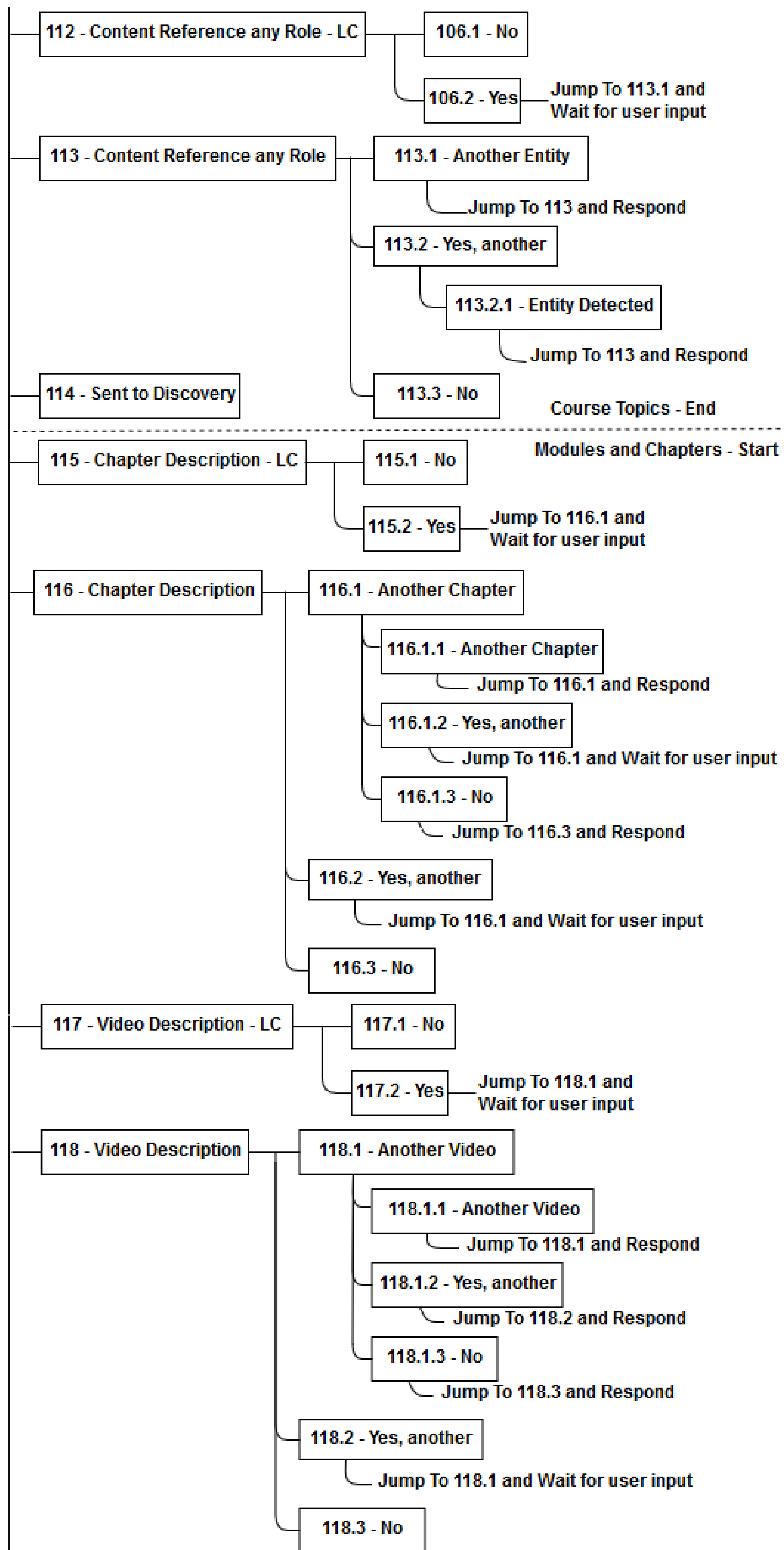

Figure A.8: Dialog tree: part 8

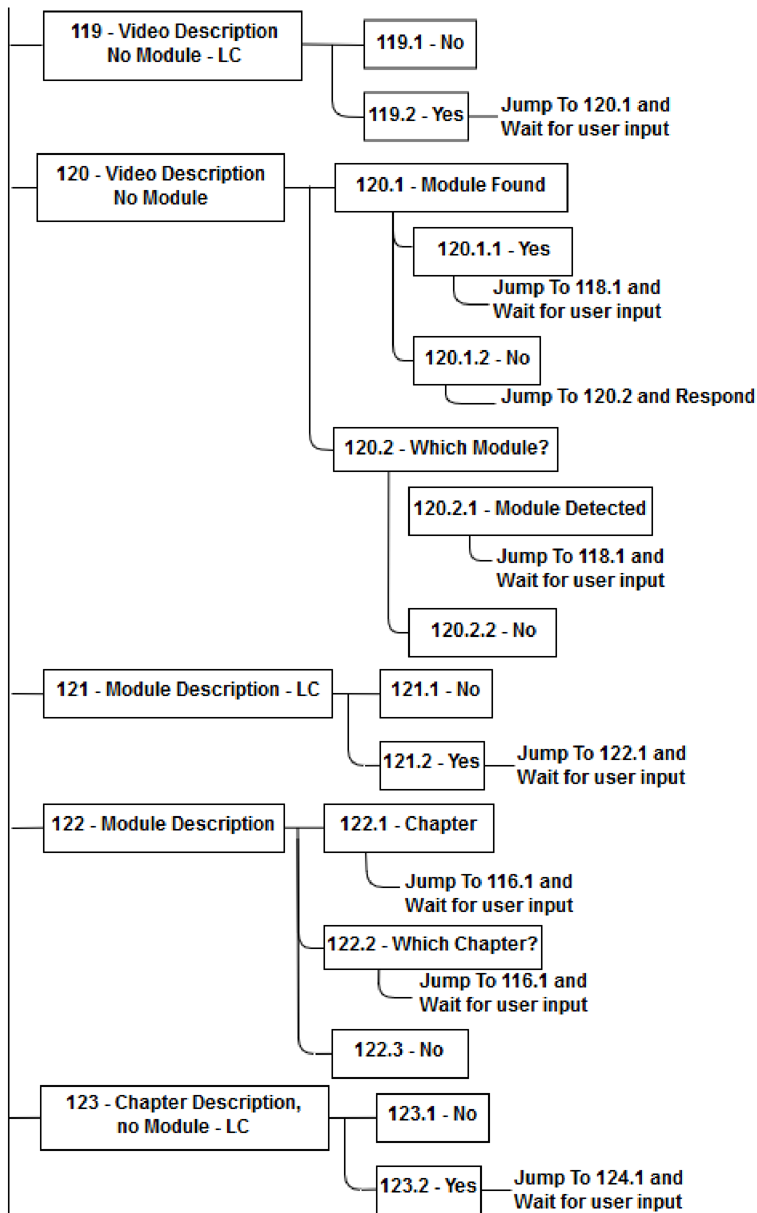

Figure A.9: Dialog tree: part 9

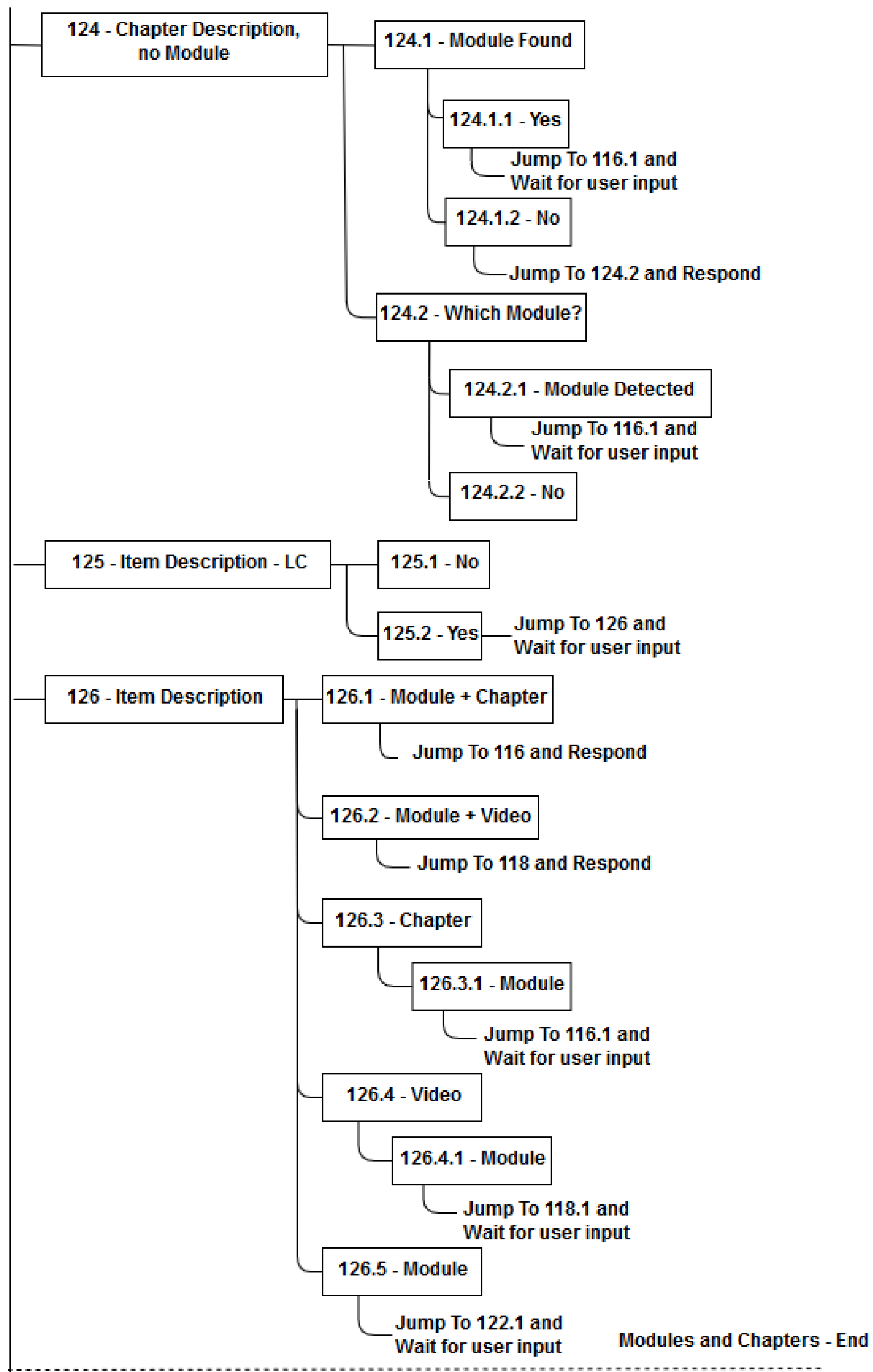

Figure A.10: Dialog tree: part 10

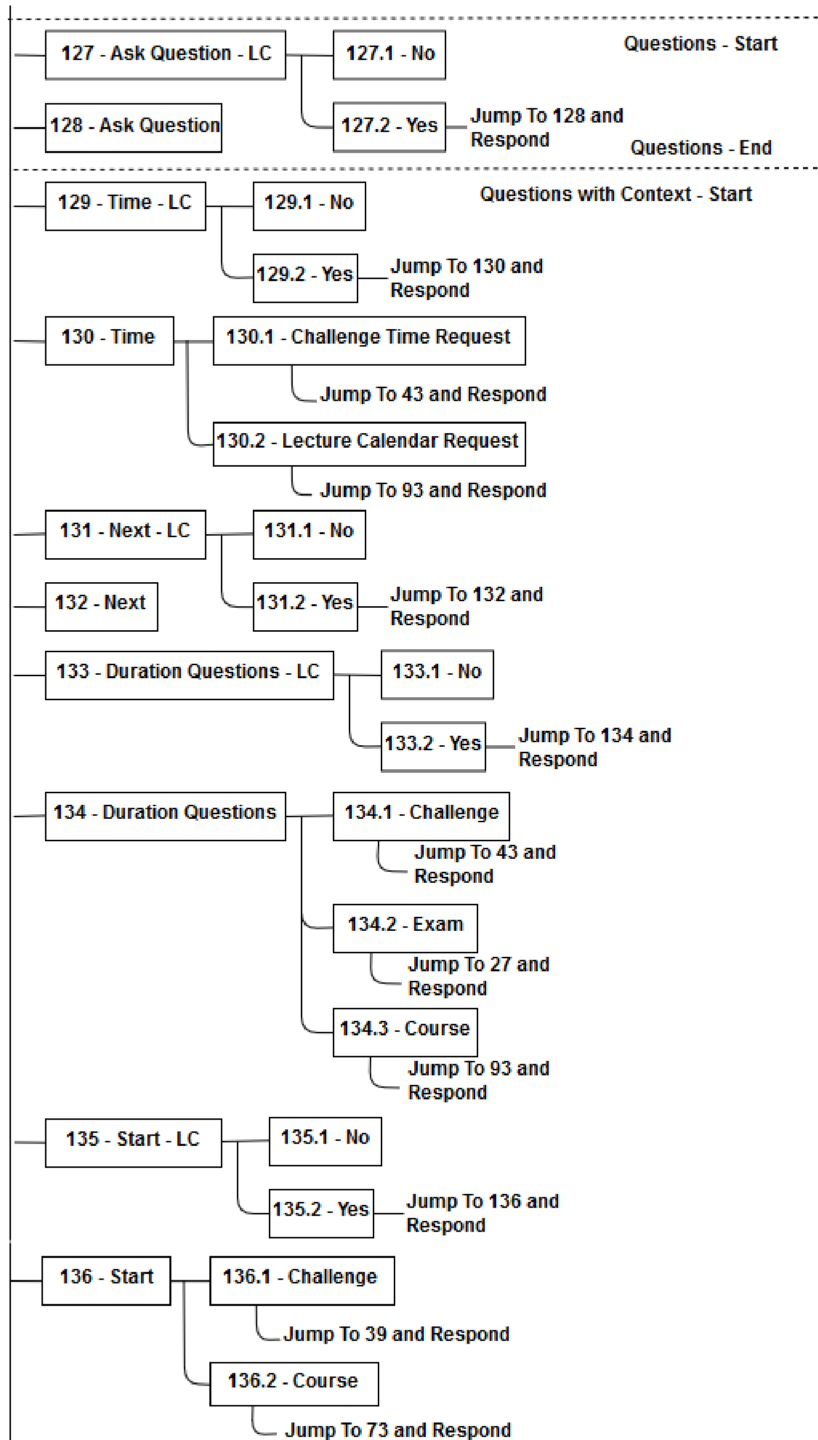

Figure A.11: Dialog tree: part 11

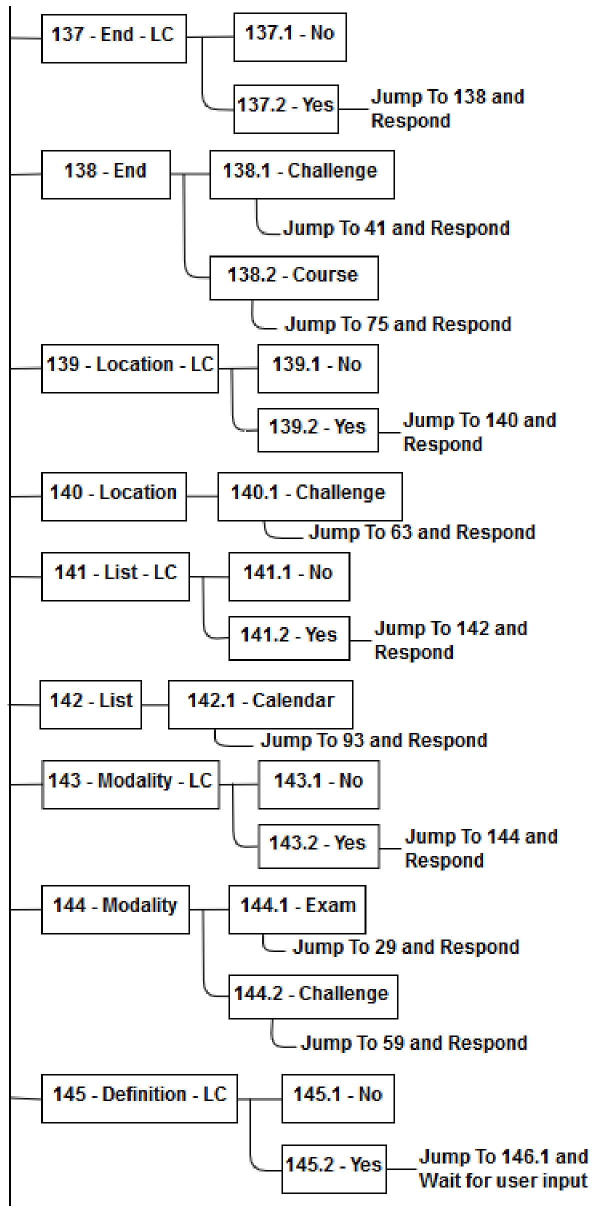

Figure A.12: Dialog tree: part 12

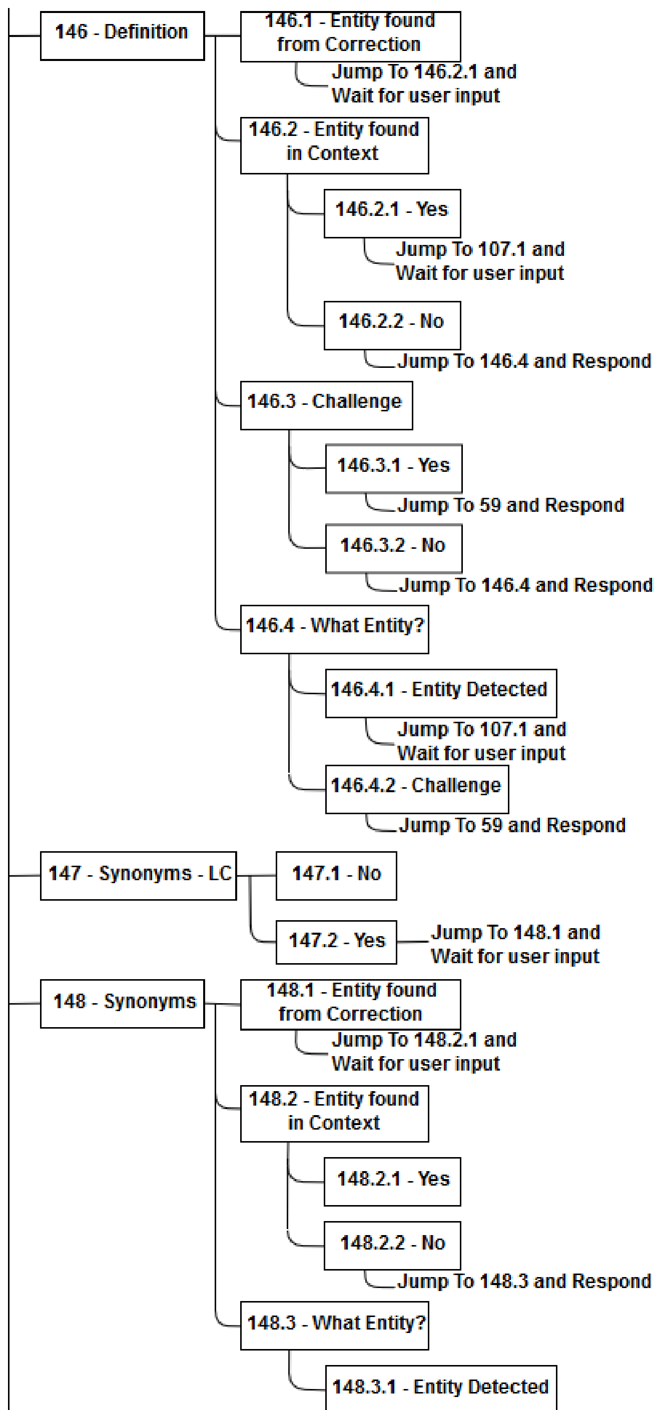

Figure A.13: Dialog tree: part 13

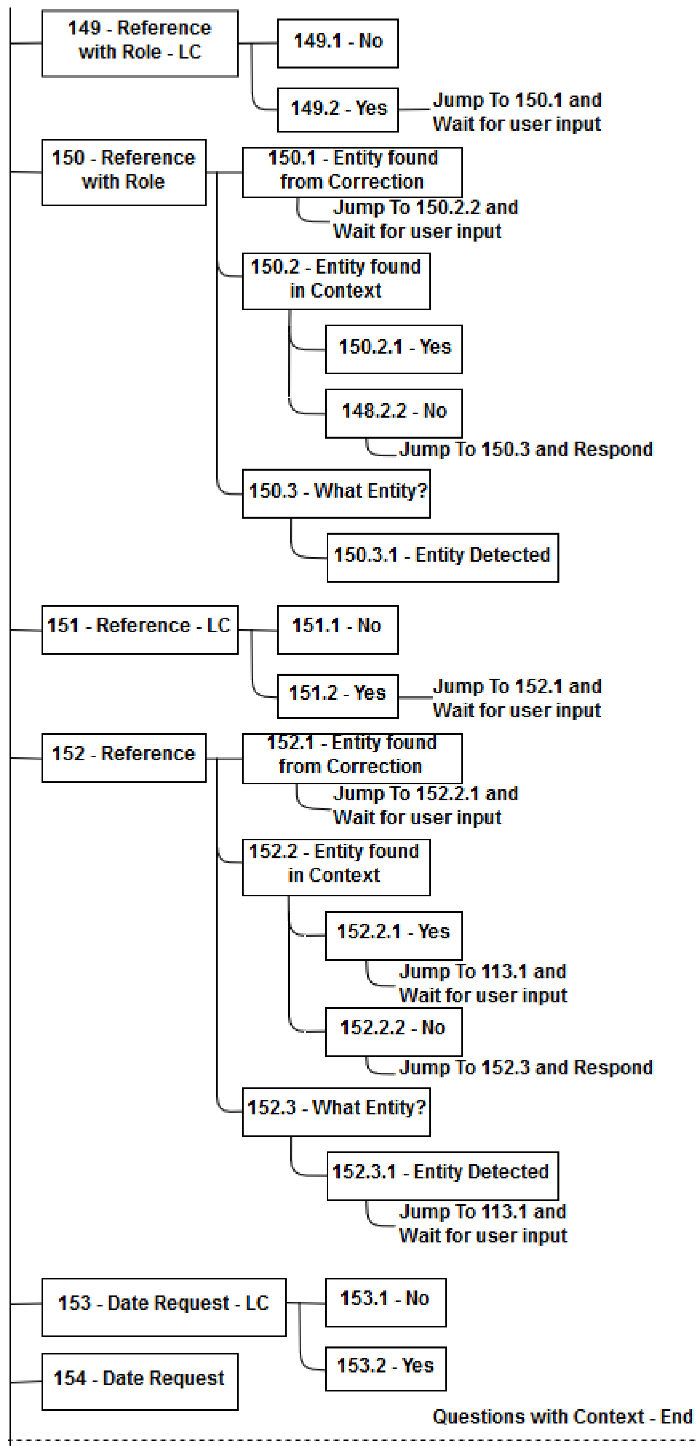

Figure A.14: Dialog tree: part 14

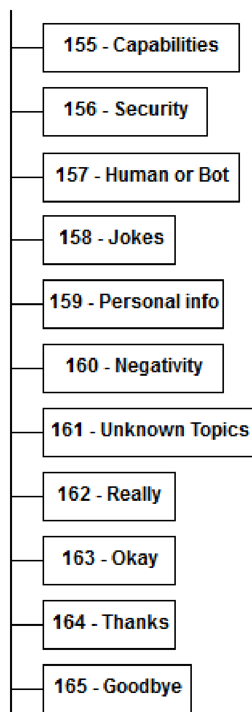

Figure A.15: Dialog tree: part 15

Appendix B

Use Cases

In this appendix some use cases of students interactions with the VTA are presented. They include input sentences that students might say, their classification in terms of intent and entities, the triggered node and the reply created by the VTA in different situations that might occur.

Table B.1: Use Case 1: Lecture schedule

Input	What's the lecture schedule?
Intent	#Lecture_Schedule
Entities	@administrative_entity: lecture
Dialog Node	91 – Lecture Schedule
Output	The schedule of this course is: Mondays: 10:15-13:15 in room E.G.6. Wednesdays: 12:15-14:15 in room D.0.1.

Table B.2: Use Case 2: Next lecture

Input	When is the next lecture?
Intent	#Lecture_Info_Nearest_Future
Entities	@administrative_entity: lecture
Dialog Node	77 – Next Lecture Information
Output in case a lecture is found	The next lecture will be on 2018-10-10 from 12:15 to 14:15 in room D.0.1. The teacher will be Cremonesi Paolo. It will be a theory lesson. The topic will be: Evaluation: Cold Start, New-Item and New-User problems.
Output in case a lecture is found but is suspended	The next lecture, scheduled for 2018-11-5 from 10:15 to 13:15, has been canceled. The next confirmed lecture will be on 2018-11-12 from 10:15 to 13:15 in room E.G.6. The teacher will be Cremonesi Paolo. It will be a theory lesson.
Output in case no lectures are found	Sorry, there are no scheduled lessons yet. Please ask for it in a while, this information may be available.

Table B.3: Use Case 3: Exam in september

Input	When is the exam in the september session?
Intent	#Exam_Date
Entities	@administrative_entity:exam @sys-date:2019-09-01 @sys-date:2019-09-30 @exam_session:september session
Dialog Node	17 – Exam Date
Output in case no exams are found	Sorry, there are no scheduled exams in this session yet. Please ask for it in a while, this information may be available.
Output in case an exam is found	The scheduled exam in the September session is on 2018-09-15 at 15:00 in room X.3.

Table B.4: Use Case 4: Exam possibilities

Input	How do I pass this course?
Intent	#Exam_Modality
Entities	@administrative_entity:course
Dialog Node	29 – Exam Possibilities
Output	You can pass this course in 3 different ways: 1) By participating in the course's challenge. This is the recommended path for all the students. At the end of the semester some groups will be asked to give a presentation about their work. 2) By doing an oral exam, which can be passed with a maximum of 26 points. Usually there are 5 exam calls distributed over the exam sessions and students can enroll to all of them. 3) By doing a project, given by the professor. Be aware that the number of available projects is limited.

Table B.5: Use Case 5: Starting day of the challenge

Input	When does the challenge start?
Intent	#Challenge_Start
Entities	@administrative_entity:challenge
Dialog Node	39 – Challenge Start
Output	The challenge starts on the 10th of October. Good luck!

Table B.6: Use Case 6: Second deadline of the challenge

Input	When is the second deadline?
Intent	#Challenge_Deadline_Date
Entities	@cardinality:2 @administrative_entity:deadline
Dialog Node	35 – Deadline Date
Output	The deadline is on 2018-12-1 at 23:59:00. Good luck!

Table B.7: Use Case 7: One of the challenge rules

Input	What are the baselines of the competition?
Intent	#Challenge_Baseline
Entities	@administrative_entity:challenge
Dialog Node	61 – Challenge Baseline
Output	A baseline is a recommender algorithm performance. In the challenge the baselines represent the performance of some anonym recommender algorithm. There are 10 of them and for each baseline that your score will exceed, you will be awarded 2 points, reaching a maximum of 20 points.

Table B.8: Use Case 8: Definition of a specified concept

Input	What is the definition of item?
Intent	#Definition
Entities	@reference_role:definition @course_entity:Item
Dialog Node	107 – Definition of an Entity
Output	This is the definition of Item: An item is any object that users can interact with, and that we might recommend to them. It might be a book, a movie, a hotel, a song and so on. Would you like to know where to find it in the modules?

Table B.9: Use Case 9: Synonym

Input	Are there other ways to call URM?
Intent	#Synonyms
Entities	@course_entity:URM
Dialog Node	109 - Synonyms
Output	A synonym of URM is User Rating Matrix.

Table B.10: Use Case 10: References of a concept of the course inside the modules

Input	Where can I find a schema of URM?
Intent	#Content_References
Entities	@reference_role:picture @course_entity:URM
Dialog Node	111 – Content Reference with Role
Output	There are pictures of User Rating Matrix (URM): In module 1, chapter 2.2 ”Data representation”. It is in video 2 at 2:19. In module 1, chapter 5.1 ”Formal representation of INPUT”. It is in video 5 at 0:10. In module 1, chapter 5.2 ”Sparsity of URM”. It is in video 5 at 1:05. In module 4, chapter 2.1 ”User Rating Matrix - URM”. It is in video 2 at 0:20.

Table B.11: Use Case 11: Description of a module

Input	What is the third module about?
Intent	#Item_Module_Description
Entities	@cardinality:3 @module:module 3
Dialog Node	122 – Module Description
Output	Here is a brief description of what you can find in Module 3: Content Based Filtering. Module 3 introduces the family of algorithms called Content Based Recommender Systems. In particular, Content Based Filtering and Users Based Filtering. It explains how to use the Item Content Matrix to compute recommendations, using some tricks for computing the similarity between items. Would you like to know what a chapter of this module talks about?

Table B.12: Use Case 12: Description of a video

Input	What does the third video introduce?
Intent	#Item_Module_Description
Entities	@cardinality:3 @video:video 3 @reference_role:introduction
Dialog Node	120 – Video Description, no Module
Output in case no module is found in the context of the conversation	Which module are you interested in?
Output in case a module is found in the context	We were talking about module 3. Do you want video 3?

Table B.13: Use Case 13: Definition without specified entities

Input	Can you tell me a definition?
Intent	#Definition
Entities	
Dialog Node	146 – Definition
Output in case no entity is found in the context of the conversation	What should I search definitions for?
Output in case an entity is found in the context	Do you want to know the definition of Item?

Table B.14: Use Case 14: Next event

Input	And the next?
Intent	#Next
Entities	
Dialog Node	132 - Next
Output in case the conversation was about exams	The next exam is on 2019-1-10 11:30:00. The room is not available yet.
Output in case the conversation was about exams and an exam date is found in the context	You recently asked about the exam on 2019-1-10. The following exam is on 2019-2-5 11:30:00. The room is not available yet.
Output in case the conversation was about lectures	The next confirmed lecture will be on 2018-11-12 from 10:15 to 13:15 in room E.G.6. The teacher will be Cremonesi Paolo. It will be a theory lesson. The topic will be: Collaborative Filtering (CF): Machine Learning approaches to CF: Slim.
Output in case the conversation was about lectures and a lecture date is found in the context	You recently asked about the lecture of 2018-11-12. The next lecture will be on 2018-11-14 from 12:15 to 14:15 in room D.0.1. The teacher will be Cremonesi Paolo. It will be a theory lesson.
Output in case the conversation was about deadlines of the challenge	The next deadline will be on 2018-11-15 23:59:00. Good luck!
Output in case the conversation was about deadlines of the challenge and a deadline date is found in the context	You recently asked about the deadline on 2018-11-15. The next deadline will be on 2018-12-1 at 23:59:00.