Chatbots: On Demand Creation of Conversational Agents

Luís Filipe Amaral Pinhanços dos Santos

Thesis to obtain the Master of Science Degree in
Engenharia Informática e de Computadores

Supervisor: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur

Examination Committee

Chairperson: Prof. Mário Jorge Costa Gaspar da Silva
Supervisor: Prof. Maria Luísa Torres Ribeiro Marques da Silva Coheur
Members of the Committee: Prof. João Miguel de Sousa de Assis Dias

October 2015
Acknowledgments

There is no easy way to start to acknowledge all the people that supported me through the long years that this course took. Many came and went but the oldest core is my family.

I would like to first mention the person that supported me by giving me space enough to grow up by myself. Through lows and highs my mother was an always present and calm presence through any adversities I may had encountered.

To my father, who always encouraged me to pursuit what I like, insisting that is never too late to change. A change of course was not necessary, I just needed to put my foot on the gas.

To my brother, who gave me advice through it all. The fault is mine that I only started listening later in life, but better late than never.

To Professora Luísa Coheur, I know I can be a pain in the neck with my relaxed mood but it was a nice ride despite the slow motion speed of progression.

To my friend, the ones that are side by side with me in all good and bad moments. I feel lucky to have you all in my life to relax and keep my head clear of most worries.

And saving the best for last, my best friend. My companion. My safe port. Could not have done it without you, in all honesty. You pushed me hard when I was starting to slow down and kept showing me the way. We have been laughing and loving through the past 8 years but I want many more. Thank you, Cátia.
Abstract

The Instituto Superior Técnico’s Spoken Language Systems Laboratory (L2F) group has a variety of dialogue systems that would be best explored if fit together in a single software engineering project. This new project should be the foundation of a system that creates a conversational agent on demand, unifying several L2F tools.

Previous to any architecture definition or implementation, a study of the system was required. We needed to identify control points, understand the structure and have a global vision of how each system works. We also wanted to observe some systems on the market, trying to acquire know-how as to what to do in our own system.

A first approach to a possible architecture was designed, trying to find a solution that would meet our demands in terms of modularity, customization and integration. Afterwards, we started the implementation process and went over an iterative process of the architecture until we reached our current state.

Ten user configuration scenarios were defined, trying to understand the level of changes necessary to satisfy each specification and evaluate the flexibility of our system. We also did user evaluation on the system, as this is a system designed for end users and not only for programmers of the L2F group.

Keywords

Conversational Agent; Chatbot; Dialogue System;
Resumo

O Laboratório de Sistemas de Língua Falada (L2F) do Instituto Superior Técnico tem uma grande variedade de sistemas de diálogo que seriam melhor exploradas se encaixadas num único projecto. Este novo projecto deverá ser a base de um sistema que cria um agente conversacional on demand, unificando diferentes sistemas do L2F.

Antes de qualquer definição ou implementação de uma arquitectura, é necessário um estudo ao sistema. Precisamos de identificar pontos de controlo, perceber a estrutura e ter uma visão global de como cada sistema funciona. Também quisemos observar alguns dos sistemas existentes no mercado, tentando adquirir o conhecimento do que fazer no nosso sistema.

Numa primeira abordagem, desenhámos uma possível arquitectura, tentando chegar a uma solução que iria de encontro aos nossos requisitos em termos de modularidade, customização e integração. Depois, começámos o processo de implementação e passámos por um processo de iteração na arquitectura até chegarmos ao presente estado.

Definimos dez cenários de configurabilidade, tentando perceber o nível de alterações necessárias para satisfazer cada especificação e avaliar a flexibilidade do nosso sistema. Realizámos ainda uma avaliação com utilizadores, visto que este sistema foi desenhado para utilizadores finais e não apenas para colaboradores do grupo L2F.

Palavras Chave

Agente Conversacional; Chatbot; Sistema de Diálogo;
# Contents

1 Introduction 1

1.1 Objectives ................................................. 3
1.2 Document Organization ........................................ 4

2 Background 5

2.1 Edgar ........................................................... 7
2.1.1 ProgramD .................................................. 8

2.2 Say Something Smart ......................................... 8
2.2.1 How does SSS work? ..................................... 8
2.2.2 The SSS measures ......................................... 9
2.2.2.A Frequency (M1) ....................................... 10
2.2.2.B Answer Similarity to the User Input (M2) ............. 11
2.2.2.C Question Similarity to the User Input (M3) ............ 12
2.2.2.D Time Difference (M4) .................................. 13
2.2.3 Other Configuration options .............................. 14

2.3 Talkpedia ....................................................... 15

2.4 TalkKit .......................................................... 15

2.5 JustChat ........................................................ 16
2.5.1 What can be given as an input to JustChat? ............... 16
2.5.2 What is the expected output in JustChat? .................. 18
2.5.3 But how can I choose the kind of outputs that I want? ... 19
2.5.4 What can I do with the JustChat Filters? .................. 21
2.5.4.A Domain Filter ......................................... 21
2.5.4.B Personal Filter ......................................... 22
2.5.4.C BlackList Filter ....................................... 22

2.6 JustAsk ......................................................... 25

3 Related Work 27

3.1 Pandora Bots .................................................. 29
3.1.1 Basic Pandora Bot ............................................. 29
3.2 Virtual Human Toolkit ........................................... 34
  3.2.1 Architecture .................................................. 34
  3.2.2 NPCEditor ..................................................... 36
    3.2.2.A Utterances .............................................. 37
    3.2.2.B Settings ................................................. 37
    3.2.2.C Classifiers ............................................. 38
    3.2.2.D People .................................................. 38
    3.2.2.E Conversations ......................................... 38
    3.2.2.F Chat .................................................... 40
3.3 Other Related Work .............................................. 40
  3.3.1 Dialog System ............................................... 41
    3.3.1.A OpenDial ............................................... 41
    3.3.1.B TRIPS .................................................. 42
    3.3.1.C Olympus and Ravenclaw ............................... 42
  3.3.2 Data Integration ............................................ 43
4 Architecture ....................................................... 45
  4.1 Introduction ................................................... 47
  4.2 Proposed Architecture ........................................ 47
    4.2.1 Presentation Layer ...................................... 47
    4.2.2 Service Layer ............................................ 48
    4.2.3 Business Logic Layer ................................... 49
    4.2.4 Domain Layer ............................................ 49
  4.3 Implemented Architecture .................................... 49
5 Implementation ...................................................... 53
  5.1 Introduction ................................................... 55
  5.2 Plugins ........................................................ 57
  5.3 Answer Retrieval Algorithm .................................. 58
  5.4 Towards a flexible configuration ............................. 60
    5.4.1 The Brain ................................................ 60
    5.4.2 Plugins .................................................. 60
    5.4.3 Language ................................................ 61
    5.4.4 Other Systems Configurations ........................... 62
# 6 Evaluation

6.1 Introduction ................................................. 65
6.2 Scenarios ......................................................... 65
6.3 User Evaluation ............................................... 69

# 7 Conclusion

7.1 Future Work .................................................... 74

# A Questionário de Avaliação

79
## List of Figures

2.1 JustChat Architecture ................................................................. 16

3.1 Pandora Bots Editor - Info Tab .................................................. 29
3.2 Example interaction with newly created "foodie" chatbot .................. 30
3.3 Example of the "Train" tab ......................................................... 30
3.4 Interaction after the training modification ................................. 31
3.5 Changes in the AIML file after the Train modification ................... 31
3.6 Interaction after the AIML modification ...................................... 32
3.7 Different types of files compose our chatbot ............................... 32
3.8 Examples of Color Set .............................................................. 33
3.9 Examples of Map Set .............................................................. 33
3.10 Examples of Substitutions ......................................................... 33
3.11 Caption for LOF ................................................................. 35
3.12 Caption for LOF ................................................................. 35
3.13 Caption for LOF ................................................................. 36
3.14 NPCEditor Tabbed Functions .................................................... 36
3.15 NPCEditor Utterances Tab ....................................................... 37
3.16 NPCEditor Settings Tab .......................................................... 38
3.17 NPCEditor Classifiers Tab ........................................................ 39
3.18 NPCEditor People Tab ............................................................ 39
3.19 NPCEditor Conversations Tab .................................................. 40
3.20 NPCEditor Dialog Manager Tab ................................................ 40
3.21 NPCEditor Chat Tab Example .................................................. 41

4.1 Proposed Architecture - colors are used to better understand the connections between modules ......................................................... 48
4.2 Implemented Architecture ....................................................... 50
5.1 Answer Retrieval Method
List of Tables

6.1 User Evaluation .................................................. 70
Listings

2.1 Example of an Edgar multilingual pair. ........................................... 7
2.2 Example of an Edgar correspondence. ........................................... 7
2.3 Example of a ProgramD set of rules. ........................................... 8
2.4 SSS Measures. ................................................................. 10
2.5 qaScorers Method .......................................................... 14
2.6 L2R Method ................................................................. 14
2.7 Configuration of SSS Language. .................................................... 14
2.8 Path of SSS Corpus. ........................................................... 15
2.9 Example of template answer when no suitable answers are found. ......... 15
2.10 AIML Example. .............................................................. 18
2.11 Example of Edgar’s format. ...................................................... 18
2.12 Example of Edgar’s format output. ............................................ 19
2.13 Example of AIML output. ..................................................... 19
2.14 Configuration of JustChat output format as Edgar’s format. .................. 20
2.15 Configuration of JustCha output format as AIML. ............................ 20
2.16 JustChat attributes configuration example. ..................................... 20
2.17 JustChat attributes configuration output example. ............................ 20
2.18 Example of Domain Filter in Configuration File. ............................. 21
2.19 Example of Domain Filter. ..................................................... 21
2.20 Example of Corrected Domain Filter. ......................................... 22
2.21 Example of Personal Filter. ..................................................... 22
2.22 Example of Blacklist Filter Config. ............................................ 23
2.23 Example of Blacklist Filter’s Levenshtein Distance Configuration. ........ 23
2.24 Example of Blacklist Filter. .................................................... 24
2.25 Example of Blacklist Filter. .................................................... 24
2.26 Configuration of the Levenshtein Distance. ..................................... 24
2.27 Blacklist Filter Result. ......................................................... 25
Introduction

Contents

1.1 Objectives ................................................................. 3
1.2 Document Organization ................................................ 4
Who would not wish to create his own chatbot, giving it a pronounced personality or a knowledge base that would turn it into an expert in a specific subject? However, creating virtual agents is not as simple as a click of a button.

Although many tools are at our disposal, the ability to personalize a chatbot is not as easy as one would wish. The configuration of features like the looks of the chatbot, knowledge base or expertise is a complex task, not to be taken lightly. Although many available tools allow users to create and develop a chatbot, with or without visual components, the level of configuration is focused mainly on the domain level. We felt that there was not enough configuration manipulation, within these tools, that would allow the user to have a more complex control on the decision process of the answer. We could define, either with Artificial Intelligence Markup Language (AIML)\(^1\) or direct trigger-answer modules, what would be the domain of the chatbot but there is not an ability to define the control points of the answer retrieval method. Each approach has its own way to get an answer but there is not any system that conjugates different tools into a single one.

The main focus of our project is how to approach this complexity, but taking into account dialogue systems tools developed in the L2F/INESC-ID group\(^2\). For this purpose, there is a need to study, understand and play with the tools that are intended to be integrated, building the structure that will allow to create a conversational agent that agglomerates the existing tools in a single system. Our goal is to guarantee that there is a foundation to integrate the studied systems, creating an answer retrieval method that will consider all the plugged in modules. As the configuration issue is one of our highest concerns, we will try to create an architectural approach that gives the user most of the control on how the system will work.

This project has to be a system with a stable main module, in which all others will connect. With this step concluded, we need to create an answer retrieval method that will follow the methodologies implemented in the studied L2F tools, keeping a decision path that will be flexible to a variety of configuration combinations that can be defined by the user.

As the study and adaptation of the code increases, so does the complexity of the changes that have to be done to the tools that were intended to be integrated. As these tools are not prepared to work in a single environment, most of the functional code has to be deeply understood so we can know what to change without totally changing the paradigm, methodology or purpose for which each tool was created.

### 1.1 Objectives

This project has the main objective of building a system to unify existing L2F tools into a single dialogue system. This system has to be able to withstand a range of configuration that we could not

---

2. https://www.l2f.inesc-id.pt
encounter in any other studied system.

The project will integrate, in this stage, four different L2F tools: Edgar, Say Something Smart (SSS), Talkpedia and TalkIt. We integrate these systems to potentiate the tools’ best features and include them in a unique system.

With this in mind, the following objectives need to be completed in order to achieve the main objective:

• Study and understand the functional aspects of some L2F tools that can deal with user interactions.

• Create an architecture that will conjugate different L2F tools.

• Implement the architecture.

• Create a system that allows the user to configure how the tools are used.

• Evaluate the implemented system.

### 1.2 Document Organization

In Chapter 2, we explore the L2F tools to be included in this project.

In Chapter 3, we describe and give examples of two chatbot creating systems, that we studied, in order to understand what could be important or not to our system.

After the study of these systems, we proposed an architecture that would meet the demands and objectives of our project in Chapter 4. How we implemented our system is described in Chapter 5 and how we evaluated its success is in Chapter 6.

In Chapter 7 we conclude the developed work in this project and point out what can be done as future work.
2 Background

Contents

2.1 Edgar ......................................................... 7
2.2 Say Something Smart ........................................ 8
2.3 Talkpedia .................................................... 15
2.4 TalKit ......................................................... 15
2.5 JustChat ...................................................... 16
2.6 JustAsk ....................................................... 25
In this chapter, we will describe different in-house tools, namely Edgar, Say Something Smart (SSS), Talkpedia, TalkIt, JustChat and JustAsk. These were the studied tools that would fit the profile required to be integrated into our system. We will give an overall approach on how the tools work and what we can do with them.

2.1 Edgar

One of the tools in the L2F group is Edgar, the butler of the Monserrate Palace in Sintra [1]. This is a conversational agent with tutoring goals, built to answer the questions of the ever inquiring visitors of the palace. Edgar may receive written or spoken inputs, outputting his answer in both forms. Received input can be perceived and answered in Portuguese, English or Spanish.

Edgar has an in-domain knowledge base, created thoroughly by hand in an XML format, that is capable of answering to direct triggers in each of the mentioned languages. This is possible by having multilingual pairs, of different paraphrases, of question and its possible answer. An example can be seen below.

Listing 2.1: Example of an Edgar multilingual pair.

```xml
<questions>
  <q en="Who are you?" es="Quien es usted?">Quem e voce?</q>
</questions>

<answers>
  <a en="My name is Edgar Smith." es="Me llamo Edgar Smith." emotion="friendly"
      intensity="100">Chamo-me Edgar Smith.</a>
</answers>
```

As we can see, this format is already contemplating attributes such as emotion and intensity, so that when a GUI interface is built, the graphical behavioural features of the chatbot can be set in motion.

These knowledge base can be extended with equivalences between words so that different triggers can be contextualized and it could lead to the same answer. Below is an example of what a correspondence could look like.

Listing 2.2: Example of an Edgar correspondence.

```xml
<equiv>Francis Cook/Viscount Monserrate/Viscount de Monserrate/Mister Cook</equiv>
```

Although Edgar is a chatbot created to mainly answer questions about the Monserrate Palace, it has also some small talk and facts in his knowledge base. But this was not enough and Edgar has the ability to use a tool created to work with one of the most popular chatbot programming languages, Artificial Intelligence Markup Language (AIML)

1http://www.alicebot.org/aiml.html
2.1.1 ProgramD

The AIML interpreter integrated in Edgar was ProgramD, an open source AIML bot system created for Alicebot\(^2\). This system is used to deal with different subjects such as slang, cinema or compliments, but it can be extended to deal with any kind of subject, as long as it is defined with the AIML set of rules. The syntax of AIML will be in Section 2.5.2.

Listing 2.3: Example of a ProgramD set of rules.

```xml
1 <category>
2   <pattern>sabes imensas coisas</pattern>
3   <template>
4     <random>
5       <li>Obrigado. Fico muito contente.</li>
6       <li>Muito obrigada pelas suas palavras.</li>
7       <li>Muito agradecido.</li>
8     </random>
9   </template>
10 </category>
```

2.2 Say Something Smart

Say Something Smart (SSS) is a tool created to deal with Out-Of-Domain (OOD) interactions, that are out of the domain of the chatbot knowledge base, using movie subtitles as corpora (latin plural of corpus) \(^2\) \(^3\). For a large knowledge base this can lead to a huge number of interactions, so SSS was integrated with Lucene\(^3\), an information retrieval library, that is capable of text indexing and fast searching among a huge amount of data.

2.2.1 How does SSS work?

For each interaction received in the SSS dialog system, Lucene will tokenize it and compare it to the triggers that are on the corpus, giving different relevance to each, and creating an organized list with the best matched trigger-answer pairs. To filter this output given by Lucene, which consists of the best X matches, with X being a number configured by the developer, SSS can be configured to use an algorithm that combines both the Jaccard and the Overlap algorithm, or any of these separately, which will compute the similarities between the user input and the trigger-answer pairs returned by Lucene.

The input files has several identifiers, such as the SubID, DialogID and Diff. The SubID is the unique identifier to a single subtitle file so that it is possible to know where the answer came from and the DialogID is the identifier of the number of the interaction in that said file, while the Diff is the time difference between two consecutive sentences, a measure that will be significant to understand that if the sentences are close in time then they may be related. An example of what could be found in the knowledge base is the following.

---

\(^2\)http://www.alicebot.org/

\(^3\)http://lucene.apache.org/core/
SSS will take the best matches from Lucene, trying to find out what is the best answer with its own algorithm. For this, SSS uses four different measures, with different weights, that combined are supposed to give the best possible answer.

As this is based only in movie subtitle interactions, not every answer is appropriate or even makes sense.

### 2.2.2 The SSS measures

SSS applies four measures which are used to discover the best possible answer given by Lucene. Each measure has its different weight and can be combined to give an organized list of twenty answers with the best possible values combined. These weights can be configured in the config.xml file located in the "config" folder inside the "resources" directory. It should be noted that it is not mandatory to use the four measures at the same time, but at least one should be active. The sum of all four weights should always be 100 per cent.

The sentence with the higher score will be the one that will be given as an answer to the user interaction.

Below, an example of what the measures in the Config file could look like.
Listing 2.4: SSS Measures.

```xml
<qaScorers>
  <qaScorer name="AnswerFrequency" weight="30"/>
  <qaScorer name="AnswerSimilarityToUserQuestion" weight="5"/>
  <qaScorer name="QuestionSimilarityToUserQuestion" weight="55"/>
  <qaScorer name="SimpleTimeDifference" weight="10"/>
</qaScorers>
```

2.2.2.A Frequency (M1)

As we can see in the code above, the Answer Frequency is the first measure in the XML file. This measure will give a higher value to the answer that is most common among the possible answers, in order to give some value to the redundancy found in the corpus. This value has can be a minimum of 0 or a maximum of 1.

The following interactions are an example of the described measure, for the question "Como estás?".

Como estás.
Bem.
Como estás?
Bem...
Como estás?
Estou Bem!

The answer "Bem" will have a higher value due to being more frequent. In the next example, only this measure was accounted and the result was as follow.

Normalized question: como estas
Retrieving Lucene results...
Retrieving QA’s from database...
Scoring the QA’s...
Due to giving a higher score to the most frequent answer, SSS assumes that if an answer is more frequent to a given question then it has a higher probability of being the right answer, making this measure a very important one.

2.2.2.B Answer Similarity to the User Input (M2)

One of the many possible ways to face the problem of giving a viable answer is considering that if an answer has many similarities to the question, then it is highly probable that it will be a good answer. Consider the example for the question “Como estás?”.

T - Como estás?
A - Como estás, doc?
M1 score - 0.09335764272516117
M2 score - 0.3333333333333333
M3 score - 1.0
M4 score - 0.8796
Final Score - 0.3333333333333333

T - Como estás?
A - Hei, como estás? Como anda o teu mundo?
M1 score - 0.0832645997771686
M2 score - 0.26666666666666666
M3 score - 1.0
M4 score - 0.0
Final Score - 0.26666666666666666
Even though the score is not very high, this is a measure that seems to give good and logical answers. This example also shows how difficult the task is.

2.2.2.C Question Similarity to the User Input (M3)

If it can be assumed that an answer could have some similarities to a question, then it is also correct to think that if there is a question in the knowledge base that is very similar to the user interaction, we can assume that any answer given to that question could also be a possible answer to the input given by the user. Considering the question “Como estás?” the following example illustrates the best matched trigger-answer pair returned by Lucene.

1
T - Como estás?
A - Vou-me aguentando.
M1 score - 0.0
M2 score - 0.0
M3 score - 1.0
M4 score - 0.7398
Final Score - 1.0

As the high score shows, this measure is very interesting and we can find a logical reasoning for the answer to the question “Como estás?”. In this case and because it is a frequent question, even the last of the 20 Lucene returned answers has a maximum score, as we show below, demonstrating that this a measure that should be highly taken into account when balancing the 4 different measures.
2.2.2.D  Time Difference (M4)

The last measure will consider the time difference between the found trigger and a possible answer, so that when a match is made in the trigger, the closest following answer could be related to the given subject and maybe a possible match.

T - Como estás.
A - Bem.
M1 score - 1.0
M2 score - 0.0
M3 score - 1.0
M4 score - 1.0
Final Score - 1.0

This is the kind of measure that does not make much sense when working alone, and better results could be achieved if the measure with the similarities between the question and the user input is also used.

With the objective of finding the best possible balance between all of the measures, it would not make much sense to use the measures in a solo base. This can not be achieved without a fair amount of trial and error in the conjugation of the different measures.
2.2.3 Other Configuration options

The weights of the measures are not the only configurable subject in the SSS, even though they are the most significant for the output. Two possible evaluations methods can be chosen. All the previous interactions examples were with the Question and Answer (QA) scorer method turned on, but a Learning To Rank (L2R) method is also available. To enable this method, it is only needed to go to the Configuration file (config.xml) and substitute the line

Listing 2.5: qaScorers Method
1 <evaluationChosen name="qaScorers"/>

for the following

Listing 2.6: L2R Method
1 <evaluationChosen name="l2r"/>

which will evaluate the Lucene output with a L2R method.

The SSS also uses different normalization techniques that will deal with some problems related with contractions, upper and lower case and punctuation. Most importantly, it uses a lemmatizer, that will take the different words and reduce it to their lemmas, so they can be evaluated as a single item, instead of being considered as different ones.

The remaining configurations are the simpler, as all that is left to configure is a few paths and the answers that can be given if everything else fails.

There is the possibility of the program recognizing both English and Portuguese languages, with only a couple of modifications needed for it to work.

Listing 2.7: Configuration of SSS Language.
1 <!english or portuguese-->
2 <language>portuguese</language>
3 <stopWordsLocation>./resources/stopwords/portuguese\textunderscore stop.txt</stopWordsLocation>
4 <lucene>
5 <indexPath>./resources/luceneIndexes</indexPath>

As we can see, from this example of the XML configuration file, we can easily indicate whether we want SSS in Portuguese or English, changing the "language" attribute from one to the other, and redirecting some folders. For instance, where the corresponding stop words are or the paths for the Corpus and Log files. The Lucene indexes will automatically be retrieved using the attribute "language" to go to the wanted folder, which makes it modular in case we want to add other languages.

SSS could not work without a corpus, so we need to define where is the folder that has the corpus for the language that it is needed, only having to change the folder in the "corpusPath" attribute of the configuration file. An example is shown below.

It is also useful if the SSS tool records all of our interactions, so it is easier to analyse our interactions and increase the chance of finding incoherent responses. This log file can be found in the resources folder, or it can be changed, also in the configuration file, simply by changing the directory of the attribute "logPath".

If no answer could be retrieved from Lucene, it is necessary that SSS still interacts with the end user. Several answers can be defined in the "noAnswerFoundMsgs", and SSS will choose one randomly to respond in case no suitable answers are found in process between Lucene and the SSS algorithms. An example of what this answers would look like in the configuration file is shown below.

Listing 2.9: Example of template answer when no suitable answers are found.

```xml
<noAnswerFoundMsgs>
  <msg>I am sorry but I did not understand what you said</msg>
  <msg>I do not know how to answer that</msg>
</noAnswerFoundMsgs>
```

2.3 Talkpedia

Talkpedia was created with the purpose of returning appropriate answers to Out-of-Domain (OOD) interactions. When Talkpedia receives an OOD interaction it will create a list of possible search terms, namely the nouns or expressions. The identification of the terms class is done by using a parametrized Portuguese TreeTagger. After the list is created, the terms will be prioritised so that the articles retrieved with higher priority terms are more likely to contain an appropriate answer.

Then Talkpedia will retrieve articles from Wikipedia, based on queries created with the terms of the list. If any answer is found, the result is presented in sets of pairs article-terms. With the priorities previously defined, a selection process begins so that one final answer is selected. This final answer will still be modified, with a prefix being added, so that it generates a more realistic answer. For instance, if the user interaction is "How is the Pope doing?", Talkpedia's answer is "Unfortunately I cannot tell you, but the Pope is the Bishop of Rome and the leader of the worldwide Catholic Church" (example from [4]).

2.4 TalKit

As a continuous project, always trying to integrate new solutions developed in the context of the L2F group, there was a module that we though it would be interesting to include. TalKit [5] is a dialogue sys-
tem based on a dialogue taxonomy used for classifying user interactions. With the resulting taxonomy, on the evaluation of the user interaction, TalKit will choose what is the best module, between SSS and Talkpedia, to answer to the user interaction.

Although there was no interest in studying the external modules that TalKit uses, its taxonomy and decision methodology was interesting to observe as it can give us leads on what path to follow in our architecture and even in our implementation, as it already has different modules implemented.

We focused our approach to TalKit in how we could change the classifier so it would best fit our needs, and not on how we could train the existing classifier to new data. The classifier is a useful tool when thinking of creating an answer retrieval method, as it can identify particularities that can help us choose the path that could lead to a more plausible answer.

2.5 JustChat

JustChat is a tool that takes an untreated Corpus (structured set of texts) and transforms it accordingly to the existing filters (Section 2.1.4), having the possibility to output it in AIML or Edgar’s format. These formats are described in Section 2.1.2.

![JustChat Architecture](image)

**Figure 2.1: JustChat Architecture**

2.5.1 What can be given as an input to JustChat?

JustChat has the ability to take two types of text inputs, the dialog and the qa (as in Question and Answer). The files that we want to process should be in the input folder.

The “dialog” option takes any two consecutive sentences, assuming the first one is the trigger (because not all interaction need to start with a question), and returning the second one as the answer, that is, each sentence works as a trigger and also as an answer. This is done in a chain way to simulate a
conversation flux that can be illustrated by the following example of how an input file would be processed.

Original input:
Hello, how are you?
I am fine and you?
Could not be better! Just got back from some holidays in Hawaii.
Really? How lucky!

This input will be interpreted in two consecutive sentences, like shown below, where the first line is the trigger and the second is the answer.
Hello, how are you?
I am fine and you?

I am fine and you?
Could not be better! Just got back from some holidays in Hawaii.

Could not be better! Just got back from some holidays in Hawaii.
Really? How lucky!

For the "qa" option, the input format should be in pairs, being the first sentence the question asked and the second the answer given by the chatbot. An example of such an interaction is the following pairs, where the first line is the trigger and the second line is the answer.

WHAT IS KARATE
Japanese art of self-defense.

WHAT IS KIND
Sympathetic, friendly, gentle, tender-hearted, generous, etc.

WHAT IS KORN
A band that started a riot at Woodstock 2.
2.5.2 What is the expected output in JustChat?

To be able to better understand what will be discussed in this section, some previous concepts should be taken into account. The Artificial Intelligence Markup Language (AIML)\textsuperscript{7} is an XML-type language used to create conversational agents.

AIML is composed of several different elements, being that categories are the crucial unit of knowledge. Categories, the basic knowledge unit of AIML, consists of two sub-elements, patterns and templates. Patterns represent the rules that will match the user input and templates represent the answers. With only these elements it is already possible to define interactions that the chatbot will be able to understand.

More about the AIML syntax can be found in the Alicebot website\textsuperscript{8}. A simple AIML example is the following:

```
Listing 2.10: AIML Example.

1  <category>
2    <pattern>WHAT IS YOUR NAME</pattern>
3    <template>My name is John.</template>
4  </category>
```

When the sentence "What is your name" is an exact match to the user input, the chatbot will answer "My name is John."

Other fundamental notion that is important to understand, specially for the JustChat tool (Section 2.1), is what we will call the Edgar’s (the butler of the Monserrate Palace in Sintra \cite{1}) format, which has also an XML-like language, but built with different elements and characteristics from AIML. In Edgar’s format we will define a group of questions and then the group of corresponding answers.

The previous interaction example would look as follows, in Edgar’s format:

```
Listing 2.11: Example of Edgar’s format.

1  <questions>
2    <q>WHAT IS YOUR NAME</q>
3  </questions>
4  <answers>
5    <a>My name is John.</a>
6  </answers>
```

Two types of outputs can be given by JustChat, depending on what is the output intended to. As JustChat was a tool created to work with Edgar, Edgar’s format is one of the two possibilities. JustChat then creates an output file, named after the input file that was given, and fills it with the results of the parser, in the required output format. This can be seen with Edgar’s format in the example below, using the interactions previously described for the "qa" function.

\footnotesize\textsuperscript{7}http://www.alicebot.org/aiml.html
\footnotesize\textsuperscript{8}http://www.alicebot.org/documentation/
The second type of output uses AIML, that is quite often used to create chatbots, as most AIML interpreters are free or open sourced. This is a different kind of XML-like language, with different elements from Edgar's format, as previously seen. Below is an example of the AIML output, the same used to demonstrate Edgar's format.

Listing 2.13: Example of AIML output.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<aiml>
  <category>
    <pattern>WHAT IS KARATE</pattern>
    <template>Japanese art of self-defense.</template>
  </category>
  <category>
    <pattern>WHAT IS KIND</pattern>
    <template>Sympathetic, friendly, gentle, tender-hearted, generous, etc.</template>
  </category>
  <category>
    <pattern>WHAT IS KORN</pattern>
    <template>A band that started a riot at Woodstock 2.</template>
  </category>
</aiml>
```

The results of interpreting the input files are then saved, with the same name, in the "results" folder. For example, being the input file named "input.txt" the consequent output would be a file named "input.txt".

### 2.5.3 But how can I choose the kind of outputs that I want?

All the important aspects of this tool can be configured in the config.xml file, which is structured for the user to be able to define what kind of input he is giving, what kind of output he wants and what kind
of filters he wants to use.

The "output-format" can be in either format, "edgar" or "aiml", as previously explained, but can not be both at the same time.

Listing 2.14: Configuration of JustChat output format as Edgar's format.

```
1 <output-format>edgar</output-format>
```

or

Listing 2.15: Configuration of JustChat output format as AIML.

```
1 <output-format>aiml</output-format>
```

We can configure the "input" type, which can be treated as "qa" or "dialog", as previously described in Section 2.1.1. That is not the only configurable aspect of how JustChat will treat our raw Corpus. In case the output is in the Edgar's format, we can also show other two flags in the output, namely the "id" and the "trustworthy" flag.

The "id" flag will show the id attribute of the sentence, which is its file name and line number, making it possible to trace the origin of the interaction just with this data. As for the "trustworthy" flag, this allows to indicate the reliability of the source.

Both these flags can be set to "true", "false" or just be absent, in which case no new information will be displayed in the results.

Listing 2.16: JustChat attributes configuration example.

```
1 <file type="qa" id="true" trustworthy="true">input</file>
```

This configuration will give the following result (for two different input files, named, for instance, "input.txt" and "food.txt").

Listing 2.17: JustChat attributes configuration output example.

```
1 <qa>
2   <questions>
3     <q id="input - line: 1445" trustworthy="true">WHAT IS KARATE</q>
4   </questions>
5   <answers>
6     <a id="input - line: 1446" trustworthy="true">Japanese art of self-defense.</a>
7   </answers>
8 </qa>
9 <qa>
10   <questions>
11     <q id="food - line: 109" trustworthy="true">WHAT DOES ELECTRICITY TASTE LIKE</q>
12   </questions>
13   <answers>
14     <a id="food - line: 110" trustworthy="true">Like your favorite food tastes to you.</a>
15   </answers>
16 </qa>
```
2.5.4 What can I do with the JustChat Filters?

The relevant purpose for the use of the JustChat tool is the ability to filter a knowledge base, using any combination of the three existing filters. The filters are used to resolve domain conflicts, such as overlapped information between one previously existing knowledge base and the newly created (the output files from JustChat).

Each one of this filters have their own objective as will be shown in the following sections.

2.5.4.A Domain Filter

The Domain Filter allows us to grow the existing knowledge base, without having any kind of conflicts and incorrect information. This is done using the already existing Edgar’s matching technique, which combines both Jaccard\(^9\) and Overlap algorithms. One example of the kind of collision that may occur, without this filter, is the “What is your name?” question, if this sentence is already in the original corpus. With the merge of various knowledge bases, the chatbot could give an incorrect answer that would contradict his original corpus, assuming that the answer to this question existed.

In the Config file, this filter can be used just by simply adding the following line:

```xml
Listing 2.18: Example of Domain Filter in Configuration File.
<domain value="true" database="filters/domain" correct-answers="true"/>
```

The flag "value" should always be true, if the domain filter is supposed to be used, or false, otherwise. The "database" attribute indicates the path to the original knowledge base. There is another flag, the "correct-answers" that will indicate whether or not we want the chatbot to answer based on the original knowledge base or based in the new one. Below is a demonstration of one of the conflict using the "correct-answers", as we forced the answer “Pie is a type of chart or graph!” into the new corpus. JustChat detected it as a conflict answer for the question "WHAT IS PIE", that had an original answer of "Pie is food with a baked crust."

```xml
Listing 2.19: Example of Domain Filter.
<qa>
  <q>WHAT IS PIE</q>
  <a>Pie is a type of chart or graph!</a>
</qa>
<additional-info>matched with: [TextInteraction text=WHAT IS PIE , wordSet=[PIE, WHAT, IS], wordList=[WHAT, IS, PIE]]</additional-info>
```

This is the example of the result of the application of the Domain Filter, which creates a file located in a sub-folder of the results, with the additional showing attribute "additional-info" that will indicate what was the conflict. After this conflict is detected and the question is filtered, JustChat will create a new file, in the same sub-directory mentioned above that will show only the corrected interactions, as follows.

---
\(^9\)http://en.wikipedia.org/wiki/Jaccard_index
Listing 2.20: Example of Corrected Domain Filter.

```xml
<qa>
  <q id="qa" trustworthy="true">WHAT IS PIE</q>
  <a id="qa" trustworthy="true">Pie is food with a baked crust.</a>
</qa>
```

So, the answer of JustChat, using the Domain Filter, to the question "What is Pie?" is "Pie is food with a baked crust.", instead of the erroneous "Pie is a type of chart or graph!" that we inserted in the new files just to create a conflict.

2.5.4.B Personal Filter

The Personal Filter is the one that will help the existing chatbot to answer unprepared personal questions, trying to filter the interactions that may contain personal information, which should be configurable to each chatbot. This filter will try to classify each question as Personal or Impersonal, using a Support Vector Machine (SVM)\(^\text{10}\), developed for another L2F tool (JustAsk [6]), and a training set made from examples of Personal and Impersonal questions. A few examples taken from this training corpus can be seen below.

PERSONAL What is your favorite kind of ethnic food?
PERSONAL Are you named after ancestors that are already dead?
IMPERSONAL What kind of creature is a coot?

Any resulting conflict, with the Personal Filter, will generate a result file showing what was identified as Personal and is in need of a customization.

The following is an example of something that could be found in the generated file.

Listing 2.21: Example of Personal Filter.

```xml
<corpus type="PersonalQuestionsFilter">
  <qa>
    <q id="food - line: 11" trustworthy="true">HOW MUCH FOOD DO YOU LIKE TO EAT</q>
    <a id="food - line: 12" trustworthy="true">A lot!!</a>
  </qa>
  <qa>
    <q id="input - line: 837" trustworthy="true">DO YOU LOVE FOOD</q>
    <a id="input - line: 838" trustworthy="true">Who doesn't?</a>
  </qa>
</corpus>
```

2.5.4.C BlackList Filter

The last available filter, the Blacklist Filter, has the ability to filter some terms or topics that the user wants to highlight.

\(^{10}\)http://www.support-vector-machines.org/
To use this filter, we should set the flag "value" to "true" and specify the source directory, using the attribute "terms" of the configuration file, where the files that contain the terms are. In the specified directory, the developer should create one or more text files with all the wanted terms.

To do the same, but with topics, and its consequent WordNet propagation, the developer should create a text file with the topics that are meant to be highlighted and indicate its full path in the attribute "topics". While many files may be allocated in the specified directory, as the parser will process them all for the attribute "terms", for the "topics" only one file can be allocated and parsed, meaning that we can have more than one file with terms on it, but only one file with topics is allowed. The following example may elucidate this, as "topics" need to have a full path and "terms" do not:

Listing 2.22: Example of Blacklist Filter Config.
```
<blacklist value="true" terms="filters/blacklist" topics="filters/blacklist/topics.txt" />
```

As we can see, we can specify the directory for the files that have the terms we want to highlight, but we must indicate specifically what text file contains the topics.

One other configuration is available for the Blacklist Filter, the Levenshtein Distance. This algorithm highlights not only exact matches of the specified terms and topics, but also words that are similar, based on a number of single-character edits. The configuration file should specify where the XML file is located. The Levenshtein Distance is an optional attribute in the Blacklist Filter.

An example of this kind of XML file is:

Listing 2.23: Example of Blacklist Filter's Levenshtein Distance Configuration.
```
<blacklist>
  <levenshtein>
    <edit>
      <min>2</min>
      <max>4</max>
      <editions>1</editions>
    </edit>
    <edit>
      <min>5</min>
      <editions>3</editions>
    </edit>
  </levenshtein>
</blacklist>
```

This represents the parameters given to JustChat, so it can calculate the Levenshtein Distance between two words, meaning the first part will allow one modification to a word with a minimum of two letters and maximum of four. The second edit allows three single-character modifications to any word with a minimum of five letters.

A practical example is the attempt to highlight any interaction that has the term "illuminati" in it. For this, we only need to write the word "illuminati" in a text file containing terms that we want to highlight. For instance, if the input file has:


http://wordnet.princeton.edu/

---

11 http://wordnet.princeton.edu/
WHAT IS THE ILLUMINATI
A secret organization believed by some to be in control of all governments through a worldwide conspiracy.

This would give the following result:

**Listing 2.24:** Example of Blacklist Filter.

```xml
<qa>
  <q>WHAT IS THE ILLUMINATI</q>
  <a>A secret organization believed by some to be in control of all governments through a worldwide conspiracy.</a>
</qa>

<additional-info>blacklist word/expression in string: "illuminati"</additional-info>
```

A match was found in the produced highlighted terms that exactly matches the word "illuminati".

**Listing 2.25:** Example of Blacklist Filter.

```xml
<qa>
  <q>WHAT IS ILLIMINATUS</q>
  <a>A secret society that has supposedly existed for centuries.</a>
</qa>

<additional-info>blacklist word/expression in string: "illuminatti"</additional-info>
```

As we can see by the difference in the form of the word "Illuminati" (the second has one more "t"), the Levenshtein distance works, and can catch the difference in the words that happened in the last letters of the highlighted term. The same can not be said to characters in the beginning of the word, as the interaction "WHAT IS ILLIMINATUS" was not filtered correctly.

Although the original algorithm is capable of recognising insertions, deletions or substitutions of characters, the implemented one is not fully functional. Trying to catch the term "Illuminati" raises another problem, as the Levenstein Distance is only working for words with the same or more characters, not catching, in this case, the term "Illuminati".

In the Levenstein XML file there are a couple more features that are not entirely clear and do not seem to work. If we look into the code below, we would suppose that we could specify what kind of terms (tokens) or topics (domain) we should restrict the chatbot from answering, having at the same time, the ability to define which replies it should give to this kind of situation.

**Listing 2.26:** Configuration of the Levenstein Distance.

```xml
<questions>
  <tokens>
    <t>economy</t>
    <t>government</t>
    <t>prime minister</t>
  </tokens>
  <domain>politics</domain>
  <replies>
    <r>wouldn’t you prefer to talk about a lighter subject?</r>
    <r>that subject is a little bit boring...</r>
  </replies>
</questions>
```
Although we tried to test the interactions that this code would seem to catch, we did not observe any produced result, leaving us to believe that this part of the code is not working.

In the end, as in all the other filters, a result file will be produced with the conflicts found, containing something like the example below:

```
Listing 2.27: Blacklist Filter Result.
<corpus type="BlacklistFilter">
  <qa>
    <q>WHAT IS THE ILLUMINATI</q>
    <a> A secret organization believed by some to be in control of all governments through a worldwide conspiracy.</a>
  </qa>
  <additional-info>blacklist word/expression in string: "illuminati"</additional-info>
</corpus>
```

Although this filter has potential, as we can restrict or potentiate a certain term or topic, it also seems the one that more work is required to make it fully functional.

### 2.6 JustAsk

JustAsk is an open-domain question answering system, focused on short factoid-like questions but not despising other questions, like definitions. JustAsk has three modules, each one with a specific task.

The Question Interpretation module analyzes and classifies a question. For this, it receives a natural language question, interprets it and returns a structure composed by different types of information extracted from the question, such as named entities or verbs. Then, the Passage Retrieval module will reformulate the question into a query containing only the keywords and excluding stopwords, question words and punctuation. This query will be searched in the available information sources, like Lucene\(^\text{12}\) or Wikipedia\(^\text{13}\), and the retrieved results will be the output of this module.

Finally, the Answer Extraction module receives the output from both previous modules. The process of gathering the final answer is divided in two stages. First, JustAsk uses Regular Expressions, Named Entities and WordNet recognizers as strategies to extract possible candidate answers. After these candidates are gathered, four tasks will be performed to help the decision of what the final answer should be. The candidates go through a process of Normalization, Aggregation, Clustering and Filtering. A final Selection task will choose the highest scored answer to return as an output of JustAsk.

\(^\text{12}\)http://lucene.apache.org/core/  
\(^\text{13}\)http://en.wikipedia.org/
3 Related Work

Contents

3.1 Pandora Bots .......................................................... 29
3.2 Virtual Human Toolkit ........................................... 34
3.3 Other Related Work ................................................. 40
In this section we will describe existing systems to create virtual conversational agents. We studied these specific systems to understand the steps needed for a user to create a chatbot and to understand how their architectures are structured. Besides Pandora Bots\(^1\) and the Virtual Human Toolkit\(^2\), we will briefly describe other tools that may be interesting to our project.

### 3.1 Pandora Bots

Pandora Bots is a software that allows anyone to create a AIML based virtual chatbot, with a proper Application Programming Interface (API) that is suitable for any user to interact and try to configure his own bot as wished. It is easy to create a template bot but the main characteristic of a chatbot is his own personality, which is a little harder, and that must be configured by the botmaster (person who creates the chatbot). It is also possible to publish the created chatbot so it will allow anyone to interact with it.

#### 3.1.1 Basic Pandora Bot

After signing up in the Pandora Bot website, we have the possibility to create our own chatbot, giving it a name and choosing one of more than a hundred different available languages.

The Pandora Bots Editor has four different tabs to help us guide through the process of configuration of the chatbot. In the info tab, that has the name we chose to our chatbot, we can consult the basic information about our chatbot, such as the language of the bot, its description and the amount of interactions (Figure 1).

![Figure 3.1: Pandora Bots Editor - Info Tab](image)

<table>
<thead>
<tr>
<th>Language</th>
<th>Description</th>
<th>Interactions (Today)</th>
<th>Interactions (All Time)</th>
<th>Created On</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>A bot that loves to eat!</td>
<td>3</td>
<td>3</td>
<td>06/11/2014, 11:48:41</td>
<td>Compiled</td>
</tr>
</tbody>
</table>

In here, it is possible to publish the chatbot to the "Clubhouse", where all the published chatbots are, which will let other people interact with it. In the initial stage, without any kind of train or new AIML files, the chatbot will not be able to maintain any kind of conversation with the user. An example of what happened in an interaction with a newly created chatbot can be seen in Figure 2.

No intended chatbot should have this kind of interactions. A basic chatbot template should at least come with some basic interactions functionality, so that the botmaster would not need to worry with basic interactions, being able to focus on the most specific interactions that it would be necessary to transform the chatbot to what the user had in mind.

---

\(^1\) [http://www.pandorabots.com/](http://www.pandorabots.com/)

\(^2\) [https://vhtoolkit.ict.usc.edu/](https://vhtoolkit.ict.usc.edu/)
To configure and create the desired answers of our chatbot, an easy interface is shown in the "Train" tab (Figure 3). Although this interface will not allow to manipulate the AIML language at a long extent, it is a good basic step for users who do not know much of this language, even though the modifications made will only be recognized as an exact match, meaning that any other minimal changes will not be recognized.

The example of Figure 3 shows a possible input in the "Train" tab, which will modify the chatbot AIML and give the chatbot the ability to answer the question "Do you like sushi?", with the answer being defined by the botmaster. This modification will have the following result.

The "Train" tab gives the users the possibility to train the chatbot simply by entering questions that they would like to see answered. This modifications will have a direct impact in the AIML files that constitute the chatbot (Figure 5).

This AIML files, which can be seen in the "Files" tab, allow the botmaster to better control what will be the final result of his chatbot. To be able to fully understand and use the potential of this tool, the botmaster should have a reasonable understanding of the AIML language. With this kind of knowledge, the botmaster will not be restricted to simple interactions, being able to manually develop the chatbot. One example, that could complement the previous example (Figure 4 and 5), is the one where the
botmaster would wish to capture any sentence that has the word "sushi" in it, so the following code would be sufficient to do that.

Listing 3.1: Example of AIML Training.

```xml
1
2 <category>
3 <pattern>\{\} SUSHI \{\} </pattern>
4 <template>Did you say sushi? I love sushi!</template>
5 </category>
```

With this, the chatbot is now capable of answering to any interaction that has the word sushi in it, without forgetting the complete sentence previously given (Figure 6).

Besides being able to modify the learning files, there is also the possibility to upload our own AIML files, as long as they are in an AIML extension (*.aiml). If they are correctly coded they will automatically be part of our chatbot knowledge base, giving us the possibility to augment the dialog ability of the chatbot using previously created files.

Our Pandora Bot chatbot can have six different types of files, each one with its own purpose. The different categories are: AIML, Sets, Maps, System and Substitutions (Figure 7). Besides this feature, we can also load our own files into it, in case we have already experimented with this kind of system and do not want to go by it all over again.

Although AIML files are the "brain" behind our chatbot, other files should not be underestimated, as only with a complete understanding of them the chatbot can be used to its fully potential.

Despite the Sets and Maps tabs being empty when the chatbot is built, their files can be created
and are supposed to aggregate different combinations of words that are meant to be evaluated as one, working as an hyperonym\(^3\). The following example illustrates the possibilities that this feature can use (Figure 8).

The Maps are similar to the Sets, but represent a list of pairs key-value, and are meant to be used with a set. When defining a map, the left value (key) should correspond to the value of the corresponding set (Figure 9). This matching will be configured through the AIML learning file.

In the System tab, the existing files will allow to better configure some nuances like the punctuation used in the Normalization process, which is used in the pre-processing phase of the sentence. We can also define what is the name of the learning file, in case an uploaded file is supposed to be used.

Several files exist in the Substitutions tab. These files will help the normalization process in transforming pronouns, gender and others into something suitable for a possible answer (Figure 10).

---

\(^3\)“A word or phrase whose referents form a set including as a subset the referents of a subordinate term.”
Much more can be done using the Pandora Bot tool just by exploring the features of the AIML language. It is a well designed tool, with a user-friendly interface and a great tutorial for users, regardless of their AIML knowledge, written with adequate language and showing many of examples that will explore the AIML language. This is the kind of user approach that we are interested in implementing into our project, as any system should be accessible to users with any kind of knowledge.
3.2 Virtual Human Toolkit

The Virtual Human Toolkit (VHToolKit) [7] [8] is a Unity4 based tool, created by the Institute for Creative Technology (ICT) of the University of Southern California (USC), that conjugates 12 different modules, each one with its own purpose, and gives the user the ability to create and configure a variety of different virtual humans. The VHToolKit is a good example of the type of architecture that we are interested in constructing for the on demand chatbot, and the system itself is a good example of how our application could work.

The VHToolkit allows the user to fully configure a virtual agent, from the answers it may give, which are configured through the NPCEditor (Section 3.3), to the way the virtual agents body reacts and moves while it is speaking. Every aspect of the VHToolKit is controlled by its own module, giving the user the possibility to individually create and develop various types of scenarios to each created agent. Virtual Humans are very complex, specially those with such a complete structure like the ones created by the VHToolKit, given that everything is configurable and the user is even able to change the renderer used from Unity to Ogre5. But this complexity makes the tool even more desirable for the ones that truly want to create the most complete conversational agent possible, as it has modules that recognise text or speech, process it and returns a spoken and written adequate response.

Most of the modules and tools that the VHToolKit has are useful and an example for what our own application can be, but there are two aspects that are fundamental to this modular application, its Architecture and the NPCEditor, so we will focus on these aspects. Unfortunately, the designed interface is quite confusing and the available tutorials are not very explicit. Only with many trial and error were we capable of achieving a result that fully explores all its potential. This is the type of interface that we do not want in our application, because we intend it to be user-friendly and for every user, trying to make the creation of a virtual agent an easier process for the end-user.

3.2.1 Architecture

One of the most interesting aspects of this tool, is how the different modules are combined into a single architecture (Figure 11 and Figure 12). Although every single module has its own function, even though some are connected (like the Natural Language Understanding module is connected to the Speech Recognizer), this type of architecture makes it easier to change the different layers of the application without having to change it all. For example, if we wished to change the way our agent would behave nonverbally, it would be sufficient to change the NonVerbal Behavior Generator (NVBG) module to a different one, with only the preocupation of making sure the input and output connections are well

---

4https://www.unity3d.com/
5http://www.ogre3d.org/
established. Even though this type of architecture is not a trivial solution it costs less maintaining it in a long term time frame.

The modular architecture and how the modules communicate between them is the most important feature in our project, so we this is an approach that we took into account when we were creating our proposed architecture (Section 4.1).

Figure 3.11: Virtual Human Toolkit Architecture

Figure 3.12: High Level Virtual Human Toolkit Architecture

6https://confluence.ict.usc.edu/display/VHTK/Architecture
7https://confluence.ict.usc.edu/display/VHTK/Architecture
3.2.2 NPCEditor

The NPCEditor [9] [10] [11] [7], integral part of the VHToolKit, is a system that helps building a Natural Language Processing (NLP) component for virtual conversational agents so that it is capable of maintaining a dialog with any user on a limited domain, using a statistical text classification to map an user input to a corresponding response. It has a specific interface designed to make it easier to build the Property List (PList) file that will store all the possible defined interactions with the virtual agent. This PList is built, in the NPCEditor system, by inserting different possibilities of questions that could be posed by the user and assigning them to one or more possible responses, defined by the bot master. This will create a kind of network between all the possible interactions, and if there is a match for the user's input the dialog manager will return the least recently used answer, breaking the tie from the classifier score, in case more than one answer is possible.

![Figure 3.13: NPCEditor High Level Architecture](https://confluence.ict.usc.edu/display/VHTK/Architecture)

The process of creating this system that controls the logic of the interactions is not easy, as NPCEditor is a modular system that includes several individual functions. Each function has its own panel in the Graphical User Interface (GUI) character editor (Figure 14).

![Figure 3.14: NPCEditor Tabbed Functions](https://confluence.ict.usc.edu/display/VHTK/Architecture)
3.2.2.A Utterances

In the "Utterances" tab the user can see the list of "Questions" and "Answers" available. The "Questions" are the interactions, questions or statements, that can be inserted by the user and the system expects to receive and recognize. For each user input interaction, the NPCEditor uses its statistical classifier to compare the input to the "Questions" list. If an exact match or a close match is found, the NPCEditor responds with the highest scoring linked answer. These scores are determined by the botmaster when he connects a "Question" and an "Answer" (Figure 15).

![NPCEditor Utterances Tab](image)

Figure 3.15: NPCEditor Utterances Tab

This function will allow the botmaster to define which interactions will be answered and with which answers, but a lot more can be configured while creating a Virtual Human agent in the NPCEditor.

3.2.2.B Settings

The botmaster can create and edit both labels and categories that can be specified as annotations to be used as a non-lexical feature in classification. This will also provide the botmaster with some help going through the database more effectively while linking questions to answers.

For example, one of these categories will be used as a default script for the dialog manager to handle user input interactions that are off topic. The other categories need to be properly set so they can communicate with the other modules from the Virtual Human creator tool. Examples can be found in "NPCEditor: A Tool for Building Question-Answering Characters" (Section 5) [9].
3.2.2.2.C Classifiers

When the language database is created or modified, the botmaster should train the parameters of the classifier so it can achieve better results. Although some advanced operations are available in the "Classifiers" tab, for most advanced botmasters training the database with the default values is as simple as clicking the "Start Training" button. This functionality generates classifiers that will map the Virtual Agent answer to the user input interaction.

3.2.2.2.D People

This "People" tab allows the botmaster to create new domains and specify its properties. Creating a new "People" will define a new domain, and not an actual character as one would think, to handle in the NPCEditor, which is associated with a different classifier. This domain will work like a state in the dialog manager, therefore being able to establish a tree of domain defined parenthood, meaning that a domain can inherit from a given domain, as long as the botmaster keeps one domain as its root state, which will handle the initial greetings. For example, setting a domain's parent property to "Anybody" means that this domain will inherit the utterances from the domain "Anybody".

3.2.2.2.E Conversations

The botmaster can choose from one of the provided dialog manager options, for example "Scriptable", which is the one used in the existing VHToolKit tutorial and that will create a new tab "Dialog..."
Manager*. This tab contains an initial script, written in Groovy9, for the manager, that can be edited to suit the botmaster needs. No information could be found on the dialog manager options (Classifier, Blackwell or SGT Star) other than that the dialog manager classification system changes in each alternative. According with [10]: “NPCEditor can also support more complex dialogue phenomena, as long as the basic paradigm of selecting preauthored outputs can be maintained. This “advanced” functioning is still rather primitive, however, requiring the designer to program the dialogue manager rather than providing graphical support and easy to follow design guidelines.” As it has been said, the dialogue manager interface is still in a primitive state so we could not fully test it, and so, did not fully understand its purpose.

9http://beta.groovy-lang.org/
3.2.2. F Chat

The "Chat" tab will allow the botmaster to test the interaction with the created classifier and observe how it classifies the available answers and its corresponding score. In the left vertical panel it is shown how the "Questions" were ranked, in the case it does not find an exact match, which will highlight the preponderant words in the "Answers" database.

3.3 Other Related Work

There are many more systems for the creation of conversational agents and we could not explore them all while studying the state of the art for this project. The systems explored in this section were dialog system that distinguished themselves over time or were recently released. Unlike the previous
described systems, there was not a full attempt of working with the tools that will be described in this section. We will briefly discuss other systems that we studied, without getting in too much detail and exploring more their architecture than their functionality.

3.3.1 Dialog System

3.3.1.A OpenDial

OpenDial\(^{10}\) is a toolkit to create conversational systems, independent of the domain. We did not test OpenDial, but instead gave an overview of the aspects that we found could be more interesting to our project.

What was most captivating about the OpenDial system was how its architecture was described as a blackboard architecture\(^{11}\). This is an interesting idea for how our knowledge base could work, constantly being updated from other modules and knowledge sources. With tools like Talkpedia (Section 2.3.2) implemented in our project this type of architecture could make sense, as our knowledge base could be updated with the retrieved answers from Talkpedia.

Another interesting aspect of the OpenDial architecture is the ability to integrate external modules. This is done using a Java\(^{12}\) module implementation defined by the developers. What would be ideal

---

\(^{10}\)https://code.google.com/p/opendial/

\(^{11}\)A blackboard architecture means that a knowledge base exists and it iteratively updated by a diverse group of knowledge sources, in search of a solution.

\(^{12}\)https://www.oracle.com/java/index.html
to include in our project would be a Plug-and-Play system but a solution like this one should not be overlooked.

As previously seen systems, OpenDial also works with an XML-like language that specifies a domain and a set of models and rules. As most of the studied L2F tools that use an XML-like language (Section 2.1 and Section 2.2), this could be an interesting approach to follow to find a common ground in the communication between the different modules. Full specification of the XML-like language can be found in the Dialogue Domains section of the OpenDial website\(^\text{13}\).

### 3.3.1.B TRIPS

TRIPS\(^\text{14}\) [12] [13] is an Interactive Planning System that interacts with a user. TRIPS has three major functionalities: Interpretation of the user input, Behavior of the agent as it decides the answer and Generation of the agent's intentions to the user. The complex network architecture\(^\text{15}\) is not very modular despite of the separation between the basic dialogue system components and the domain-specific components. Even though each module has a well defined role in the overall architecture, being a complex network architecture makes the implementation of external modules harder as there is a need to maintain the communication between several modules.

All communication is made using KQML messages [14] and each TRIPS component is connected to a Hub that is responsible for handling the message system. This is an interesting approach to the communication problem that we have in our project and a possibility that should not be discarded.

### 3.3.1.C Olympus and Ravenclaw

Olympus\(^\text{16}\) is a framework to build spoken dialog systems. Its architecture is based on a pipeline approach, with the messages between modules being coordinated by a Hub. Two modules handle the speech recognition and translation. It then reaches the Dialogue Manager (Ravenclaw\(^\text{17}\)) that will decide the plan list to be executed. It does so by creating a task tree. The tasks available are created by the system's developer. In order to complete these tasks, Ravenclaw can communicate with a Database layer to help it reach the solution for the interaction. Ravenclaw then delegates the solution to two modules that will handle the generation and synthetization of the solution to the end user.

The use of a Hub as a central tool will agilize the communication process. A central communication tool that finds a common ground to the different types of inputs and outputs is an approach that could be interesting for the development of our work.

\(^{13}\)https://code.google.com/p/opendial/wiki/DialogueDomains
\(^{14}\)https://www.cs.rochester.edu/research/trips/
\(^{15}\)https://www.cs.rochester.edu/research/clisd/projects/trips/architecture/
\(^{16}\)http://wiki.speech.cs.cmu.edu/olympus/index.php/Olympus
\(^{17}\)http://wiki.speech.cs.cmu.edu/olympus/index.php/RavenClaw
3.3.2 Data Integration

There are some systems in the market, like Talend\textsuperscript{18} or Teradata\textsuperscript{19}, that have focused their product in the problem that is Data Integration\textsuperscript{20}. Data Integration will be one of the issues that will need to be solved so that the different modules can communicate between them. As we are trying to build a modular architecture that can implement external modules, independent of their input and output format, it is harder to find a system that meets our requirements. One possible solution for our project is the creation of a Translation Manager module that will work only for the available modules. This Translation Manager could be based in the solution developed in the systems mentioned, at the beginning of this Section, where all the data is unified in a single structure that could be a common access point for all modules.

\textsuperscript{18}https://www.talend.com
\textsuperscript{19}http://www.teradata.com/
\textsuperscript{20}Data Integration is the combination of data from different sources.
4

Architecture

Contents

4.1 Introduction ......................................................... 47
4.2 Proposed Architecture ........................................... 47
4.3 Implemented Architecture ............................... 49
4.1 Introduction

As we first approached this project, there was an intention to build an architecture that could support future new features with a plug-and-play methodology. The design of such an architecture would need to be modular and with a well structured layered design. The relations between each layer would have to contemplate the interaction between modules, but keeping the layers and their purpose well defined.

As this project was intended to merge several systems into a single one, the architecture solution first designed was thought believing that these projects could support such an approach with standardized changes to the code, that could be replied between each new system. An utopic view of the solution this was. The longer we meddled around with the code, the harder this kind of implementation looked feasible. Our proposed architecture seemed further away from reality.

We will now present and explain the proposed architecture and the implemented architecture.

4.2 Proposed Architecture

In order to achieve the goal of maintaining a high modularity architecture, a four-layer architecture was proposed, composed of the Presentation Layer, Service Layer, Business Logic Layer and Domain Layer, as shown in Figure 4.1. This kind of architecture would allow the developer to replace one module, or layer, without affecting the full functionality of the program, as long as the correct communication between layers and modules were kept.

4.2.1 Presentation Layer

This layer would be responsible for how the system would be presented to the user. For this effect, five modules would have different functions, corresponding to five different possible functionalities of the chatbot creator: Interaction with the user, Tools Configuration, Visual Configuration of the chatbot, creation of Trigger/Answer interactions and an Advanced Configuration module.

The User Interaction interface module would be responsible for acquiring and dispatching all the input of the user and how the user would interact with the application.

There were two configuration modules, the configuration of the used tools and the configuration of the visual aspect of the chatbot. The user would be able to configure the tools that the chatbot creator uses to find a proper answer to a given interaction, meaning that most features of systems like JustChat and SSS would be configurable. The user could choose which tools would be used or not and give each tool a confidence value. That would define what the answer would be, accordingly to the confidence the user has in each tool. In the visual configuration, the user would be able to define how the chatbot
would look, from the existing models, or loading his own. Then, the user would be able to save this configuration, so it could be reusable in the future.

In the Trigger and Answer module, the user could define and link specific triggers and answers that the chatbot would be able to receive and return, respectively, much like it is done in the NPCEditor (Section 3.2.2).

An Advanced Configuration module would also be available, which would capacitate the user to load external modules to the Business Logic Layer (Section 4.2.3), like the ability to load different files in the Pandora Bots (Section 3.1), and to manage chained connections between different modules from the Business Logic Layer.

### 4.2.2 Service Layer

The Service Layer would deal with the data transmitted by the Presentation Layer, being responsible to process this data. IT could either store it in the Domain Layer, in case of configurations that the user can change or the Trigger and Answer domain data, or to communicate and redirect the user input to the Business Logic Layer and its corresponding modules (which would be done mostly by a Classifier module). The communication between the Service and the Business Logic layers was not trivial because each module in the Business Logic Layer was prepared to receive different inputs, so a Translation Manager module was needed, to guarantee that a proper input was given to each module in
the Business Logic Layer [15].

To store, register and edit the data, services would be provided by a Registration module, which would directly communicate with the Domain Layer, where the data would be stored persistently.

4.2.3 Business Logic Layer

The Business Logic Layer would feature all the systems used to evaluate the given triggers and search an appropriate answer to it. In addition to the tools described in Section 2, a module to search and evaluate the Trigger and Answer Domain data would be available (T&A Exact Match in Figure 4.1). This module would evaluate the trigger and compare it to the Trigger and Answer Domain data. If a match was found it would then choose one of the available answers.

When all possible answers were found, from all different active components, the Final Evaluation module would evaluate them, and decide what the best answer was, making the decision based on the confidence values defined by the user.

4.2.4 Domain Layer

The Domain Layer represented a Data Storage Layer, where all the relevant data would be stored, for example, all the existing knowledge base used in the Business Logic Layer, such as subtitles for SSS or the Trigger and Answers that can be defined by the user. All the configuration definitions would be stored in this layer in a persistent format so that this type of data would always be available for the remaining layers to access.

4.3 Implemented Architecture

The goal was to have an architecture that would withstand all possible plugins with minimal to none effort on the programming side. But soon into the code exploration, we realized that this was something that would collide with our configurability feature, as it would be necessary for a programmer to guarantee that the WebServices would be established and the usage of each plugin, in terms of their own purpose.

Initially, a 4-tier architecture was thought (in Figure 4.1), with the intention of keeping a defined separation between each layer. This was designed with the purpose to achieve the goal of maintaining a high modularity architecture. As the implementation was progressing, this architecture stopped following the approach that we were trying to achieve, as we chose Edgar to be our main brain. Looking back on different studied architectures (in Chapter 3), we were considering a central HUB to deal with the different messages that could come from the different plugins. Using this paradigm, we though that one
of those systems could work as our central brain, our main agent, dealing with the communication with other plugins, and withstanding the function to receive a user interaction and returning its answer.

This choice was made because we thought that the basis, already built in Edgar, would be a good starting point as to where our plugins would meet. This approach would allow us to take an already established and functional system, and build on it. The growth of our project started with Edgar as our nuclear system and we decided to maintain it, giving up on the 4-tier architecture and merging the Service Layer with the Business Logic Layer.

![Implemented Architecture](image)

**Figure 4.2:** Implemented Architecture

There are two more significant changes to the previously thought architecture. The Service Layer included a Translation Manager to adapt each user interaction to the intended system. Instead, we chose to directly adapt the systems to our needs, as we could not develop a module that would adapt to every different possibility. With Edgar being the central module of our system, the user interaction no longer flows directly on the classifier to know its way. Instead, it receives the interaction from Edgar and reports directly to it. The decision method is implemented in Edgar, so that even if the classifier is not working, the chatbot will still be able to find an answer.

The employment of Edgar as our core module, with the singularity that its module receives a user interaction and returns a final answer, allows us to perceive that the implementation of the Presentation Layer would be similar to the one Edgar already uses, allowing future work to build on this approach and create a new interface, independent of the Business Logic Layer. Every other Business Logic Module, that we implemented as WebService plugins, are directly connected to Edgar. This will keep
our intention of achieving modularity because whatever the GUI that could be used, it will only need to send its information to a single module and wait a response from it.

The previous architecture was expected to have incorporated a direct match methodology between user interaction (or Trigger) and answer. As this functionality is already implemented in Say Something Smart, we found no need to create a tool that would replicate one possible behavior of the said system. Instead, the user has the free will to define the weights of the four possible measures.

The Domain Layer is kept, as it has direct connection with each respective module, meaning that the Domain of each module can only be changed inside said module. As we were integrating the plugins, the thought of separating each domain into a single unified layer would imply major changes to every system without a real gain to it, because if we would want to permanently remove a system, two operations would be required: eliminating the system and then eliminate by hand all its Domain. By keeping the domains and their original locations would be a higher modular approach. If a new system is to be integrated, as long as the Domain is in its structure, there is lesser risk to the changes made to the code. If a single unified layer would be used, the risk of breaking the dependencies would be higher.

Not all the tools were integrated in our project, as we chose Edgar to be our basis module, our main brain, deciding to complement it with ProgramD, SSS and Talkpedia, as they are of different natures when it comes to the answer domain. We also integrated TalkIt, not to be used as a dialogue system but opted to use its classifier. We can integrate JustChat as a filter module, taking advantage of its three filters. A user interaction would enter, the module would filter it based on what the user wants, and would return a treated interaction to the Answer Retrieval Algorithm. JustAsk can also be integrated, as its functionality resembles Talkpedia. We would need to create the WebService relation, and insert the module in our algorithm.
5

Implementation

Contents

5.1 Introduction ......................................................... 55
5.2 Plugins ............................................................. 57
5.3 Answer Retrieval Algorithm ...................................... 58
5.4 Towards a flexible configuration ................................. 60
5.1 Introduction

Our main feature is the integration of several L2F systems, maintaining a fully functional base and a more complex ability to configure the systems integrated. The most challenging feature of this thesis project is how to harmoniously conjugate all different systems in one. Each was built to work on its own, with the exception of Edgar and ProgramD that were already working together, and they were never thought as systems that one day would work in a synergetic environment.

The logical choice for the programming language would always be the one that would allow continuation of previous works, instead of utterly changing the paradigm and base work of the systems that would be incorporated. As all system were developed in Java, there was no question in trying to explore different options.

Exploring the different systems would give us an idea on how to conjugate them all, with an effort to make its architecture as modular as possible. Our initial idea was to focus on the modularity of our project, so that its scalability could be guaranteed with the addition of other functionalities. This would also allow us to separate the core of the project from its graphic component. As we went to a deeper level of the study of our applications, and how they could be integrated with each other, we started realizing that this modularity would not be as functional as desired.

Every system was designed, and built, to work in a single architecture, with the exception being Edgar and ProgramD. Even though TalKit also integrated SSS and Talkpedia, using a Java wrapper, the approach used by Edgar guarantees a higher tolerance to faults. If the code from any of the integrated systems should stop working, we have to assure the chatbot would keep functioning and giving believable answers to the users, until the system can be put back online. With the Java wrapper approach, if any unexpected exception or any code sequence that would break the execution happened, our whole system would stop working, causing a major malfunction. Also, if any bug is reported and there is a need to update the malfunctioning code, we would have to change it, compile it, create the wrapper and copy it to our systems folder.

If we can use a main system, integrating the other systems with it, but using a Plugin approach with WebServices, like the one exampled with Edgar and ProgramD, even if one of our systems would break, the rest of the system would keep functioning. The only issue is that we had to guarantee, programatically, that our code would be prepared to respond accurately to any plugin instantly breaking. This was implemented in our Answer Retrieval Method, by stating that if no response was returned from the evoked plugin, the system would keep trying other plugins, until one had an answer that could be retrieved to the user. But we still had to choose one system to be able to receive these plugins, and to gather a response if every other system would fail. As Edgar is the most complex system, and the one that already implements an example of this plugin methodology, it was our logical choice to use him as our main brain. If any bug is reported in this methodology, we can correct it while the system is still
working, exception made to bugs in Edgar, and put it immediately back online so that our system is able to again use that system.

Using Edgar as our main brain for the answer retrieval method, we integrated it with the TalkIt classifier as well as with other answer retrieval systems like Say Something Smart and Talkpedia. The result is a more resourceful Edgar, specially with Out Of Domain interactions, with diverse methodologies incorporated to its answer retrieval ability. If we want a clean basis, Edgar is a system that is capable of functioning on minimal services, and from which we can build on our knowledge base through its XML files.

The central focus point was how to connect all the different systems, maintaining a stable environment without destroying all the previous work and still being able to give the user a high-level configuration permission to decide how the new system may work. Two major conflicting issues were then discovered, as we wanted to maintain the systems specific configuration in their own modules, but would also wanted to create an architecture that would easily allow a plug-and-play methodology for new plugins. This revealed to be a task bigger than the initial purpose, as plugins may have different purposes, and so they would affect different sections of our system. For example, we integrated the TalKit classifier so it could return the module that is more appropriate to answer to the perceived user interaction. This is completely different than using answer retrieval systems like Say Something Smart or Talkpedia, that return a single answer to each user interaction.

Although some modularity is allowed, every new plugin still requires programming skills. To guarantee the functioning of the base system, even if some plugins would break its execution, a WebService methodology was implemented. As the new systems were not built to work in this methodology, it required that the plugins code needed to be changed, trying always to maintain its level of authenticity and main purpose, but having to make some structural adjustments. For example, in the TalkIt classifier, there was not a method that would return the system to use and we could not use the answer retrieved by the classifier. Instead, we created a method that, following the answer retrieval algorithm already used [5], would return the system to use. With this result we can directly invoke the system and retrieve an answer to the user interaction.

For the answer retrieval method, two different methodologies are used, depending if the TalkIt classifier is active or not. If the TalKit classifier is active, the answer retrieval method follows the algorithm previously described, and when inputting a new plugin the programmer needs to guarantee several steps, namely that the WebService is well established, with a defined communication port and the existence of a method that returns an answer to a given utterance and that the new plugin is incorporated in the system decision algorithm of the classifier. If the classifier is turned off, the programmer needs to guarantee that the WebService is well established, with all the work that it is linked to it, but the invocation of the new plugin does not need to be treated so thoroughly.
We will now discuss the different configuration scopes that were implemented followed by an explanation of our Answer Retrieval Algorithm.

### 5.2 Plugins

Several L2F in-house systems were studied but only Edgar and ProgramD were prepared to be integrated in a single system. We had to make adaptations to the core code of each system, following a step by step implementation to make sure the WebService was well implemented in each one. We started by defining the new plugin’s attribute in Edgar’s main configuration file with a hostname, port and the attributes for the Answer Retrieval Algorithm.

Listing 5.1: Example of the Plugin Configuration in Edgar’s configuration file.

```xml
1 <!-- independent plugins -->
2 <plugin activate="true" class="l2f.dialog.plugin.ProgramDWebserviceClientPlugin" host="localhost"
3 port="9011" confidence="100"/>
4 <plugin activate="true" class="l2f.dialog.plugin.ClassifierWebserviceClientPlugin" host="localhost"
5 port="9015"/>
6 <plugin activate="true" class="l2f.dialog.plugin.TalkpediaWebserviceClientPlugin" host="localhost"
7 port="9016" confidence="50"/>
8 <plugin activate="true" class="l2f.dialog.plugin.SSSWebserviceClientPlugin" host="localhost"
9 port="9017" confidence="70"/>
```

In the system’s own configuration file, the same hostname and port also needed to be specified.

Listing 5.2: Example of the Talkpedia configuration in own file.

```xml
1 <config>
3 <maxWikiSearchResults>10</maxWikiSearchResults>
4 <wikiCache>resources/wikiBot/paragraphs/wikiCache.txt</wikiCache>
5 <htmlCache>resources/wikiBot/htmlPages/htmlCache.txt</htmlCache>
6 <POSTaggerConfig>resources/config/config_tagger_DM.xml</POSTaggerConfig>
7 <debugMode>false</debugMode>
8 <host>localhost</host>
9 <port>9016</port>
10 </config>
```

But the XML representations needs to be interpreted, so we had to add the code to read the new attributes, both in Edgar’s and the plugin’s Configuration parser.

In Edgar’s package l2f.webservice, a new set of files was necessary to materialize the new plugin WebService Client. In the plugin side the same was needed to implement the WebService, creating a package l2f.webservice but with the implementation of the service instead of the client.

This process was completed with the respective binding class in the package l2f.dialog.plugin of Edgar.

The new system now needed a listener to keep the channel open and trying to receive the calls that were made. For our system to be able to retrieve an answer, it was necessary the existence of a method that would receive a string, our user interaction, and would return another string, our possible answer.
Some of the integrated systems did not have this kind of method and so, we had to manually create it with the core functionality that would permit us to retrieve an answer.

5.3 Answer Retrieval Algorithm

With the level of configuration combinations that we allow the user to define, our answer retrieval method would have to be refined to contemplate all the possibilities. But it is not only the configurations that we had to be concerned, when dealing with multiple plugin systems, as their execution breaking from a malfunction would also be of some concern. We had to have a system that would keep its ability to respond to user interactions, even without being able to connect to all of the integrated plugins.

Because of this issue, there is a lot of code that can be reused but with some subtle particularities, depending on the configurations defined or if the plugins are offline.

![Figure 5.1: Answer Retrieval Method](image)

Our algorithm begins by receiving a user interaction, evaluating whether or not the classifier is active. If the classifier is inactive, the main agent, Edgar, will try to retrieve an answer from its knowledge base. After this step, every active plugin, defined by the user configuration, will be evoked and a list of valid answers, from the plugins, will be created. The answer retrieved from the main agent will be compared with the list of answers retrieved from the plugins, and the system with the highest confidence, defined by the user, will be the answer given to the user. If no answer is retrieved, the algorithm will retry this approach one last time before returning a failed answer. If still no answer can be found the system will inform the user that no agent or plugin was able to answer. This is a rare happening, since Edgar has implemented a system of diversion, with the invocation of an interaction \_REPEAT\_, that is prepared to avoid this situation with a believable answer such as seen in Listing 5.3.
### Listing 5.3: Example of Edgar repeat interactions.

```
<qa id="120209114" category="repeat">
  <questions>
    <q en="REPEAT." es="REPEAT.">_REPEAT_</q>
  </questions>
  <answers>
    <a en="I don't think I understood." es="Creo que no he entendido bien." >Se calhar nao percebi.</a>
    <a en="I didn't quite understand it." es="Creo que no he comprendido." >Acho que nao percebi.</a>
    <a en="Could you repeat, please?" es="Puede repetir, por favor?" >Podia repetir, por favor?"</a>
  </answers>
</qa>
```

If the classifier is active, the decision to which path to follow will reflect the configuration defined by the user of the attribute forcePriority. This attribute gives the user the ability to try to force the retrieval of an answer in the agent or plugin with the highest confidence. If such an attribute is active, the algorithm will firstly try to get an answer of the chosen agent or plugin. If an answer is retrieved, this will be the output given to the user. In case the agent or plugin with the highest confidence is Edgar, the algorithm will still run the user interaction with ProgramD. We chose to implement it this way as ProgramD is mainly an addition to the main agent Edgar.

In case that the chosen system is not able to retrieve any answer, the classifier routine will be used and the classifier will define the taxonomy category of the user interaction. If no category is found, the classifier is not able to retrieve which system to use and the algorithm will try to retrieve an answer as if the classifier was not active at all, just considering which systems are online and retrieving the answer from the system with the highest confidence.

If the classifier is able to determine a taxonomy to the user interaction, the answer retrieval algorithm will be similar to the one used in TalKit. Edgar will deal with most of the on-domain interactions, relegating to the other integrated plugins the task to deal with out-of-domain interactions, namely Talkpedia will deal with factoid questions and SSS will deal with non personal interactions. If the chosen plugin is SSS but it is not active, for definition, execution error or even if it can not retrieve an answer, Edgar will be the system chosen to deal with the interaction. If Talkpedia is chosen by TalKit but it is not active, for any of the already mentioned reasons, the algorithm will secondly try to use SSS, as it is an out-of-domain interaction, and only if SSS also fails, then the algorithm will call upon Edgar to try and retrieve an answer.

The algorithm path to retrieve an answer, if the classifier is active but the attribute forcePriority is inactive, is similar to the one when the attribute forcePriority is active but the agent or plugin with the highest confidence can not retrieve an answer.
5.4 Towards a flexible configuration

Making all systems flow together in a single environment was the main goal of our project. This was the most time consuming goal to achieve, always trying to understand how the systems would better fit together, but staying true to the purpose for what they were built. But this goal would not be completed with one single step. We wanted to give the user the power to structure and define this environment to his will, without losing focus of what our project final goal is, retrieving a believable answer to a user interaction.

We wanted the user to feel the power to choose what he thinks is the better conjugation of the participant systems. If the user feels that his chatbot should be focused on answering a majority of factoid questions, for example, for a fun-fact intended chatbot as his own personal knowledge base, then the user has the ability to give a higher confidence to Talkpedia. Determining what plugins are active or not, and the respective confidence given, will influence the answer retrieval algorithm.

5.4.1 The Brain

Although all plugged in systems may be deactivated, the main agent, what we defined as Edgar and worked as our brain, can not be deactivated and also has a confidence property attached to it. Imagining that all systems would fail, and only Edgar, our main agent, would still be online, our system would always try to determine a believable answer, with his current knowledge base. Only if all corpora would be removed, would Edgar stop being able to try to determine a valid answer.

<table>
<thead>
<tr>
<th>Listing 5.4: Edgar Configuration.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt;webservice&gt;</td>
</tr>
<tr>
<td>2  &lt;!-- &lt;host&gt;localhost&lt;/host&gt; --&gt;</td>
</tr>
<tr>
<td>3 &lt;host&gt;172.16.254.34&lt;/host&gt;</td>
</tr>
<tr>
<td>4 &lt;port&gt;9012&lt;/port&gt;</td>
</tr>
<tr>
<td>5 &lt;agentName&gt;Edgar&lt;/agentName&gt;</td>
</tr>
<tr>
<td>6 &lt;agentConfidence&gt;90&lt;/agentConfidence&gt;</td>
</tr>
<tr>
<td>8 &lt;/webservice&gt;</td>
</tr>
</tbody>
</table>

In Listing 5.4 we can see the basic configuration of our agent. We define a hostname and a port to which the WebService should be pointed to. A symbolic name, in this case Edgar, can be given to the agent, although it will not have any direct effect on the dialogue system, as it used most for log purposes. The agentConfidence property is the attribute that will represent, to our system, how much trust does the user have in the agent's ability to retrieve a believable answer.

5.4.2 Plugins

The range of confidence is not limited purposely to ensure that, no matter how many systems may be connected, there is always the possibility to rank them in a different unitary system, where each has
its own value.

The TalkIt classifier is the first decision paradigm that the answer retrieval method faces. Whether it is active or not, the agent will choose different paths to follow. For example, if the classifier is deactivated, the agent will order its retrieved answers to satisfy the user's configuration of the plugins confidence, without overlooking the main agent's confidence. The system that retrieves a valid answer and has the highest confidence will be the one chosen as the final answer. If the TalkIt classifier is activated, one of two paths may occur, depending on the configuration of the property forcePriority. This priority, when active as the boolean true, will always force the plugin with the highest confidence to retrieve an answer. If such answer does not exist, then, the answer retrieval method will invoke the classifier and follow the path that TalKit has defined.

If this forcePriority property is false, the TalkIt classifier will run the same path as if no answer was retrieved from the highest confidence system. This implementation was made to give more freedom of decision to the user. For example, if he has most confidence that Talkpedia will be able to give the most valid answers, but wishes to ensure that, if Talkpedia fails, the chatbot will be able to follow the logical and trained path defined by the classifier, without leaving the answer retrieval method to be solely based on the confidence the user has on the systems.

Listing 5.5: Configuration attribute forcePriority.

```xml
1  <!-- Whether the classifier is active or not, always give priority to the plugin (or agent) with the highest confidence level.-->
2  <forcePriority>true</forcePriority>
```

5.4.3 Language

At this stage, our project does not allow the language to be altered from Portuguese to any other desired. Although Edgar and Say Something Smart are able to, individually, retrieve answers in other languages, ProgramD, TalkIt and Talkpedia were designed to simply function in Portuguese. To change this and make it viable to be used in multiple languages as a whole, it would require work that a user without programming knowledge would not be able to do. As ProgramD works with an AIML database, all AIML rules and respective answers would have to be translated into new files. TalkIt is a system trained only for the Portuguese language, but the TalkIt classifier could be substituted or even a new classifier could be plugged in. This would require that many changes would have to be made to the main code, as our project is not prepared to face more than one classifier. Talkpedia would be the system that would require less programming changes to work in a different language, although it is not prepared to simultaneously work in a multi-language environment. The URL pointer would have to be addressed to the desired language Wikipedia, assuming it exists, and the default base sentences for the final answer would have to be translated.
5.4.4 Other Systems Configurations

Despite having this main configuration possibilities, every system has its own configuration, although not easily modifiable by a user without previous knowledge of the system or without programming skills. These were purposely left inside each system, so it better fits our modular approach to the problem. Our main brain, Edgar, can be used with a different corpora or even with its own knowledge base enlarged by the addition of XML files with interactions that he could directly answer. ProgramD, as previously mentioned, allows us to create a set of rules and equivalences that Edgar is able to detect and answer in conformity. TalkIt is a system that is able to retrieve an answer to a user interaction but, in our project, we decided to use only its most eloquent tool, the classifier. To retrain the classifier, a new training set and corpora would be needed. In Say Something Smart, there are 4 measures to be accounted for, as previously described in Chapter 2. The user can configure the weight of these measures as he assumes they would better work together, by instinct, by trial-and-error or any other way the user sees fit. Talkpedia is the less configurable system, as all its knowledge is based on a URL connection to the Wikipedia server, retrieving the necessary information and trimming it. The user can only define the possible template phrases where this information will be used.
6 Evaluation

Contents

6.1 Introduction ......................................................... 65
6.2 Scenarios ............................................................. 65
6.3 User Evaluation ...................................................... 69
6.1 Introduction

With most of the focus on this project to its architectural structure and system integration, we could not overlook the purpose for what the project was being built, becoming a conversational agent system. There are some ways to test a software architecture but more from an applicative and technical point of view, usually being done at an early stage of the development of the project.

In an ideal project, we would start to develop and plan several architectural approaches to understand what would best fit our needs. Then, we would simulate the effort to keep the dependencies between possible modules, dividing them in layers. An iterative methodology would be used to understand flaws and rectify them until a possible solution is reached. Only then would the implementation phase start.

As much as this project was intended to result in a specific architecture, the limitations of working with already developed systems resulted in an adaptation of the architecture and in an inadequate timing for this type of testing. When a project has the kind of artifacts that this one had, already produced, it is more appropriate to perform more of a functional testing.

With this in mind, two types of functional testing were applied. Ten different scenarios were defined, by the Thesis’ supervisor Professora Luísa Coheur, to understand how limited or not our project is. These scenarios identify possible uses, that a user with any kind of interests could realize, and we will try to see if such demands are possible or not. But this is, in fact, a system built for the end user to interact with. Although the quality of the answers is not entirely related to the developed work, it is still a goal to produce believable answers. A script, briefly explaining the project, is handed to a user and he will have to perform two different tasks that involve configuration and interaction with the system. In the end, the user will fill a form so that we can understand how satisfied he is with the retrieved answers and how adequate they were.

6.2 Scenarios

Our objectives always revolved around how configurable this system could be to the end user. We intended to give more power and control to the users, while integrating systems that already have their own configurations. Not only this is not an easy and straight forward task, as it is not simple to evaluate. We can not evaluate the system simply on the answers retrieved because the dialogue systems that have that functionality were already created and we did not have the task to improve them.

Through the definition of ten possible user configuration scenarios, we were trying to simulate needs and desires of an end user. We did not test them with users but we tried to respond to the needs, in a configuration point of view, of each scenario, trying to figure out if such is possible or not, and how we could satisfy those specifications.

Scenarios:
1. How to define an agent where all the possible interactions are previously defined by hand and that if it can not answer, it does not?

Both Edgar and ProgramD are a viable and valid solution to a user that wishes to define each interaction that the conversational agent can answer. While a user interface is not available, the user would need to have some knowledge of XML or AIML. With a GUI implemented it would be an interesting addition to have a tool that could automatically create the code necessary to expand the corpora by hand.

**Listing 6.1:** Example of XML Scenario 1.

```xml
<qa id="1202090">
  <questions>
    <q en="What's your name?" es="Como te llamas?"">Como e que te chamas?</q>
    <q en="What's your full name?" es="Como es su nombre completo?">Qual e que e o seu nome completo?</q>
    <q en="Who are you?" es="Quien eres?">Quem e que es?</q>
  </questions>
  <answers>
    <a en="My name is Edgar Smith." es="Me llamo Edgar Smith." emotion="friendly" intensity="100">Chamo-me Edgar Smith.</a>
  </answers>
</qa>
```

**Listing 6.2:** Example of AIML Scenario 1.

```xml
<category>
  <pattern>* es brilhante</pattern>
  <random>
    <li>Obrigado. Fico muito contente.</li>
    <li>Obrigado. O seu comentario e lisonjeador.</li>
    <li>Muito obrigada pelas suas palavras.</li>
    <li>Muito agradecido.</li>
  </random>
</category>
```

2. How to define an agent that can answer to factoid questions and some personal questions?

By activating Talkpedia, our system is automatically able to respond to factoid questions. To define some personality traces to which our agent could answer, there would be a need to, as in Listing 6.1, define the interactions by hand through an XML corpora file.

---

1 edgar/resources/qa/corpusXML/edgar-2013-05-23/*.xml
2 ProgramD/aiml/edgar/*.aiml
3 edgar/resources/qa/config/dialogConfig_pt.xml
3. How to define an agent that has some hand defined personality traits but only answers to small talk.

Every aspect of the personality of a conversational agent can be defined in an XML file located in the ProgramD plugin\(^4\). The corpora in Edgar defines the domain to which the agent can answer, so if we want the agent to only be able to interact with small talk, the corpora should be decreased to the files that have that specifications\(^5\).

4. How to define an agent that can answer factoid questions, has hand defined personality traits and answers to small talk.

We already covered the steps necessary to satisfy each condition, so we only need to conjugate them in a single configuration. Activate Talkpedia, have a small talk file in our corpora and define personality traits in ProgramD.

5. How to hand define an agent that can answer in Spanish.

The integration of multiple systems did not contemplate the multi language possibility. Some systems were created to work in Portuguese and some in English. Only Edgar has the ability to answer in Spanish but ProgramD, with its AIML, can be hand defined to understand Spanish. The other integrated systems would require more effort. SSS would need a knowledge base in Spanish, Talkpedia would need its templates to be written in Spanish and to change the Wikipedia used, from the Portuguese to the Spanish, and TalKit would need to be trained with a Spanish Corpora before being used.

6. How to define an agent that can integrate a new Question & Answer engine to the specifications of item 4.

---

\(^4\)ProgramD/conf/edgar/properties.xml

\(^5\)edgar/resources/qa/corpusXML/edgar-2013-05-23/small-talk.xml
We do not have an implemented Plug & Play methodology for our external systems, since most systems were developed without the contemplation to work in an integrated environment and it involves a lot of effort to adapt the code, but an WebService methodology was implemented that can, with less effort, combine any new QA system. As the WebService classes would need to be created and adapted, the system would also need to be contemplated in the main configuration file and in our Answer Retrieval Algorithm.

7. **How to define an agent that can answer to interactions like Yes/No Questions or Rhetorical Questions.**

The TalKit system as a whole is prepared to answer to that type of interactions, but we decided only to integrate the classifier, leaving the answer functionality to other systems. With some effort, specially in the aspect of the changes necessary to the existing TalKit code, our system would be able do include that feature. In this moment, we can not guarantee an agent that would be completely able to do this kind of interactions, with our best possibility being the configuration of SSS measures to focus in user input similarity or the direct definition of that kind of trigger in Edgar's corpora.

8. **How to define an agent with a new knowledge base of factoid answers, for example, based on Answers.com\(^6\), and using it with high priority.**

Although Talkpedia is a system used to retrieve answers from a given URL, it is only prepared to deal with the schematics and structure of Wikipedia, not being able to withstand other factoid answering websites, like Answers.com. To perform this feature a whole new system would be necessary.

9. **How to define a new classifier, new QA Module and how to adjust the new classifier.**

To replace the current classifier TalKit, we would need to create a new method that relates the classification with the integrated modules in our project. Afterwards, we would need to relate the existing WebService methodology with the new classifier. Based on the new taxonomy and evaluation, we would have to specify which systems would be best fit to answer each type of interactions. The same process would need to be applied to a new QA module to be integrated in our project, but the answer retrieval algorithm would need to be revised to include this new module in its consideration and decision methodology.

10. **How to define different priorities to different systems.**

Any dialogue system plugged in our system can have a numerical confidence. The higher the confidence, the higher the priority given to an answer coming from that module. Our project is

\(^6\)http://www.answers.com/
prepared to work with the expected and integrated systems all at the same time or even with only Edgar active, as Edgar can not be deactivated. If Edgar corpora is reduced to a minimum or, let us say, non existing, then our agent would not be able to retrieve any answer as it would not have any knowledge base to work with.

6.3 User Evaluation

Even though we do not want to judge the success of our project based on the quality of the retrieved answer, as it is a process that little has to do with the developed work, we still want to gather some information about the adequacy of the interactions. The evaluation is performed in our hardware, where the project is already installed and the paths configured. As our project include other tools, with big corpora, knowledge bases or even training files for the classifier, it has a memory space requisition that we did not want to put our users through. Doing the whole user evaluation process in our computers saves everybody time and effort.

A script was defined (Appendix A), explaining what our system consists of and briefly explaining the tools integrated with it. As the user is reading the script, we open the configuration files in a text editor so the user can handle them while performing the tasks. In a completed system, the project would have a simple GUI that would spare the user from the configuration code, with simple buttons and text to alternate between all possible configurations. Since we do not yet have this implemented, and the user does not have to know where the configuration files are, we decided to open them in an editor but all interaction with the files would be of the user’s responsibility. Then, we launch the system in the command line but delegate all textual interaction to the user, also. We want to evaluate the interactions and how well the user interacts with the configuration possibilities. Each task will end with the registration of ten user interactions, the user evaluation of the answers adequacy and the possibility given to the user to write what he thought of the system.

In the first task, where a more factoid agent was defined, the agent answered all factoid questions with a high level of satisfaction, where all answers were categorized with the maximum grade. It did not behave as well when asked personal questions or interactions that would go out of factoid domain, with the adequacy ranging from the lowest grade to medium. Since the forcePriority attribute was active and focused on Talkpedia, it is normal that a high level of adequacy would be verified. Some adjustments should be made to the Answer Retrieval Algorithm, as the agent returns very inadequate answers to small talk or personal questions, even with all systems active. The answers returned from out-of-factoid questions would only reach the medium grade if a topic or main word could be retrieved from the user interaction. For example, “do you like sushi?” would not get a personal answer but instead it would retrieve the Wikipedia definition of sushi. As it is not entirely incorrect, because the agent is more
focused on factoid interactions, it does not completely satisfy the question because of its personal tone.

The second task defined a more embracing agent, without a specific domain, and with SSS as the system with the highest confidence, with the TalKit classifier active but without the forcePriority attribute. This agent did not perform as expected, with most of the answers being graded average or below. The interactions that performed worst were the ones that TalKit would classify as Personal, and which the classifier would delegate the task of answer retrieval to SSS. This approach followed the algorithm that was already implemented in TalKit that, derived from our user evaluation, we can now perceive as inadequate to a system with such a large spectrum of implemented tools. TalKit was concluded as a Dialogue System, able to return answers to the user interactions based on SSS, Talkpedia and some predefined templates. We opted to maintain the usage of SSS and Talkpedia in the same taxonomy categories, Personal and Impersonal respectively, and replace the template answers, for the other categories, with Edgar combined with ProgramD.

This approach is not the best when dealing with our systems, as the answers did not satisfy the user evaluation. A better approach for the TalKit classifier would be to delegate to Edgar and ProgramD our Personal Interactions, as they can be built around personal specifications, keep Talkpedia with Impersonal questions, where most of them are factoid, and assign SSS to other taxonomies. Other possible approach would need to relate our answer retrieval algorithm to the TalKit domain so that we can also retrieve the templates. This was initially considered but, as our goal was to use TalKit only as a classifier and not a Dialogue System, we decided to only use the other dialogue systems as valid answer retrieving tools.

Because of a lack of time planning and delays caused by upsets in the implementation, our user evaluation only had one test subject. In the first task, more than half of the answer received the maximum level of adequacy, with one being completely off and the remaining three keeping their ranking at medium level, which ranks the global adequacy of the answers for this task at 4.

The second task had a more even distribution of values as we can see in Table 6.1. The average adequacy for this task was 2.7, which is below our expectations and noteworthy for any future work.

For a more accurate user evaluation, it would be necessary to conduct these tasks with a higher number of persons.

**Table 6.1: User Evaluation**

<table>
<thead>
<tr>
<th>Rank</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 - Muito Adequada</td>
<td>6</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>4 - Adequada</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3 - Médio</td>
<td>3</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2 - Inadequada</td>
<td>0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1 - Completamente Inadequada</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
7 Conclusion

Contents

7.1 Future Work ............................................................ 74
More than a scientific research assignment, this Master’s Thesis was a hands-on project. We were not aiming to reach new scientific achievements but, instead, we were aiming to build the foundation of a system that could integrate and support most of the L2F Dialogue Systems. New L2F projects are created every semester and this is a project that does not have a definite ending, as it should be the main system where all these new projects should be incorporated. Instead, this is a project that represents the end of a cycle and the beginning of a new one.

We started studying what were the systems we had to work with. What was their purpose, how they worked and what were the inputs and outputs that each was prepared to use. We could not look at these systems in a black box methodology. We needed to understand where the dots would connect and what was the bigger picture. Where we were heading was as important as how we would get there.

But we could not look only to systems developed in-house. We needed to broaden our horizons, step outside and see what was out there. We were trying to understand where our project could be unique and where it could distinguish from others. Our biggest goal would have to explore the lack of configuration possibilities that other systems allow. They will let you define the Domain, but how many would allow the user to configure how he wants the system to act? Others may have a more flexible architecture, but we built an architecture that, with some level of modularity and layer distinction, can integrate our own systems. Systems who were never designed, or even expected, to work in a single symbiotic environment.

We needed to define an architecture and we were ambitious. We aimed high and envisioned a design that would integrate all systems with a Plug & Play methodology, in a 4-tier layer architecture. We wanted high modularity combined with high functionality. Although ambitious, our modularity would diminish the quality of our functionality and we had to adapt our initial architecture to one that could conjugate goals of modularity, functionality and customization. As we learned in latter stages, the creation and planning of such a project, where our artifacts are already designed and implemented, is a process that involves several iterations. And that was exactly how things ended up, with iterations over iterations where we were refining our design, so that it would meet our needs.

This architectural iteration process, as it was not planned initially, ended up meddling with the implementation phase of the project. As we were facing integration obstacles, created by the systems we defined would make the foundation of our system, we were adapting our architecture so that we would still have a layer definition, but with the difference that we started using one system, Edgar, as our central module, where all others would connect. This way, the handling of the communication would be exclusive of Edgar and the Edgar module would be responsible for delegating tasks and withstanding our Answer Retrieval Algorithm.

As we were integrating the systems, we had to guarantee that our objective of giving the user more power over the system, than in other observed systems, was being accomplished. We defined con-
fidences so that the user could define what are his most trusted answer retrieving tools and how the system should order them. The inclusion of the TalKit classifier opened more decision paths in our algorithm, as well as the ability to, no matter what is the configuration, try to force one of the systems to return an answer. This all complements the already possible configurations of each system, that we decided not to include in our main configuration file, as that would create dependencies that would break the modular aspect of the architecture.

To evaluate the created system, a set of constraints and specifications were defined. We wanted to understand and explain how our system would adapt to new possibilities and the amount of effort that each specification would need to be integrated. We also performed user oriented evaluation to see the performance of our system in a user interaction environment, although the adequacy of answers was not one of our main goals, as it has dependencies that were not implemented by us. From this user evaluation we were able to understand what else we need to improve in our own Answer Retrieval Algorithm.

In conclusion, we believe that the basis of our structure is successfully implemented and that we can continue to build on the developed system.

7.1 Future Work

Our system is a continuous project. There are new systems being developed in-house and it will always be an objective to integrate new features in our system. It would be interesting to add a Graphical User Interface (GUI), specially to facilitate the usage of the configuration aspect by the end user. For example, an interface where the XML files would be translated into buttons to activate or deactivate properties of our system, to add new set of rules to the AIML of ProgramD or even to be able to define the weights of SSS in a graphical environment.

We worked on this project with an approach to its logical side, but a conversational agent can have more features than just the ability to return answers. It would be interesting to add a context driven algorithm so that the answers become more believable and more related to the user input.

The refinement of the Answer Retrieval Algorithm should always be a work in progress. As more systems are integrated, the broaden is the spectrum of paths from which to choose, to retrieve the best possible answer.

A more extent user evaluation would be useful, to determine where we can improve the effectiveness of our system. The definition of new tasks would also help us to prevent a higher range of different configurations.

New features of customization could be implemented, always with the goal to give the user more power of his own conversational agent. As the system gets bigger, with the addition of new plugins,
there may be a way to create relations and volatile dependencies that could be defined by the user.
Bibliography


77


Questionário de Avaliação

No âmbito da tese de Mestrado em Engenharia Informática e de Computadores, gostaria de pedir a tua disponibilidade para a realização de 2 cenários de avaliação sobre a plataforma desenvolvida.

Esta plataforma destina-se à criação de agentes conversacionais, aglomerando diversas plataformas já existentes numa só. Apesar de existir sempre uma relevância dada quanto à adequação da resposta à interacção, pretendemos avaliar principalmente a nossa componente de configuração da plataforma. Para tal, criámos 2 tarefas que deverás realizar sem qualquer ajuda que não seja o presente guião.

No final de cada tarefa, deverás realizar 10 interacções e responder a um pequeno questionário sobre a adequação das respectivas respostas. A nossa plataforma está preparada apenas para interacções em Português.

Regista, no formulário disponibilizado, as interacções que utilizaste e se consideras a resposta adequada. A classificação utilizada deve ser:

- 1 - Completamente Inadequada
- 2 - Inadequada
• 3 - Médio
• 4 - Adequada
• 5 - Muito Adequada

Existem 5 ferramentas importantes na nossa plataforma: Edgar, ProgramD, Say Something Smart (SSS), Talkpedia e TalKit.

- O Edgar é o nosso cérebro principal e o agente conversacional em si. No seu ficheiro de configuração poderemos definir que ferramentas estão activas ou não e qual o nível de confiança que lhes atribuímos. Para a activação (ou desactivação), basta passar a propriedade activate para true ou false, respectivamente. No caso da confiança, podes defini-la no campo agentConfidence como um inteiro, onde quanto maior o número, maior a confiança. Este número poderá variar entre -2147483648 e 2147483648. Como o Edgar é o nosso agente principal, este nunca poderá ser desligado.

- No ProgramD podemos definir um conjunto de regras AML que o Edgar fica habilitado a responder, complementando a informação e base de conhecimento do agente principal.

- O SSS está preparado para responder a questões fora do domínio habitual do agente. Para tal, as suas respostas são dadas com base em legendas de filmes. Existem 4 medidas configuráveis: Answer Frequency, Answer Similarity to User Question, Question Similarity to User Question e Time Difference. Estas medidas podem ser conjugadas com diferentes pesos, podendo cada uma receber um valor inteiro entre 0 e 100, sendo que a soma do seu total tem de ser 100.

- A Talkpedia utiliza a Wikipedia como base do seu conhecimento, ou seja, está preparada para responder a factóides. Uma questão factóide é uma pergunta que questiona características de um facto.

- O TalKit classifica as interacções do utilizador e decide qual o melhor caminho a escolher para que a probabilidade de existência de uma resposta adequada seja maior. Também pode ser desligado, passando a ser por ordem de confiança que o agente tentará responder à interacção. Mesmo com o TalKit activo, existe a possibilidade de continuar a ter uma ferramenta preferencial e onde procuramos sempre obter a resposta antes de recorrer aos restantes componentes. Se o utilizador desejar pode activar ou desactivar esta funcionalidade na propriedade forcePriority (passando a propriedade a true para activar ou false para desactivar) do ficheiro de configuração do Edgar.

Tarefa 1: Configura o agente para que este seja o teu acesso directo à Wikipedia e possa resolver a maior parte das dúvidas que possam surgir, como quem foi Eusébio ou o que é sushi. Para tal:

• Activa a Talkpedia e dá-lhe mais confiança do que a qualquer uma das outras plataformas.

• Activa a propriedade forcePriority, para que a Talkpedia seja sempre a ferramenta principal para obter uma resposta, mesmo que tenhas desligado o TalKit.
• Podes definir as restantes confianças como bem entenderes ou até desligar os restantes plugins.

• Realiza 10 interacções onde procura utilizar a plataforma como base de conhecimento factual e regista-as, como previamente explicado.

Tarefa 2: És um fanático de filmes e queres interagir com o agente sobre um pouco de tudo.

• Activa o SSS e dâ-lhe mais confiança do que a qualquer outra das plataformas.

• Vai ao ficheiro de configuração do SSS e define as medidas como melhor entenderes

• Activa o TalKit.

• Desactiva a propriedade forcePriority.

• Podes definir as restantes confianças como bem entenderes ou até desligar os restantes plugins.

• Realiza 10 interacções livres e regista-as, como previamente explicado.
Avaliação da Plataforma

Regista as interacções que utilizaste e se consideras a resposta adequada. A classificação utilizada deve ser:
1 - Completamente Inadequada
2 - Inadequada
3 - Médio
4 - Adequada
5 - Muito Adequada

1ª Interacção
Qual foi a tua interacção?

1ª Interacção

1 - Completamente Inadequada
2 - Inadequada
3 - Médio
4 - Adequada
5 - Muito Adequada

2ª Interacção
Qual foi a tua interacção?

2ª Interacção

1 - Completamente Inadequada
2 - Inadequada
3 - Médio
4 - Adequada
5 - Muito Adequada

3ª Interacção
Qual foi a tua interacção?

3ª Interacção

1 - Completamente Inadequada
2 - Inadequada
3 - Médio
4 - Adequada
5 - Muito Adequada

4ª Interacção
Qual foi a tua interacção?

4ª Interacção
1 - Completamente Inadequada
2 - Inadequada
3 - Médio
4 - Adequada
5 - Muito Adequada

5ª Interacção
Qual foi a tua interacção?

5ª Interacção
1 - Completamente Inadequada
2 - Inadequada
3 - Médio
4 - Adequada
5 - Muito Adequada

6ª Interacção
Qual foi a tua interacção?

6ª Interacção
1 - Completamente Inadequada
2 - Inadequada
3 - Médio
4 - Adequada
5 - Muito Adequada

7ª Interacção
Qual foi a tua interacção?
7ª Interacção
○ 1 - Completamente Inadequada
○ 2 - Inadequada
○ 3 - Médio
○ 4 - Adequada
○ 5 - Muito Adequada

8ª Interacção
Qual foi a tua interacção?

8ª Interacção
○ 1 - Completamente Inadequada
○ 2 - Inadequada
○ 3 - Médio
○ 4 - Adequada
○ 5 - Muito Adequada

9ª Interacção
Qual foi a tua interacção?

9ª Interacção
○ 1 - Completamente Inadequada
○ 2 - Inadequada
○ 3 - Médio
○ 4 - Adequada
○ 5 - Muito Adequada

10ª Interacção
Qual foi a tua interacção?

10ª Interacção
○ 1 - Completamente Inadequada
○ 2 - Inadequada
○ 3 - Médio
○ 4 - Adequada
○ 5 - Muito Adequada

Ajuda-nos a melhorar! Fala sobre o que gostaste mais, menos e sugestões de melhoria!
Never submit passwords through Google Forms.