

**Say Something Smart 3.0:  
A Multi-Agent Chatbot in Open Domain**

**João Miguel Lucas Santos**

Dissertation submitted to obtain the Master Degree in  
**Information Systems and Computer Engineering**

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

**Examination Committee**

Chairperson: Prof. José Carlos Martins Delgado

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

Member of the Committee: Prof. Rui Filipe Fernandes Prada

**November 2019**



# Abstract

Dialogue engines that focus on a multi-agent architecture often trace a single, linear path from the moment when the system receives a query until an answer is generated by selecting a single agent, deemed to be the most appropriate to respond to the given query, not granting to any of the other available agents the opportunity to provide an answer.

In this work, we present an alternative approach to multi-agent conversational systems through a retrieval-based architecture, which not only takes the answers of each agent into account and uses a decision model to determine the most appropriate answer, but also provides a plug-and-play framework that allows users to set up and test their own conversational agents.

Say Something Smart, a conversational system that answers user requests based on movie subtitles, is used as the base for our work. Edgar, a chatbot specifically built to answer requests related to the Monserrate Palace, is also incorporated into our system in the form of a domain-oriented agent. Furthermore, our work is embedded in Discord, a social text chat application which allows for users all over the world to engage with our chatbot.

We integrate work previously done on Online Learning into our platform, allowing the system to learn which agents have the best results when answering a given query. We evaluate our system on the matter of dealing with both out-of-domain queries and domain-specific questions, comparing it to the previous instances of Edgar and Say Something Smart. Finally, we evaluate the outcome of our system's learning against previous experiments done on Say Something Smart, achieving a better answer plausibility than previous systems when interacting with human users.

## Keywords

Dialogue Systems

Plug-and-Play Agents

Multi-Agent Platforms

Online Learning



# Resumo

Os sistemas de diálogo construídos com base em arquiteturas multi-agente são normalmente compostos por um percurso linear desde do momento em que o sistema recebe um pedido de um utilizador até à resposta ser gerada através da selecção de um único agente, considerado o mais apropriado para responder ao pedido recebido, sem dar aos restantes agentes qualquer oportunidade de apresentarem a sua própria resposta.

Neste trabalho, apresentamos uma abordagem alternativa aos sistemas de diálogo multi-agente através de uma arquitectura baseada em retrieval que não só tem em conta as respostas de cada um dos seus agentes e utiliza um modelo de decisão para decidir qual a resposta mais apropriada, mas fornece também uma estrutura plug-and-play que concede a qualquer utilizador a possibilidade de configurar e testar os seus próprios agentes conversacionais.

Como base para o nosso trabalho utilizamos o Say Something Smart, um sistema de diálogo que responde aos pedidos do utilizador através de uma base de conhecimento de legendas de filme. Edgar, um chatbot construído especificamente para responder a perguntas sobre o Palácio de Monserrate, é também incorporado no nosso sistema como um agente de domínio específico. O nosso trabalho é também integrado no Discord, uma aplicação de troca de mensagens, de forma a permitir que utilizadores de qualquer parte do mundo tenham a possibilidade de interagir com o nosso chatbot.

Integramos também na nossa plataforma trabalho previamente feito acerca de Aprendizagem Online, o que permite que o nosso sistema aprenda quais os agentes com melhor desempenho a responder a uma dada pergunta. Avaliamos o nosso sistema em relação a perguntas de domínio específico e também pedidos fora-de-domínio, e comparamos os nossos resultados com o trabalho feito anteriormente do Edgar e do Say Something Smart. Por fim, avaliamos os resultados da aprendizagem do nosso sistema contra os resultados obtidos na aprendizagem original do Say Something Smart, obtendo uma maior plausibilidade nas respostas dadas a utilizadores humanos.

## Palavras-Chave

Sistemas de Diálogo

Agentes Plug-and-Play

Plataformas Multi-Agente

Aprendizagem Online





# Acknowledgements

Firstly, I would like to thank my thesis advisor, Prof. Luísa Coheur. She's one of the most amazing people I've ever met, and I'm so glad I had the privilege of working with her for this past year. Thank you for taking me into your care, and for everything you've done.

I would like to thank Mariana and Leonor, who have been awesome throughout this whole year, and whose friendship and cooperation greatly helped me get through this work without going insane.

I would also like to thank Vânia for all of her help, availability and overall cheerfulness throughout the development of this work.

Next on, I would like to thank my mother Maria do Carmo, who's the strongest person I've ever known, and to my father Francisco, who's always done his best to take every burden out of my shoulders. Thank you both for always supporting my decisions with all your heart, for calling me out when I needed it, and for all the boundless love you've given me. I can only hope to be able to reciprocate it as best as I can.

To my brother Rodrigo, thank you for being my greatest teacher ever since I was little, for always being there to listen to my silly self, and for being someone who makes me strive to do better just by being there. I don't believe I'd ever reach this point if it wasn't for you.

To my sister Sílvia, thank you for being there every time I start faltering and teaching me, time and time again, to pick myself up and focus on what matters, and thank you for making it so easy to smile. No one could ever replace you.

To my grandparents, Nazaré and Albino, thank you for taking care of me throughout these twenty-two years, and for supporting me so much, especially during these past two years.

To my brother-in-law Mário and to my nephew Lourenço, thank you for bringing so much joy to my life in such a short period of time. I'm looking forward to the future with you guys in the family.

To Dário, thank you for always, always being there for me and always having my back. We may not be bound by blood, but you're still my brother, and nothing in this world can change that.

To all of my friends, thank you for being a part of my life. I'm extremely lucky to have been a part of yours, and I can only hope that we share many years into the future. I would like to thank Luís, Gonny, Ruben, Gui, Ico, Emanuel, Sam, Soares, Tiago and Desert, as all of you have greatly influenced my life throughout the past two years. As a special note, I would like to thank everyone in the SO Discord server for all the fun moments and for always going along with my wacky ideas and inventions.

Finally, I would like to dedicate this work to my nephew Lourenço. Someday, you'll choose your own path as well.

*João Santos*

*Escravilheira, 27 de Outubro de 2019*



This work contributes to the FCT, Portugal INCoDe 2030 National Digital Skills Initiative, within the scope of the demonstration project "Agente Inteligente para Atendimento no Balcão do Empreendedor" (AIA).



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	2
1.2	Objectives . . . . .	3
1.3	Document Organization . . . . .	3
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	The SubTle Corpus . . . . .	6
2.2	Say Something Smart . . . . .	7
2.3	SSS's Redundancy Bug . . . . .	8
2.4	Edgar . . . . .	10
<b>3</b>	<b>Related Work</b>	<b>13</b>
3.1	Indexers . . . . .	14
3.1.1	Lucene . . . . .	15
3.1.2	Elasticsearch . . . . .	16
3.1.3	Others . . . . .	16
3.2	Similar Architectures . . . . .	16
3.2.1	DocChat . . . . .	17
3.2.2	IR System Using STC Data . . . . .	17
3.2.3	Gunrock . . . . .	19
3.3	Similarity Metrics . . . . .	20
<b>4</b>	<b>Upgrading SSS into a Multiagent System</b>	<b>23</b>
4.1	Proposed Architecture and Workflow . . . . .	24
4.2	Building the Agents . . . . .	25
4.3	Building the Manager . . . . .	26
4.4	Building the Decision Maker . . . . .	26
4.4.1	Case study A – Voting Model . . . . .	27
4.4.2	Case study B – David vs. Goliath . . . . .	30

4.5	Integrating the Chatbot on Discord . . . . .	33
<b>5</b>	<b>Upgrading the Multiagent System with a Learning Module</b>	<b>37</b>
5.1	Online Learning for Conversational Agents . . . . .	38
5.2	Weighted Majority Algorithm . . . . .	39
5.3	Setting up the Environment . . . . .	40
5.4	I Am Groot: Gauging the Learning Module’s Effectiveness . . . . .	42
5.5	Learning the Agents’ Weights . . . . .	43
5.6	Evaluating the System’s Accuracy . . . . .	45
5.7	Evaluating the Systems with Human Annotators . . . . .	48
5.8	Comparing with our Previous Results . . . . .	49
<b>6</b>	<b>Conclusions and Future Work</b>	<b>52</b>
6.1	Main Contributions . . . . .	53
6.2	Future Work . . . . .	53
<b>A</b>	<b>Corpus of Interactions Gathered Through Discord</b>	<b>56</b>
<b>B</b>	<b>Basic Questions used in the Evaluations</b>	<b>63</b>
<b>C</b>	<b>Complex Questions used in the Evaluations</b>	<b>66</b>



# List of Figures

2.1	Interacting with Say Something Smart . . . . .	8
2.2	Comparing SSS and Edgar when answering personal information requests . . . . .	11
3.1	Retrieval-based STC System Architecture, extracted from [11] . . . . .	18
3.2	Gunrock Framework Architecture, extracted from [13] . . . . .	19
4.1	SSS Multiagent Architecture . . . . .	24
4.2	Interacting with the improved SSS in Discord, using Edgar as its Persona . . . . .	34
5.1	Evolution of the weight bestowed to each agent through 30 iterations of learning. . . . .	43
5.2	Evolution of the weight given to each agent throughout the 18000 iterations of training. . . . .	44



# List of Tables

2.1	Results of the redundancy measure evaluation . . . . .	10
4.1	Magarreiro’s system against our system when answering basic questions . . . . .	29
4.2	Magarreiro’s system against our system when answering complex questions . . . . .	29
4.3	Edgar evaluated against our system when answering questions about the Monserrate Palace	32
4.4	Edgar evaluated against our system when answering out-of-domain questions . . . . .	33
5.1	Weight of each agent after 30 iterations in the Groot experiment. . . . .	43
5.2	Weight of each agent throughout the 18000 iterations of training. . . . .	45
5.3	Accuracy of the systems when evaluated with 2000 interactions of the CMD corpus. . . .	47
5.4	Mendonça et al.’s system against our trained system when answering basic questions. . .	48
5.5	Mendonça et al.’s system against our trained system when answering complex questions. .	49
5.6	The four systems compared when answering basic questions . . . . .	49
5.7	Magarreiro’s system against our system when answering complex questions . . . . .	50





# Acronyms

<b>SSS</b>	Say Something Smart
<b>T-A</b>	Trigger-Answer
<b>CMD</b>	Cornell Movie Dialogs
<b>TF-IDF</b>	Term Frequency - Inverse Document Frequency
<b>API</b>	Application Programming Interface
<b>REST</b>	Representational State Transfer
<b>JSON</b>	JavaScript Object Notation
<b>NLP</b>	Natural Language Processing
<b>QA</b>	Question Answering
<b>STC</b>	Short-Text Conversation
<b>LCS</b>	Longest Common Subsequence
<b>EWAF</b>	Exponentially Weighted Average Forecaster



# 1 Introduction

## Contents

---

1.1 Motivation . . . . .	2
1.2 Objectives . . . . .	3
1.3 Document Organization . . . . .	3

---

## 1.1 Motivation

Technology advancements through the twentieth century brought the development of chatbots, software programs that are capable of interacting with people through natural language. In today's world, chatbots are ever more present in our daily lives, with sophisticated platforms such as Alexa [1] taking the role of personal assistants and being able to carry requests such as giving information regarding a specific subject when prompted, but also make conversation if the user wishes for a more casual interaction. Chatbots are also prominent as virtual assistants, being able to accurately clarify questions regarding a certain domain, or to carry out requests such as ordering a pizza.

Typically, each virtual assistant has a single agent behind it, deciding the answer to deliver to the user, and in the case of systems that have multiple agents, each agent is delegated distinct tasks, with a user request's answer being delivered by a single agent without being considered by the others. Outside of the virtual world, however, each person has its strengths and weaknesses: no one is excellent at everything, and a person who is better at a certain domain may be weaker in another. For example, suppose the case of a library managed by three librarians: one of them is a person who knows everything there is to know about the Mystery and Romance sections, but has only very superficial knowledge about the other sections; the second librarian is someone who knows a bit about every section except for the French Literature section; and the third librarian is someone who is new at the job, but is extremely enthusiastic about Mystery and Adventure books as a reader.

One may argue that, when requesting a novel written by Agatha Christie (ergo, a Mystery book), there's no point in speaking to the second librarian, as the first one will certainly have a greater knowledge of Mystery books, but the third librarian will also be able to give their own legitimate insights, perhaps even better than the first librarian. What if you are looking for a cook book about traditional desserts? While none of the librarians are specialized in that subject, or in other words, the subject of desserts is not their domain, they may be able to help you find that book, as they know the library better than anyone.

Bearing this in mind, we propose a plug-and-play architecture that allows for each agent's answer to be taken into account and, through a decision strategy, decides which answer it should deliver to the user. Agents can be implemented externally or internally within the system, with a Manager module building the bridge between the system's agents and the external agents. We also show how to implement three types of agents with varying parameters that take external corpora into account, as well as a domain-oriented agent.

Two primary decision strategies are also presented: a Voting Model, which implements a majority vote between all of the agents and picks the most voted answer, and a Priority System, that prioritizes the answer of an agent over the others, but consults the other agents if the prioritized agent cannot give a response. Both of these decision systems can restrict each agent to give a single answer to a given query, or allow them to deliver multiple answers of equal value.

Furthermore, we present two case studies: in the first one, we evaluate our system using the Voting

Model against a single-agent system oriented to answer out-of-domain queries, and, in the second case study, we evaluate our system’s performance when answering both domain-specific queries and out-of-domain requests by prioritizing a specialized agent.

Finally, we integrate a sequential learning algorithm into our system in order to learn which agents perform best, and we evaluate it against past work done on Say Something Smart (SSS), a dialogue engine developed with the aim of answering out-of-domain requests. We also incorporate our system with a social chat platform in order to improve accessibility and gather interactions from users all around the world, as well as continuously improve the results of our chatbot.

## 1.2 Objectives

Taking into account the motivations described, our main objectives are:

1. To develop a multi-agent dialogue system, in order to improve the current single-agent architecture provided by SSS;
2. To build a plug-and-play architecture that allows any feature-based agents to be implemented in the system without being bound by the system’s limitations. Previously, SSS was limited to a single agent that, although it took multiple features into account, could only deliver a single answer. We aim to develop a model that allows multiple agents to examine the retrieved candidates and each deliver their own answer to a given query;
3. To implement a decision system that accurately judges the given answers by each agent, and to implement a sequential learning algorithm in order to allow the system to figure out which agents perform best.

## 1.3 Document Organization

This document is organized as follows: Section 2 describes background work surrounding the development of SSS and other systems that directly interact with our work. Section 3 explores related work surrounding state-of-the art indexers, similar architectures to SSS and existing feature-based agents. Section 4 presents our system’s architecture proposal and our case studies, displaying how our system improves on the provided background work. Section 5 presents the implementation of the learning module of our system, as well as its evaluation against SSS’s baseline. Section 6 explains the conclusions of our work and proposes future work for a further iteration of our system.



## 2 Background

### Contents

---

2.1	The SubTle Corpus . . . . .	6
2.2	Say Something Smart . . . . .	7
2.3	SSS's Redundancy Bug . . . . .	8
2.4	Edgar . . . . .	10

---



This section describes the background work originally done for the development of Say Something Smart [15]. We specify the creation of the SubTle Corpus [7], SSS’s previous architecture and the work developed on Edgar [10].

## 2.1 The SubTle Corpus

The SubTle corpus is a corpus created by Ameixa et al. for the purpose of handling out-of-domain interactions through Say Something Smart, composed of a large collection of movie subtitles. The creation process of this corpus can be roughly divided into two parts: the collection of the subtitles, and the consequent division of those subtitles into Trigger-Answer (T-A) Pairs, which are composed by the statement spoken by a given character and its corresponding answer.

```
SubId - 1
DialogId - 320
Diff - 1608
T - Hey!
A - Get in, loser. We're going shopping.
```

In order to collect the subtitles, the authors focused on movies of the four following genres: Horror, Sci-fi, Western and Romance. The subtitles were provided by Opensubtitles’s<sup>1</sup> administrator.

Once the subtitle files had been gathered, T-A pairs were generated and filtered in order to try and fulfill the following conditions: a subtitle from one movie character must be followed by a subtitle from a different character (that is, the pair of subtitles must represent a dialogue between two characters), and the pair of subtitles must be separated by a time difference reasonable enough for it to represent a dialogue.

While the subtitles did not provide this information, certain lexical cues can be followed in order to estimate if two subtitles represent a dialogue, such as verifying if the trigger statement and the answer statement are independent sentences: if said sentences start with a capital letter and end with a punctuation mark, then they are considered to be independent sentences.

In contrast, if the trigger statement begins with a capital letter but ends with characters that do not conclude the sentence, such as “...”, commas or a lowercase word, and its answer starts with similar characters, then the statements are deemed to belong to the same character, and the Trigger-Answer pair is considered to be invalid.

Regarding the time difference condition, for example, if we have a pair of sequential subtitles S1: *“It’s your turn, Heather.”*, which ended at 00:02:09 (HH:MM:SS) and S2: *“No, Heather. It’s Heather’s turn.”*, which began at 00:02:13, and the maximum time difference allowed for a subtitle pair to be considered a T-A pair is 3 seconds, then that pair would not be considered a dialogue, as there exists a 4 second difference between the two subtitles.

---

<sup>1</sup><https://www.opensubtitles.org/>

In the work developed by Magarreiro et al. [8], a configuration file was added to SubTle so that the maximum time difference could be set by the user, therefore adding the possibility of keeping all of the interactions regardless of the time difference.

While it may still have some limitations (such as orthographic errors, as not all of the subtitles are written by experts), SubTle provides a huge amount of data without requiring these interactions to be manually written, with its English corpus containing over 500,000 T-A pairs that cover a wide amount of out-of-domain subjects.

## 2.2 Say Something Smart

Say Something Smart (SSS) is a dialogue system built for the purpose of dealing with out-of-domain interactions through a retrieval-based approach using the SubTle corpus, which was described in Section 2.1.

To index and search for possible query answers in the SubTle corpus, SSS uses Lucene, which will be described in Section 3.1.1, with each entry in SubTle being described as a document in Lucene with the fields *Subtitle ID*, *Dialogue ID*, *Trigger*, *Answer* and *Time Diff* (time difference between trigger and answer), ergo, the parameters that compose a T-A Pair described in Section 2.1.

Upon receiving a user query, SSS sends that query to Lucene, which retrieves the  $N$  corpus entries whose *Trigger* field is the most similar to the user query. After the corpus entries have been retrieved, each of the entries is rated according to four measures: **Text similarity with input**, **Response frequency**, **Answer similarity with input** and **Time difference between trigger and answer**.

- Text similarity with input ( $M_1$ ) is measured through the calculation of the Jaccard similarity [5] between the user query and the trigger, as entries whose questions are more similar to the user input are more likely to give plausible answers.
- Response frequency ( $M_2$ ) evaluates the redundancy of each extracted entry by computing the Jaccard similarity between each pair of extracted answers.
- Answer similarity with input ( $M_3$ ) verifies each answer's similarity to the user query, once again, using the Jaccard similarity formula.
- Finally, the time difference between trigger and answer ( $M_4$ ) gives an inversely proportional score to each entry according to the time elapsed between a T-A pair, that is, an entry with a *Time Diff* value of 100 *ms* will have a higher score for this measure than an entry with a *Time Diff* value of 1500 *ms*.

The final score of an Answer  $A_i$  given the user query  $u$  used to retrieve it corresponds to the sum of the scores obtained for each of the four measures  $M_j, j = \{1, 2, 3, 4\}$  when evaluating the answer's T-A pair  $(T_i, A_i)$ , each multiplied by the corresponding weight  $w_j$  assigned to that measure, and the answer

corresponding to the entry with the highest final score is delivered to the user. The scoring formula is described below:

$$score(A_i, u) = \sum_{j=1}^4 w_j M_j(T_i, A_i, u)$$

In other words, the score of each T-A pair is defined by how similar they are to the user query through a given measure, proportionally to how important that measure is.

A configuration file is provided to the user, which allows for the configuration of the weights for each measure, as well as to define the number of  $N$  responses to retrieve per query and to configure other parameters such as the path to the corpora.

Upon interacting with SSS, it was able to deliver plausible answers to several of our requests, such as “*Do you like sushi?*”, but struggled with some of the requests either because its answer was dependent on previous context (for example, the answer to the question “*What’s your phone number?*”), or because the request itself was domain-specific, as shown with the answer to “*Can you tell me how to get to the Humberto Delgado Airport?*”. The result of these interactions is shown in Figure 2.1.

```
Say something:
Do you like sushi?
Answer: Sushi... I hate the stuff.

Say something:
What's your phone number?
Answer: Talk to me before you talk to Frank, okay?

Say something:
How are you?
Answer: Good, and you?

Say something:
Can you tell me how to get to the Humberto Delgado Airport?
Answer: Can you tell me how to get to--
```

Figure 2.1: Interacting with Say Something Smart

### 2.3 SSS’s Redundancy Bug

Upon using SSS in subsequent projects, we discovered that the system was not taking redundancy into account. There are two main causes for that issue, with the first being the fact that Lucene does not use the frequency of an answer as a component when retrieving the best results, and the other one being due to SSS’s redundancy metric M2 only comparing the similarity between the best  $N$  answers, and not with all the answers given for a certain query.

1. After indexing the SubTle corpus, upon receiving a user query, Lucene gives a score to each T-A pair in its index according to the similarity between the user query and the trigger. For example, if the pairs  $P1$ : ( $T$  - “*Are you okay?*”,  $A$  - “*Yeah.*”),  $P2$ : ( $T$  - “*Are you okay?*”,  $A$  - “*I’m fine.*”) and  $P3$ : ( $T$  - “*Are you alright?*”,  $A$  - “*Yeah.*”) are inserted in Lucene’s index, and a user query

containing the question “*Are you okay?*” is given to Lucene, then Lucene’s scorer will give the same score to P1 and P2, given that both of those pairs contain the same trigger (which is also equal to the user query’s content), and a lower score to P3 (whose trigger differs slightly from the user query).

2. On its own, Lucene does not take redundancy into account. That is, if Lucene indexes a corpus composed by 100 documents, of which 99 of them are the T-A pair containing the tuple (*T* - “*Mitchel, are you okay?*”, *A* - “*I’m fine.*”), and the last document contains the tuple (*T* - “*Mitchel, are you okay?*”, *A* - “*Of course not.*”), Lucene will treat all 100 entries as independent documents, and if we give Lucene the query “*Mitchel, are you okay?*”, Lucene will give the same score to every document, therefore not distinguishing repeated documents from individual documents.
3. In the work developed by Magarreiro et al. [8], the metric M2 evaluated the frequency of the best responses extracted by Lucene using a similarity algorithm. For example, if Lucene extracts the best 20 trigger-answer pairs for a user query “*Are you okay?*”, in which 19 of them have the sentence “*I’m fine*” as their answer, and the last one has “*I’m not fine*” as its answer, then the last pair will still have a high redundancy score despite the sentence only appearing once, as “*I’m not fine*” shares two words with the sentence “*I’m fine*”.
4. As referred above, in order to ease the decision process, upon receiving a query, SSS asks Lucene for the *N* best responses to that query, and consequently decides the best answer to the query among those *N* responses through SSS’s similarity metrics.

Looking at each of these elements on their own, there are no apparent issues:

- It makes sense for SSS to compare the contents of the user query with the indexed questions, as triggers similar to the user input are more likely to give plausible answers.
- Lucene’s job is not to detect redundancy, but to extract the results that match a given query.
- Answers with similar content tend to have similar meaning, thus, it is not the redundancy metric (M2) by itself that is causing the problem.
- Using an indexer to extract the best answers is also reasonable, as most of the documents in the corpus will be meaningless given a user query, and an indexer pinpoints the best answers between the reasonable ones.

The redundancy issue happens when we merge those four facts together. Take, for example, the following scenario, with a corpus composed by 100 Trigger-Answer pairs:

- 66 are *P1*: (*T* - “*Mitchel, are you okay?*”, *A* - “*I’m fine.*”);
- 33 are *P2*: (*T* - “*Mitchel, are you okay?*”, *A* - “*Of course not.*”);
- The last pair is *P3*: (*T* - “*Mitchel, are you okay?*”, *A* - “*Of course I’m not fine.*”).

Lucene receives this corpus and indexes it, assigning a numeric index to each document by the order in which they are found in the corpus file, such as:

(1): (*T* - “*Mitchel, are you okay?*”, *A* - “*Of course I’m not fine.*”)

(2): (*T* - “*Mitchel, are you okay?*”, *A* - “*I’m fine.*”)

(3 - 35): (*T* - “*Mitchel, are you okay?*”, *A* - “*Of course not.*”)

(36 - 100): (*T* - “*Mitchel, are you okay?*”, *A* - “*I’m fine.*”)

As such, the first two index entries correspond to P3 and a single iteration of P1, the entries between 3 and 35 are all of the entries for P2, and the remaining entries are from P1.

When SSS receives the query “*Mitchel, are you okay?*”, it asks Lucene for the 20 best matches to that query. To extract the best matches to a query, Lucene gives a score to each of the 100 pairs based on the similarity between the user query and the trigger. As all of the pairs have the same trigger (being “*Mitchel, are you okay?*”), the same score is given to each one and the best 20 responses extracted by Lucene correspond to the first 20 entries in its numeric index, which in turn correspond to 1 P1 pair, 18 P2 pairs and 1 P3 pair.

The obtained results in this test were the following:

	Corpus Frequency	Best Responses Frequency	Redundancy Score
P1	66	1	0.0227
P2	33	18	1
P3	1	1	0.6364

Table 2.1: Results of the redundancy measure evaluation

Upon receiving the responses from Lucene, SSS’s response frequency metric M2 evaluates the most frequent answer between the 20 extracted entries, which results in P2’s answer “*Of course not*” to be rated as the most redundant regardless of it only appearing 33 times in the possible answers prior to Lucene’s retrieval, while P1’s answer “*I’m fine*” is ignored despite appearing 66 times in the corpus. Additionally, P3 obtains a redundancy score quite higher than P1, despite only appearing a single time in the entire corpus.

## 2.4 Edgar

Edgar (Fialho et al., 2013) is a conversational platform developed for the purpose of answering queries related to the Monserrate Palace. Additionally, it also contains an out-of-domain knowledge source in order to answer out-of-domain queries, composed by 152 question-answer pairs, and can answer queries regarding personal information, such as “What’s your name?” or “How old are you?” as shown in Figure 2.2.

To be able to answer queries, Edgar relies on a manually created corpus of Question-Answer pairs regarding the Monserrate Palace, personal questions and, as mentioned above, certain out-of-domain

topics as well. Multiple questions with the same meaning are associated to the same answer, as to attempt to cover the greatest amount of possible wordings a user can employ when formulating its query.

For instance, in Edgar’s corpus, the questions “*Em que ano é que foi construído o palácio de Monserrate?*”, “*Que idade tem o palácio de Monserrate?*”, and “*Quando foi construído o palácio?*” all have the same answer: “*O palácio foi construído entre 1858 e 1864, sobre a estrutura de um outro palácio.*”

However, this set of question-answer pairs is not sufficient to answer most of the issued out-of-domain queries, and as such, for most out-of-domain questions, Edgar will either not be able to answer the question, or give an answer that does not make sense. This is a significant issue, as users tend to get more engaged with conversational platforms when plausible answers to small-talk queries are delivered, and will be one of the major focus points of our work. On the other hand, systems akin to SSS often struggle with answering queries for personal information, with their answers often contradicting each other as shown in Figure 2.2.

```
Say something:
What's your full name?
Question: What's your full name?
Answer: Michael Fratti.

Say something:
What's your name?
Question: What's your name?
Answer: I'm Rachel. Can you tell me your name?

Say something:
Who are you?
Question: Who are you?
Answer: Who dammit are you?
```

(a) Interacting with SSS

```
Say something:
What's your full name?
Question: What's your full name?
Answer: My name is Edgar Smith.

Say something:
What's your name?
Question: What's your name?
Answer: My name is Edgar Smith.

Say something:
Who are you?
Question: Who are you?
Answer: My name is Edgar Smith.
```

(b) Interacting with Edgar

Figure 2.2: Comparing SSS and Edgar when answering personal information requests

From this perspective, we believe that both Edgar and SSS would greatly benefit from being embedded into the same system: Edgar would obtain the support of a system capable of dealing with out-of-domain interactions, and SSS would not only be able to accurately respond to requests from a specific domain, but would also have access to an agent that could shape a personality for the system. Edgar’s character has a name, an age and a job, as well as likes and dislikes, while SSS does not have a defined character behind it.



# 3 Related Work

## Contents

---

- 3.1 Indexers . . . . . 14
  - 3.1.1 Lucene . . . . . 15
  - 3.1.2 Elasticsearch . . . . . 16
  - 3.1.3 Others . . . . . 16
- 3.2 Similar Architectures . . . . . 16
  - 3.2.1 DocChat . . . . . 17
  - 3.2.2 IR System Using STC Data . . . . . 17
  - 3.2.3 Gunrock . . . . . 19
- 3.3 Similarity Metrics . . . . . 20

---



This section describes relevant work studied for the subsequent development of our system. We detail state-of-the-art indexers, similar architectures to SSS, and similarity metrics used in other systems.

### 3.1 Indexers

In the context of Information Retrieval, indexing can be defined as the process of organizing a collection of documents inside a search engine according to a certain logic or criteria. An index's main purpose is to accelerate the process of searching a set of documents given a user query.

As an example, suppose that a user wants to search a large collection of documents for the word “yoghurt”. If an index does not exist, the search engine will have to check, for every document, if it contains the word “yoghurt” or not. Having an index, however, allows the search engine to access those documents directly without having to iterate through them every time a query is made.

For most search engines, the indexing process consists in two steps. The first step is to read every document in a given collection and to extract its tokens, which is often performed by an analyzer. The second step is to store these tokens inside a data structure, such as a database or a hash table, with a link to its original document, or in other words, to index the tokens. This process is standardized in most state-of-the-art indexers such as Lucene [2], Elasticsearch<sup>2</sup> and Xapian<sup>3</sup>, and is also commonly known as reverse indexing.

Upon receiving a document, the analyzer tokenizes it, transforming its fields into a set of tokens. Afterwards, the obtained tokens pass through the analyzer's stemmer, which reduces them to their root form. Finally, the analyzer removes any words that hold little information value (such as common words like “it”, “the”, which are considered stop-words) from the set of tokens.

Once an analyzed set of tokens is obtained for that document, the indexer takes each token and stores it in an inverted index with a link to its document of origin. That way, if a user desires to retrieve all the documents where a certain combination of words appear, the indexer only has to check the index entries for each of those words and deliver the documents associated to each word to the user.

For a practical example related to this thesis, suppose we wish to index two documents, represented by the following sentences, inside a given index  $L$ :

texttt

D1: “Are you enjoying the yoghurt, Mitchel?”

D2: “Mitchel, are you okay?”

The analyzer would tokenize document  $D1$  into  $[\backslash\text{are}”, \backslash\text{you}”, \backslash\text{enjoying}”, \backslash\text{the}”, \backslash\text{yoghurt}”, \backslash\text{mitchel}”]$  and  $D2$  into  $[“\text{mitchel}”, “\text{are}”, “\text{you}”, “\text{okay}”]$ , its stemmer would reduce the token “*enjoying*” in the first set into “*enjoy*”, and the stop-words (“*are*”, “*you*”, “*the*”) would be removed from both sets, leaving us with the following final sets of tokens:

---

<sup>2</sup><https://www.elastic.co/>

<sup>3</sup><https://xapian.org/>

T1: [\enjoy", "yoghurt", "mitchel"]

T2: [\mitchel", \okay"].

Each one of the tokens would subsequently be stored in index  $L$  with a reference to their original documents, as follows:

$L[\backslash\text{enjoy}] = [D1]$

$L[\backslash\text{yoghurt}] = [D1]$

$L[\backslash\text{mitchel}] = [D1, D2]$

$L[\backslash\text{okay}] = [D2]$

If a user made a query for all documents that had the words “*enjoy yoghurt*”, the indexer would only have to check the entries in index  $L$  for each of those words to find out which documents contain them, and would subsequently deliver  $D1$  as a result of that query.

### 3.1.1 Lucene

Lucene is an open-source information retrieval software library commonly used to store, index and retrieve huge amounts of data at an efficient pace. The previous iterations of SSS were built using Lucene as both its indexing and searching engine, and Lucene continues to be used by an enormous number of applications nowadays.

In SSS, Lucene fulfilled the roles of analyzing and indexing the documents provided by the SubTle corpus and, upon receiving a user query, retrieving the documents which matched that same query, prioritizing documents with higher similarity to the query. Lucene integrates a ranking algorithm in order to measure similarity between the given user query and each retrieved document.

Similarly to the indexing process previously described above for the documents, when Lucene receives a user query, the analyzer divides it into a set of tokens, passes those tokens through the stemmer, and removes any existing stop-words. Afterwards, the searcher checks the inverted index entries for each token, and subsequently retrieves the documents that match the query terms. Finally, the retrieved documents are ranked in relevance using the BM25 similarity measure [3], which is an algorithm used to rank documents according to their similarity to a given user query, and the best matches are delivered to the user.

There are a variety of possible approaches to the TF-IDF formula, but in the case of document retrieval through Lucene, given a set of documents  $D$  and a user query  $q$ , the BM25 can be represented as follows for each document  $d_j$ :

$$\text{sim}(d_j, q) = \sum_{i \in q} IDF_i \cdot TF_{i,j}$$

With  $IDF_i$  and  $TF_{i,j}$  being described as follows:

$$IDF_i = \log \frac{N - n_i + 0.5}{n_i + 0.5}$$

$$TF_{i,j} = \frac{f_{i,j} \cdot (k_1 + 1)}{f_{i,j} + k_1 \cdot (1 - b + b \frac{|d_j|}{avgdl})}$$

$N$  corresponds to the total number of documents in set  $D$ ,  $n_i$  represents the number of documents in  $D$  containing the term  $i$ ,  $f_{i,j}$  is the frequency of the term  $i$  in document  $j$ ,  $avgdl$  is the average document length for set  $D$  and both  $k_1$  and  $b$  are arbitrary parameters, often having default values of  $k_1 = 1.2$  and  $b = 0.75$ .

### 3.1.2 Elasticsearch

Elasticsearch<sup>4</sup> is a distributed search engine that uses Lucene for its core operations, and provides a web server on top of Lucene. It is renowned for its speed and scalability, as well as ease of use.

In order to deal with indexes of enormous sizes, Elasticsearch allows for an index to be divided into shards. Each shard is a Lucene index, and upon receiving a query, Elasticsearch can perform search operations on multiple shards at the same time, which in turn results in better performance results.

Unlike Lucene, which depended on its *IndexSearcher* class API, user requests to Elasticsearch’s cluster are made through its REST API. For instance, the query “GET test1/\_search?q=Trigger:Jeremy” would search the index ‘test1’ for all documents containing the term ‘Jeremy’ in their ‘Trigger’ field. In response, Elasticsearch delivers a JSON string containing the query hits, as well as any relevant information for the query such as the score given to each hit.

### 3.1.3 Others

Similarly to Elasticsearch, Apache Solr<sup>5</sup> is a search engine that uses the Lucene API at its core for its indexing and search operations. In comparison to Elasticsearch, Solr depends on Apache Zookeeper<sup>6</sup> to maintain a server cluster, whereas Elasticsearch only depends on its own nodes. In terms of functionality and performance, both Elasticsearch and Solr rely on the Lucene API for their core operations.

The Xapian Project<sup>7</sup> is an open source search engine with support for ranked search, allowing for the use of most basic NLP operations as well. However, its core architecture involves the use of a database, and having both Lucene and Elasticsearch as alternatives, we decided to discard the possibility of using Xapian in this project.

## 3.2 Similar Architectures

This section describes existing systems whose architecture is similar to our proposal.

---

<sup>4</sup><https://www.elastic.co/>

<sup>5</sup><http://lucene.apache.org/solr/>

<sup>6</sup><https://zookeeper.apache.org/>

<sup>7</sup><https://xapian.org/>

### 3.2.1 DocChat

DocChat [12] is a system which, given a set of unstructured documents, finds a response to an user query by evaluating all possible sentences through a multitude of features and selecting the best-ranked response candidate. This system is similar to SSS in the manner that both DocChat and SSS are based on information retrieval; however, unlike SSS, DocChat can be better defined as a QA system, as it is centered on giving answers to information queries in natural language.

Upon receiving a query  $Q$ , DocChat’s work loop consists in three steps: response retrieval, response ranking and response triggering. Response retrieval consists in finding sentences as close as possible to  $Q$ , which will be considered the response candidates. The BM25 ranking function [3] is used to find the most similar sentences to the query and a set of sentence triples  $(S^{\text{prev}}, S, S^{\text{next}})$  is retrieved, with each set containing a sentence  $S$  and both its previous and next sentences.

After the response candidates are gathered, the ranking  $Rank(S, Q)$  of each response is calculated through the sum of several similarity features, each with a given weight. The features presented in DocChat vary between word-level, phrase-level, sentence-level, document-level, relation-level, type-level and topic-level, and the highest-ranked response is selected for retrieval.

Finally, after that highest-ranked response is selected, the response triggering step is performed. In this step, the system calculates three parameters to check if the selected response is an adequate answer to the user query, and if the query itself is adequate for the system to answer. The first parameter checks if the given query is an informative query. For that purpose, DocChat incorporates a classifier which classifies queries into *informative queries* or into *chit-chat queries*, depending on the contents of the query. The second parameter checks if the score  $s(S, Q)$  given to the selected answer is higher than a threshold  $\tau$ , with  $s$  being calculated by:

$$s(S, Q) = \frac{1}{1 + e^{-\alpha \cdot Rank(S, Q)}}$$

with both variables  $\alpha$  and  $\tau$  being set by the user depending on the type of dataset.

The third parameter evaluates if the selected answer’s length is lower than a given threshold, and if the answer does not begin with expressions such as “besides”, “however”, “as such”, which indicate dependency on previous context.

Summing it up, if a given query  $Q$  is considered an informative query, its response candidate  $S$  has a similarity score towards  $Q$  higher than a certain threshold, and that same response candidate  $S$  is an independent sentence (that is, if it does not depend on previous context), then  $S$  shall be given as a response to the user query  $Q$ .

### 3.2.2 IR System Using STC Data

In August 2014, Ji et al. [11] proposed a conversational system based on information retrieval with the goal of addressing the problem of human-computer dialogue through the use of short-text conversation

(STC) data gathered in the form of post-comment pairs from social media platforms such as Twitter<sup>8</sup>.

This system’s architecture is represented in Figure 3.1. Given a query, the system retrieves a number of candidate post-comment pairs according to three metrics: similarity between the query and the comment, similarity between the query and the post, and through a deep-learning model that calculates the inner product between the query and the candidate response [16].

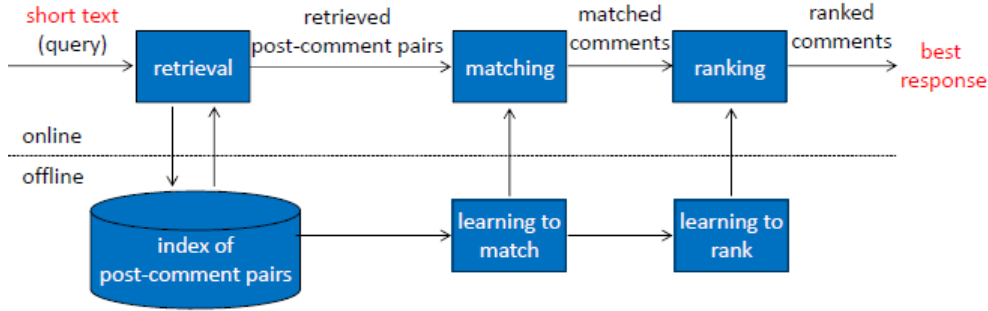


Figure 3.1: Retrieval-based STC System Architecture, extracted from [11]

After the retrieval of the candidate post-comment pairs, a set of matching models is used to evaluate the post-comment pairs, returning a feature set for each candidate pair. The presented models are:

- A translation-based language model (TransLM, Xue et al. [17]), which is used to mitigate the lexical gap between a given query and its candidate response. This is done through the estimation of unigrams between the words of the query and both the post-comment pair and the entire collection of possible responses, and through the estimation of the word-to-word translation probabilities for each word in the post-comment pair to be translated into a word in the query.
- A model based on neural networks (DeepMatch), which is built through the modelling of pairs of words that are related, and the training of a neural network with a given ranking function. The similarity between the query and the post-comment pair is, therefore, given by the perceived relation of their words in the neural network.
- A topic-word model which focuses on learning the most important words of a post-comment pair based on the following features: the position of the word, the word’s corresponding part-of-speech tag, and if the word is a named entity or not. The model is trained using manually labeled data, and the similarity between a query and a post-comment pair is given by the probability of each of their words referring to a similar topic.

Finally, the system uses the obtained feature set to calculate a score for each candidate post-comment pair according to the formula below, and delivers the highest-scored response to the user.

$$score(q, (p, c)) = \sum_{i \in \Omega} w_i \Phi_i(q, (p, c))$$

<sup>8</sup><https://twitter.com/>

For a given query  $q$ , the score of each post-comment pair  $(p, c)$  is given by the sum of the scores of each feature multiplied by its respective weight, with  $\Phi_i(q, (p, c))$  representing the score of the feature, and  $w_i$  representing its corresponding weight.

This system shares a number of similarities to SSS, as both are retrieval-based systems that use conversational data to answer user queries, and both deliver a single response to the user through the combination of several features.

The main difference in the two is the complexity of the features used to choose an appropriate answer: whereas SSS accounted only for the redundancy of a candidate response given a query and the lexical similarity of the query and the candidate response, Ji’s system implements three complex models based on word-to-word translation, deep learning, and classification respectively.

### 3.2.3 Gunrock

Regarding multiple-agent approaches, the Alexa Prize 2018 winner, Gunrock (Chen et al. 2018) [13], employs an open-domain conversational engine through a generative model, in contrast to our retrieval-based approach.

Gunrock’s architecture is focused on identifying the intents and context behind a user request, and consequently assigning a dialogue module to provide information regarding the answer to that user request. Each dialogue module is specialized in a specific topic (such as sports, movies, or casual conversation), and the main challenge in this framework is to determine which dialogue module should feed information to the answer, as opposed to our architecture, which focuses on deciding the best answer after receiving an answer from every module. That is, while Gunrock takes a linear approach to answer a query, where only one of the available modules is chosen and all of the other modules are ignored, our system focuses on a branching approach, where each agent contributes to the formulation of the final answer.

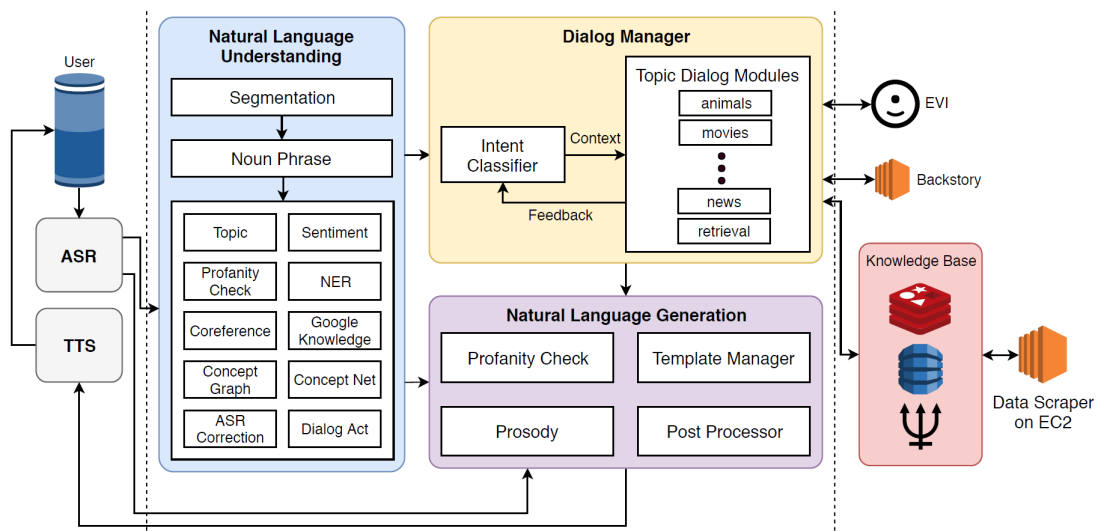


Figure 3.2: Gunrock Framework Architecture, extracted from [13]

As mentioned above, Gunrock was evaluated in the Alexa Prize competition, where it managed to achieve an engaging conversation for an average of 9 minutes and 59 seconds when interacting with Alexa Prize’s judges.

### 3.3 Similarity Metrics

As described earlier in section 2.2, SSS uses four metrics to score each candidate answer. However, with the exception of measure M4 (time difference between trigger and answer), all of the measures are built around the Jaccard similarity coefficient.

In 2016, Fialho et al. [19] participated in “Avaliação de Similaridade Semântica e Inferência Textual” (ASSIN) for the task of semantic similarity and textual entailment recognition. The proposed system is based around the extraction of multiple features from the given pairs of sentences, and the subsequent training of classification models based on said features. For this work, we will be focusing on the extracted features.

The string-level similarity features are as follows:

1. *Longest Common Subsequence (LCS)*, that is, the size of the LCS between two sentences. It is normalized through the division of the size of the LCS by the length of the longer sentence.
2. *Edit Distance*, which corresponds to the minimum edit distance between the words of two given sentences.
3. *Length*, comprising three features: the difference in length between the two sentences, the maximum length of a sentence, and the minimum length of a sentence. Note that, in this case, length refers to the number of words and not the number of characters in a sentence.
4. *Cosine Similarity*, which translates into the cosine similarity between two given sentences, with the vector of each document representing the term frequency of each word.
5. *Jaccard Similarity* between two sentences, which can be defined by the number of words that are common to both of the sentences divided by the number of words that appear in at least one of the sentences.
6. *Soft TF-IDF Similarity*, which measures the similarity between two given sentences when represented as vectors while taking internal similarity into account.

For paraphrase identification, the following metrics were used:

1. *BLEU* [20] is given by the sum of all n-gram matches between two given sentences, and penalizes sentences which are considered too long or too short.
2. *METEOR* [21] is a metric developed to address the issues of BLEU. It is based on the combination of the precision and recall through a harmonic mean, with most of the weight being supported by recall.

3. *TER* [22], or Translation Edit Rate, is measured by computing the number of edits that, given two sentences, are necessary to transform one of the sentences into the other.
4. *NCD*, or Normalized Compression Distance, is a metric used to measure the similarity between two objects (in our case, between two sentences). If two sentences are compressed, only the information common to both of them will be extracted.
5. *ROUGE* [23] is a metric based on n-gram occurrence statistics, with two variations presented in INESC-ID@ASSIN's work: one focusing on the length of the longest common subsequence, and the other focusing on skip-bigrams.

For similarity grading, among the most relevant features were the Cosine Similarity, Soft TF-IDF, Jaccard and the ROUGE variations. Given their good results, we decided to implement some of these features as agents in our work.





# 4 Upgrading SSS into a Multiagent System

## Contents

---

- 4.1 Proposed Architecture and Workflow . . . . . 24
- 4.2 Building the Agents . . . . . 25
- 4.3 Building the Manager . . . . . 26
- 4.4 Building the Decision Maker . . . . . 26
  - 4.4.1 Case study A – Voting Model . . . . . 27
  - 4.4.2 Case study B – David vs. Goliath . . . . . 30
- 4.5 Integrating the Chatbot on Discord . . . . . 33

---

This section describes our system’s proposal and architecture, focusing on the enhancement of Say Something Smart into a multiagent system, as well as the integration of domain-oriented agents such as Edgar.

## 4.1 Proposed Architecture and Workflow

Most approaches to conversational engines tend to either focus on constructing a single agent that evaluates and decides the answer to any given request, or building multiple agents with distinct domains and directing the task of answering a query to one of them, depending on the domain of that query. We seek to challenge both of these approaches, and, instead, opt for the reverse: our system will assume that all available agents can answer a given question, and the most appropriate answer to deliver to the user will be determined through a voting model.

For that purpose, we built a plug-and-play framework that allows the implementation of any agent, regardless of the formal process through which the agent reaches its answer. Being a retrieval-based conversational engine, our framework is assisted by Lucene regarding the indexing and searching operations on its corpus. The proposed architecture is further represented in Figure 4.1.

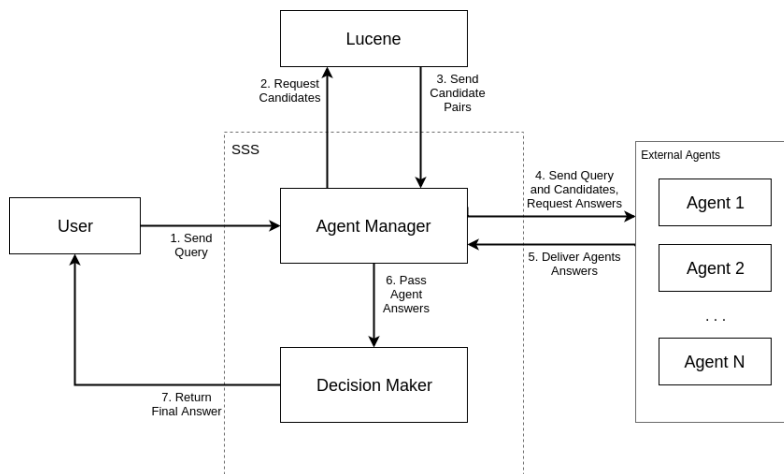


Figure 4.1: SSS Multiagent Architecture

In the case of our system, Lucene is charged with the operations of analyzing and indexing all entries of the given corpus, as well as the search and retrieval operations over the created index. As described in Section 3.1.1, Lucene employs a ranking algorithm named BM25 [3] to measure the similarity between a given query and each one of its matching index documents, which takes into account not only the common words between the query and each document, but also the relative frequency of a word in the entire corpus. The implementation of this last metric implies that words that appear less in the corpus will have greater weight, while words that are more common (such as stopwords) will have less weight regarding the final rank of a document.

Following the protocol set by SubTle and shown in Section 2.2, each corpus is treated as a set of Trigger-Answer pairs that we will refer to as the *candidates*. When a user query is received, our

conversational engine redirects that query to Lucene, which will search for the possible *candidates* in the previously indexed corpus using the keywords given by the user query, and then rank each candidate through the BM25 similarity measure, which computes the similarity between the user query and the interaction parameter of each candidate.

The highest-ranking candidates are subsequently delivered to our framework and sent to each available agent alongside the user query. Given the user query and the candidates retrieved by Lucene, each agent computes its answer to the query using its own algorithm, and submits that answer to the *Agent Manager*.

The answers of the agents are passed to the *Decision Maker*, which, given a strictly defined decision method, decides on the best answer to deliver to the user. Our proposed decision methods are a *Voting Model*, which selects the most common answer given by the agents, and a *Priority System*, which, given a set of defined priorities for each agent, delivers the response of the agent with the highest priority that is able to answer the query. If there is more than one highest-rated answer, the first answer to enter the system will be delivered to the user, and if no agents can answer the query, the system will tell the user “*I do not know how to answer that.*”.

## 4.2 Building the Agents

In the context of this framework, an agent is defined as a piece of software that, upon receiving an user query, delivers an answer to that query through a given algorithm. An agent may use provided resources such as external corpora, part-of-speech taggers, or other necessary means to reach its final answer.

In order to build a new agent for our system, two components are needed: a configuration file, and the source files of the agent.

The configuration file serves as the “header” of the agent for our system: it allows the agent to be called by the system’s *Agent Manager*, and it also allows the user to set configurable parameters without directly interacting with the source files. On the other hand, the source files are the core of the agent: they receive the queries (and all of the needed resources) from our system, and promptly deliver an answer.

Below is an example of a simple configuration file for an agent named *MixAgent*:

```
1 <config>
2   <mainClass>MixAgent</mainClass>
3   <receiveLuceneCandidates>true</receiveLuceneCandidates>
4   <questionSimValue>0.5</questionSimValue>
5   <answerSimValue>0.5</answerSimValue>
6 </config>
```

The *mainClass* parameter indicates the name of the agent’s callable class; the *receiveLuceneCandidates* parameter tells the *Agent Manager* to send the generated candidates in order to assist the decision process, and both the *questionSimValue* and *answerSimValue* are parameters used in *MixAgent*’s internal algorithm when generating the agent’s answer, which correspond to the similarity between a user query

and a received candidate's question or answer, respectively.

In this case, *MixAgent* will receive the user query and the Lucene candidates, and calculate the Jaccard Similarity between both the Trigger (question) and the Answer components of the query and the candidates, giving an equal weight (0.5) to both.

### 4.3 Building the Manager

The *Agent Manager* is a module built with the aim of providing an interaction point between our system and the provided agents. It is responsible for creating the agents, and for guaranteeing that the communication between the system and the agents is done correctly.

When our system is booted up, the *Agent Manager* locates all of the available agents through their configuration files and integrates an instance of each of the agents into the system. If the system's user wishes to disable certain agents, that option is also provided through a text file named `disabledAgents.txt`, which contains the names of the agents to be disabled.

Upon receiving an user query, this module contacts each agent and verifies if they have specific needs. For example, if a certain agent needs its response candidates to be generated from a separate corpus rather than the default one, the *Agent Manager* contacts Lucene in order to provide the candidates with the specified corpus. The response candidates are subsequently generated, and both the user query and those candidates are sent to each agent in order to gather all of the possible answers.

Finally, after having sent the user query and the candidates, the *Agent Manager* receives and stores the answer of each agent to allow the proper identification of each agent upon the evaluation of the final answer to deliver to the user. At this point in time, our system has obtained a set of answers from all of the available agents, and is ready to evaluate the best answer to deliver to the user.

### 4.4 Building the Decision Maker

As introduced earlier, the *Decision Maker* is the module that holds the role of deciding the final answer delivered to the user when given a set of possible answers generated by the system's agents. There are two primary decision methods that the *Decision Maker* can employ: the *Simple Majority* and the *Priority System*.

When using the *Simple Majority*, the *Decision Maker* will deliver the most frequent answer between the set of answers received from the agents. This decision method is based in two principles: each agent is able to accurately answer most queries, and a plausible answer is likely to be shared by multiple agents.

In order to exemplify the use of this decision method, let us suppose the following scenario:

- A user sends the query "How are you?", which is received by agents A, B, C and D.

- Agents A and B return “*I’m fine.*” as their answer to that query, while agent C delivers “*You are beautiful.*” and agent D delivers both the answers “*I’m fine, thank you.*” and “*Mind your own business.*”.

Under the *Simple Majority* decision method, the answer “*I’m fine.*” would be delivered to the user, as it is the answer given by the greatest amount of agents.

#### 4.4.1 Case study A – Voting Model

So far, we have established and proposed a conversational architecture that supports the inclusion of multiple agents, each working on its own to provide an answer when faced with an user query.

In order to test a multiagent environment, it is only natural to build various unique agents. Given our access to external corpora and the available assistance provided by Lucene regarding the retrieval of possible answers, we decided to focus on agents that used the properties of the generated QA Pairs to reach an answer. Therefore, based on the results gathered by Fialho et al. (2016) [19], we chose three distinct lexical features and built agents based on them: *Cosine Similarity* [4], *Jaccard Similarity* [5] and *Levenshtein Distance* [6].

SubTle, the corpus composed by Trigger-Answer pairs of movie subtitles described in Section 2.1, was used for the purpose of this experiment as to fit the criteria of an information source with any type of content, not being limited to a specific topic. Each entry of the corpus was indexed by Lucene beforehand, and when faced with a user query, Lucene would retrieve the subtitle pairs whose Trigger parameter was most similar to that query.

For the *Cosine Agent*, as we are working with text, we converted the two sentences (the user query, and the candidate’s question or answer) into vectors to be compared, where each column of the vector represents the frequency of each word. When determining the cosine similarity, both sentences were lowercased and had their punctuation taken out in preparation for the vectorization process. Stopwords were kept, and the sentence’s words were not stemmed.

The *Levenshtein Agent* calculated the edit distance at the sentence-level, while keeping the punctuation of each sentence and without filtering stopwords. The evaluated sentences were lowercased, and the responses with the lowest edit distance were deemed to be the best-scored.

Finally, for the *Jaccard Agent*, each evaluated sentence was normalized through the filtering of punctuation and the lowercasing of each word (be it the user input or the Trigger/Answer parameters), and a set of stopwords was used to filter the most significant terms of each sentence before the Jaccard similarity was computed.

All agents received the user query and a set of Lucene candidates generated from the SubTle corpus. Additionally, all agents evaluated the similarity between the candidate’s question and the user query, and between the candidate’s answer and the user query. However, we decided to create three instances of each agent in order to measure the performance of the agents depending on the weight delegated to the

question similarity and answer similarity scores. *CosineAgent1*, *JaccardAgent1* and *LevenshteinAgent1* are characterized by a 50% split between the two similarity scores, *CosineAgent2*, *JaccardAgent2* and *LevenshteinAgent2* assign a weight of 75% to the question similarity and 25% to the answer similarity, and *CosineAgent3*, *JaccardAgent3* and *LevenshteinAgent3* grant a weight of 100% to the question similarity, not taking into account the answer similarity at all. Finally, our system’s Decision Method was the *Simple Majority*, which gave an equal presence to each agent.

With the system’s environment built, we decided to test it against SSS, the dialogue engine built by Magarreiro et al. (2014) described in Section 2.2, which also employed the use of the SubTle corpus as its main source of information and based itself upon lexical features to determine its answers. For that goal, a set of 100 simple questions (such as “What’s your name?” and “How are you?”) and a set of 100 more complex questions (such as “What’s your opinion on Linux-based platforms?” and “If the world ended tomorrow, what would you do today?”) created by students of the Natural Language Processing course at IST were in turn evaluated and answered by both our system and SSS.

Subsequently, four annotators were given a sample of 50 questions and answers from each of the following sets:

- Simple questions answered by our system.
- Simple questions answered by Magarreiro’s system.
- Complex questions answered by our system.
- Complex questions answered by Magarreiro’s system.

The team of annotators was composed by the author of this work and his advisor, as well as two students currently developing work related to the area of NLP. The annotators did not have the knowledge of which system had given which answer, and assigned each of the responses a score between 1 and 4 through the following criteria:

4 - The given answer is plausible without needing any additional context.

3 - The given answer is plausible in a specific context, or does not actively answer the query but maintains its context.

2 - The given answer actively changes the context (for example, an answer that delivers a question to the user which does not match the initial query’s topic), or the given answer contains issues in its structure (even if its content fits in the context of the question).

1 - The given answer has no plausibility value.

The simple questions vary from “*Olá, como estás?*”<sup>9</sup> to “*De que tipo de música gostas?*”<sup>10</sup>, mostly focusing on personal questions. On the other hand, the complex questions can range between deeply philosophical questions such as “*Porque é que a galinha atravessou a estrada?*”<sup>11</sup> or critical thinking

---

<sup>9</sup> “Hi, how are you?”

<sup>10</sup> “What kind of music do you like?”

<sup>11</sup> “Why did the chicken cross the road?”

questions like “*Quantas janelas existem em Nova Iorque?*”<sup>12</sup>.

To perform the evaluation, the mean value of all answers’ scores for each system was calculated. Additionally, the mode of all annotated values was also noted in order to better understand the nature of the obtained results. Finally, taking into consideration the evaluations done in other engines such as AliMe [9], we considered that to be discerned as acceptable to deliver to a user, a given answer would need to have an average score of at least 2.75 between the four annotations. This corresponds, for example, to the case where three annotators give a score of 3 to that answer and the last annotator gives it a score of 2, as the set of scores indicated that the answer is acceptable in most contexts according to the majority of the annotators. This metric is named the Approval rating of each system.

Regarding basic questions, our multiagent system had practically the same results as Magarreiro’s, with, in a scale of 1 to 4 (following the annotation system), the mean score of Magarreiro’s system being 2.68, and our system’s being 2.6, and Magarreiro’s system obtaining 46% approval against our system’s 48%, as shown in Table 4.1. Finally, while the scores attributed to Magarreiro’s system by the annotators were more polarized, with a special focus on scores of 2 and 4 to its answers, our system’s answer scores were more evenly distributed.

Metric	Magarreiro’s	Multiagent
Mean Score	2.68	2.6
Approval	46%	48%

Table 4.1: Magarreiro’s system against our system when answering basic questions

On the other hand, testing with complex questions yielded more interesting results, with our system performing significantly better than Magarreiro’s, as shown in Table 4.2. The mean score of Magarreiro’s system was 2.26 and its approval rate was 22%, while our system maintained its score of 2.6, with an approval rate of 44%. Both systems had the score 2 being the most commonly assigned to its answers, with a stronger preponderance for Magarreiro’s system to deliver implausible answers. This can be explained by the fact that answers to more complex questions are usually not consistent between multiple agents: while Magarreiro’s system relies on a single agent to decide on all its answers, our system takes into account what possible answers are more common from multiple points of view.

Metric	Magarreiro’s	Multiagent
Mean Score	2.26	2.6
Approval	22%	44%

Table 4.2: Magarreiro’s system against our system when answering complex questions

So far, we have evaluated two systems oriented to answer out-of-domain queries, and our system has been shown to keep up with Magarreiro’s (in the case of basic questions) or outright outperform

---

<sup>12</sup>“How many windows are there in New York City?”



it completely, as shown with the complex questions. With that done, let us see how it performs when taking into account a domain-oriented agent.

#### 4.4.2 Case study B – David vs. Goliath

Every day we pass by hundreds of people in our daily, mundane lives. We all have our own strengths and weaknesses: it is natural for people to be more skilled in a certain task than in another, and that trait of our world extends to virtual environments as well. In a multiagent environment, it is likely that each agent may have a greater aptitude to answer specific types of questions, and fall in performance regarding another specific kind of queries.

Additionally, retrieval-based platforms often have difficulty with maintaining consistency when answering queries regarding personal information, as multiple agents with different features will likely deliver different answers, even if these agents are using the same corpus to retrieve their answers.

Let us take a step back to the beginning of this chapter. Earlier, this example was introduced:

- A user sends the query *“How are you?”*, which is received by agents A, B, C and D.
- Agents A and B return *“I’m fine.”* as their answer to that query, while agent C delivers *“You are beautiful.”* and agent D delivers both the answers *“I’m fine, thank you.”* and *“Mind your own business.”*.

At the time, the answer *“I’m fine.”* was deemed the most appropriate answer by the Simple Majority. While one could argue that *“I’m fine, thank you.”* is not only a plausible response as well, but also a similar response to *“I’m fine.”*, the fact remains that only one agent delivered that answer, and thus, it is ignored in favour of the answer *“I’m fine.”*. This can, of course, also be beneficial in occasions where inappropriate responses to the query are ignored, such as the case of the answer *“You are beautiful.”*, given by agent C.

However, in a scenario where our system could guarantee that a certain agent’s answer would be accurate, even if it did not always deliver an answer to the given query, then it would make sense for that agent to be prioritized. For that purpose, we built the *Priority System*, a decision method that allows the user to explicitly set priorities for the available agents.

When an agent is given a higher priority over its peers, the system recognizes that this agent’s responses hold greater importance than the answers given by the rest of the agents. As such, when evaluating the answer each agent delivered to a given user query, our system verifies if the prioritized agent was able to deliver a response to the user query: if that agent delivered an answer, that answer is deemed to be a plausible one and is immediately returned to the user, with the answers of its peers being disregarded.

On the other hand, if the prioritized agent is not able to answer the user query, the system will verify the answers given by each of the remaining non-prioritized agents and select the final answer through

*Simple Majority*, as described earlier.

Granting priority to an agent should not be taken lightly, and it is usually recommended to be done upon agents that cover a very specific domain. Furthermore, the prioritized agent must be able to establish when it should give an answer to the user query or not. In order to counter this issue, we present a type of agent called “Persona Agents”.

Persona agents are agents that specialize in building a “character” for the chatbot and, therefore, answer personal questions such as “What’s your name?” or “How old are you?”. These agents often take priority when answering queries, and therefore, if the answer of a Persona Agent is deemed to be a proper response, the answers from the other agents are disregarded. In the event that the Persona agent cannot answer a query, then the other regular agents take its stead on delivering an answer to the user. However, a Persona agent is not necessarily a domain-oriented agent, as its main responsibility is to answer personal questions, rather than questions from a specific domain. These two concepts merge themselves in the work of Edgar, as we will see further in this chapter.

A common approach to ensure that an agent delivers an adequate answer is to set a *similarity threshold* for a certain pair of parameters. A similarity threshold defines the minimum level of similarity that the defined parameter must have to be accepted as an adequate response: if that threshold is met, then the agent delivers its best answer. Otherwise, it delivers a message of failure stating that it could not find an adequate answer.

In Section 2.4 we described Edgar, a conversational platform developed for the specific purpose of answering queries related to the Monserrate Palace. As mentioned before, Edgar struggles when providing answers to out-of-domain requests, and, for most of those requests, Edgar will either not be able to answer the question or give an answer that does not make sense. This is a significant issue, as users tend to get more engaged with conversational platforms when plausible answers to small-talk queries are delivered.

In order to address the issue regarding Edgar’s responses to out-of-domain requests, we decided to set up our system in order to use Edgar as a Priority Agent, with the nine agents described in Case Study A being used as Edgar’s back-up, in the case that Edgar was not able to provide an answer. A score threshold was also applied on Edgar in order to avoid delivering answers given by Edgar that had a low score, and through a sequence of accuracy tests, the similarity threshold of 0.35 gave the best results when discerning which questions should be answered by Edgar, compared to other similarity threshold values. Thus, our version of Edgar used the Jaccard Similarity measure, and would fail to deliver an answer to a query if its best answer had a similarity score lower than 0.35. When Edgar did not deliver an answer, the answers of the other nine agents would be evaluated, as described in the *Simple Majority* decision method.

To test our system, which we label as *Multi + Edgar*, we put it against the original Edgar, and built two sets of requests to be answered with 100 questions each. The first set was composed by out-of-domain requests, such as personal questions and trivia questions (for example, “Como te chamas?” and “Gostas de cantar?”) gathered from past interactions with Edgar in the Monserrate Palace. The second set was

composed entirely by questions about the Monserrate Palace, as to test Edgar’s accuracy on his domain and verify if results changed significantly when answered by our system (which should not be the case, as Edgar would supposedly take priority when faced with queries regarding the Monserrate Palace).

Similarly to the previous case study, four annotators were given a sample of 50 questions and answers from each of these sets:

- Out-of-domain requests answered by our system, with Edgar as a Priority Agent.
- Out-of-domain requests answered by the original Edgar.
- Monserrate Palace questions answered by our system, with Edgar as a Priority Agent.
- Monserrate Palace questions answered by the original Edgar.

Without knowledge of which system had given which answer, the annotators assigned each of the responses a score between 1 and 4 through the same criteria described in the previous case:

4 - The given answer is plausible without needing any additional context.

3 - The given answer is plausible in a specific context, or does not actively answer the query but maintains its context.

2 - The given answer actively changes the context (for example, an answer that delivers a question to the user which does not match the initial query’s topic), or the given answer contains issues in its structure (even if its content fits in the context of the question).

1 - The given answer has no plausibility value.

Once again, for the evaluation, we registered the mean value of all answers’ scores for each system and the mode of all annotated values, as well as the approval rate (with the threshold of 2.75 being used yet again).

First of all, we wished to verify if our system’s performance was significantly worse than Edgar when confronted with queries regarding Edgar’s domain, as the backup agents would, most likely, act as background noise without being able to provide plausible answers for a domain as specific as the Monserrate Palace in the case that a given question did not meet the threshold. As shown in Table 4.3, when tested with questions about the Monserrate Palace, our system had a mean score of 2.87 and an approval rate of 58%, while Edgar had a mean score of 3.015 and an approval rate of 62%, obtaining a mean score equal or greater to 2.75 in 31 answers out of the total 50. While Edgar had the best performance (as expected), our system managed to keep up with minor accuracy costs, with both systems having score 4 attributed to most of their answers.

Metric	Edgar	Multi + Edgar
Mean Score	3.015	2.87
Approval	62%	58%

Table 4.3: Edgar evaluated against our system when answering questions about the Monserrate Palace

Following that experience, and taking into account that Edgar is able to answer certain out-of-domain queries (especially queries regarding his personal information), we tested both systems with a set of out-of-domain requests. Regarding these questions, our system beat Edgar by a significant margin, with a mean score of 2.625 and an approval rate of 44% for our system, and a mean score of 2.37 and approval rate of 36% for Edgar, as presented in Table 4.4.

Metric	Edgar	Multi + Edgar
Mean Score	2.37	2.625
Approval	36%	44%

Table 4.4: Edgar evaluated against our system when answering out-of-domain questions

Note that our system was, again, giving priority to Edgar in the case that he was able to answer a certain query, as described in the earlier experiment, and yet the approval rate gain for out-of-domain queries was greater than the approval rate loss when answering queries regarding the Monserrate Palace.

## 4.5 Integrating the Chatbot on Discord

Discord<sup>13</sup> is a freeware social chat platform with millions of users around the world that allows users to integrate their own bots with the application through a freely-provided API. These bots may be used for varied purposes, such as administration tasks (managing user permissions, for example), but they are allowed to send messages. With that in mind, it is possible to make our system communicate through Discord with users from any country, anywhere in the world.

This application provides a tangible and user-friendly interface when compared to a command line, while providing the “illusion of humanity” at the same time: as users of the platform usually communicate through text, we want them to interact with a chatbot in the same way that they would communicate with any other person in their chatroom. Additionally, through its condition of a social chat platform, Discord allows multiple users to interact with the system at the same time, in opposition to the single-person environment that limited SSS to this day.

Users of the platform do not need to configure our system in order to interact with it, nor do they need to install or download any applications or files into their computer, as although Discord provides a desktop application, it can also be directly used through a web browser. Furthermore, users can send messages in Discord through their mobile devices, therefore erasing the necessity of a computer altogether.

The downsides of using Discord boil down to the constant requirement of internet in two ways. Firstly, using Discord requires an internet connection, which means that a user who wishes to interact with our system will not be able to do so if they do not have an internet access point. The most crucial downside is the fact that the system requires a dedicated machine in order to reliably serve users at any time, ergo, it should preferentially run in a dedicated server.

---

<sup>13</sup><https://discordapp.com/>

To set up a bot in Discord, the bot's host must create a Discord account and request a Token for the bot to be allowed in a Discord server. After the setup is complete and the bot is up and running, the dialogue process functions as follows:

- A user sends a message in a chatroom where the bot is present and allowed to speak;
- The user's message is caught by the bot and processed as a query, conveyed to Lucene which, as usual, gathers a set of candidates to be evaluated by each available agent;
- The system decides the final answer based on each agent's delivered answers and the configured decision method, and the final answer is sent to the chatroom where the message was received (that is, the bot replies to the user). Three reaction emojis are also added, corresponding to a green checkmark, a red cross and a gray question mark, as represented in Figure 4.2.
- After the bot's answer is sent, in order to keep a record of the dialogues, the interaction is stored in the system.

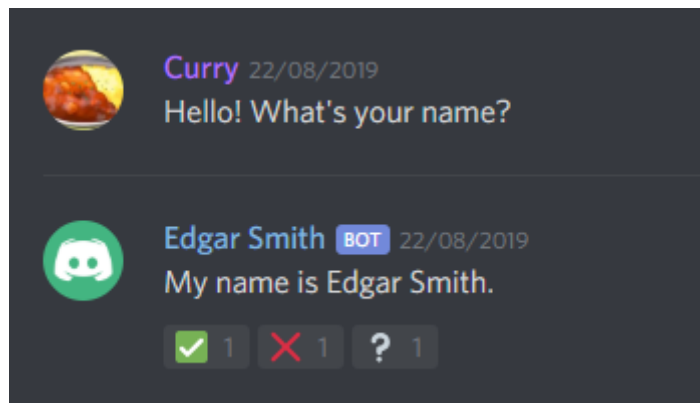


Figure 4.2: Interacting with the improved SSS in Discord, using Edgar as its Persona

An interaction is composed by the following attributes:

- **Query** – The user's original message.
- **Answer** – The bot's answer to the Query.
- **UserID** – The user's internal Discord ID.
- **PosReacts** – The amount of positive reactions (e.g. green checkmark emojis) given to the bot's answer.
- **NegReacts** – The amount of negative reactions (e.g. red cross emojis) given to the bot's answer.
- **MidReacts** – The amount of neutral reactions (e.g. gray question mark emojis) given to the bot's answer.
- **Timestamp** – The time at which the bot's answer was delivered.

- **MessageID** – The internal Discord ID of the message containing the bot’s Answer.
- **ServerID** – The internal Discord ID of the server where the message was sent.

Interactions are stored in the form of JSON<sup>14</sup> files, which can then be used for a multitude of purposes: positively rated interactions can be used to dynamically create a corpus of interactions, negatively rated interactions can filter out inappropriate dialogue options, and the context of the interactions can be taken into account, as they are stored sequentially.

For our Discord chatbot, we set Edgar as our Persona Agent (as shown in Case Study 2) and we built a bot instance in order to recognize requests (messages) sent through Discord, pass said requests to our system, and subsequently deliver the produced answer to the user.

We introduced the chatbot as Edgar in order to give it an identity and improve the engagement of users, as to simulate the experience of speaking with another person. In a preliminary experience, through our first interaction experiences with a handful of users, we noticed that users were particularly interested in asking questions regarding Edgar’s personal details, and found that the repetition of answers often broke the engagement of users. The interactions between the users and the chatbot were collected through the notation presented earlier, and we exhibit an excerpt of the gathered corpus in Annex A.

---

<sup>14</sup>JavaScript Object Notation



# 5 Upgrading the Multiagent System with a Learning Module

## Contents

---

5.1	Online Learning for Conversational Agents . . . . .	38
5.2	Weighted Majority Algorithm . . . . .	39
5.3	Setting up the Environment . . . . .	40
5.4	I Am Groot: Gauging the Learning Module's Effectiveness . . . . .	42
5.5	Learning the Agents' Weights . . . . .	43
5.6	Evaluating the System's Accuracy . . . . .	45
5.7	Evaluating the Systems with Human Annotators . . . . .	48
5.8	Comparing with our Previous Results . . . . .	49

---



This section describes the development of the learning module, the training and experiments performed, and the evaluations against similar systems.

## 5.1 Online Learning for Conversational Agents

In 2017, Mendonça et al. [14] proposed a learning approach for SSS that took into account the user feedback through the association of weights to each feature of the given agent and continuously updated the weights according to the quality of the candidate answers.

The user starts by making a query to the agent, which is used to obtain a set of candidate responses. In turn, each feature in the agent rates each response, and the best-scored response from each of the features is delivered to the user. Having received the best-scored responses, the user evaluates the quality of each response, and the weights of each feature are accordingly adjusted.

This approach uses the Exponentially Weighted Average Forecaster (EWA) algorithm [18] to update the feature weights, and it presented promising results when compared to the usage of fixed weights. SSS was used as an evaluation scenario, with the “user” being simulated by a learning module which selects a T-A pair from SSS’s corpus and makes a query to Lucene containing the selected Trigger, which in turn extracts the candidate responses as described in Section 2.2.

The candidate responses are scored by each feature in the agent, with the features corresponding to the first three measures described in Section 2.2 (that is, text similarity with input, response frequency, and answer similarity with input). The best-scored responses are evaluated by a reward function that calculates the Jaccard similarity between the best-scored response and the actual answer, and the reward function is then used to update each of the feature weights.

Upon training the feature weights with the Cornell Movie Dialogs corpus, a corpus composed of dialogues extracted from several movies, the obtained results showed a huge improvement over fixed weights. However, it was also verified that the choice of reward function can greatly influence the performance of the algorithm.

We decided to follow Mendonça et al.’s footsteps and implement this algorithm in our system, treating each of the described features as an agent, and therefore learning the weights for each given agent in our system.

## 5.2 Weighted Majority Algorithm

As mentioned before, Mendonça et al.'s approach uses the Exponentially Weighted Average Forecaster, a variant of the Weighted Majority algorithm [18], detailed below in Algorithm 1.

---

**Algorithm 1** Learning weights using Weighted Majority (adapted from Mendonça et al.)

---

**Input:** *interactions, experts*

**Output:** *weights*

```

 $t \leftarrow 1$ 
 $K \leftarrow \text{length}(\text{experts})$ 
 $w^k(t) \in \text{weights} \leftarrow 1/K : 1 \leq k \leq K$ 
for each  $u(t) = (T_u, A_u) \in \text{interactions}$  do
   $\text{candidates} \leftarrow \text{getCandidates}(T_u)$ 
  for each  $E_k \in \text{experts}$  do
     $\text{bestCandidate} \leftarrow \text{none}$ 
     $\text{bestScore} \leftarrow 0$ 
    for each  $c_i = (T_i, A_i) \in \text{candidates}$  do
       $\text{score}_{ki} \leftarrow \text{computeScore}(T_u, c_i, E^k)$ 
      if  $\text{bestScore} < \text{score}_{ki}$  then
         $\text{bestScore} \leftarrow \text{score}_{ki}$ 
         $\text{bestCandidate} \leftarrow c_i$ 
      end if
    end for
     $r^k(t) \leftarrow \text{rewardExpert}(A_u, \text{bestAnswer})$ 
     $R^k(1, \dots, t) \leftarrow R^k(1, \dots, t-1) + r^k(t)$ 
     $\text{old\_weights} \leftarrow \text{weights}$ 
     $w^k(t+1) \leftarrow \text{updateWeight}(R^l(1, \dots, t))$ 
     $w_{\text{total}}(t+1) \leftarrow w_{\text{total}}(t) + w^k(t+1)$ 
  end for
  for each  $E^k \in \text{experts}$  do
     $w^k(t+1) \leftarrow w^k(t+1)/w_{\text{total}}(t+1)$ 
  end for
   $t \leftarrow t+1$ 
end for
return weights

```

---

The algorithm receives a set of interactions (which, in our case, corresponds to a part of the Cornell Movie Dialogs corpus) and a set of experts that will evaluate each answer proposed by our system (ergo, our system's agents shall be the experts in the learning algorithm). For each iteration  $t$ , an interaction  $u(t)$

is picked from the reference corpus, and a set of candidates is retrieved from the subtitle corpus through Lucene, similarly to the procedure described in section 4. Each expert  $E_k$  takes turns evaluating all of the provided candidates, giving a score to each one. To compute the reward  $r^k(t)$  given to each expert, the expert’s best-scored candidate’s answer  $A_n$  is compared to the reference interaction’s corresponding answer  $A_{u(t)}$  and the Jaccard similarity between the two answers is retrieved and rounded by  $\alpha$  decimal places ( $\alpha$  being a configurable parameter), and serving as the reward for that expert, as expressed through the following formula:

$$r^k(t) = Jac(A_n, A_{u(t)})$$

This reward is then used to update the agent’s weight, with the weights being updated according to the sum of rewards received so far,  $R^k(1, \dots, t)$  as shown in the equation:

$$w^k(t+1) = e^{\eta R^k(1, \dots, t)}$$

The variable  $\eta$  depends on the number of experts  $K$ , the expected number of iterations  $U$ , and a configurable parameter  $\beta$ , as presented in the formula:

$$\eta = \sqrt{\frac{\beta \log K}{U}}$$

Unless otherwise specified, each expert will initially have the same weight, which will correspond to a fraction of the total number of experts in the system.

### 5.3 Setting up the Environment

With the algorithm established, we also chose to use the Cornell Movie Dialogs reference corpus to train and evaluate our system: while the SubTle corpus was created without the guarantee that two sequential dialogues belonged to different characters (and thus, composed a conversation) and relied on lexical clues to separate the subtitles into Trigger-Answer pairs, the Cornell Movie Dialogs corpus was built with the knowledge of which character spoke what line, resulting in a metadata-rich corpus assuredly composed of conversations between a pair of distinct characters. Adding to that, as previously stated, this was the corpus used to train Mendonça et al.’s data system (which we will refer to as **SSS + Learning** from this point on), permitting us to replicate the environment of their experiments.

This corpus is comprised of several files pertaining to the structure of the conversations (`movie_conversations.txt`), the written content of the subtitles (`movie_lines.txt`), and metadata regarding the actual movies (`movie_titles_metadata.txt`) and its characters (`movie_characters_metadata.txt`). For the purpose of our work, we will be using the files containing the movie conversations and lines, which we further describe below.

The `movie_conversations.txt` text file contains the structure of the conversations. A typical line of this text file is represented as follows:

```
u2 +++$+++ u7 +++$+++ m0 +++$+++ ['L778', 'L779']
```

Breaking it up, the set of characters `+++${}+++` serves as the separator of each field. The first field (`u2`) represents the ID of the first character of the interaction, with the second field (`u7`) corresponding to the ID of the second character. The third field (`m0`) is the unique ID of the movie, and the last field (`['L778', 'L779']`) makes up the list of line IDs that constitute the conversation, which will be matched with the `movie_lines.txt` file in order to reform the actual contents of the conversation.

On that note, the following is what a line from the `movie_lines.txt` file looks like:

```
L366 +++${}+++ u0 +++${}+++ m0 +++${}+++ BIANCA +++${}+++ You're sweet.
```

Once again, the set of characters `+++${}+++` is the field separator, with the first field (`L366`) corresponding to the ID of the line. The second field (`u0`) denotes the ID of the character who spoke the line, and the third field (`m0`) is the movie ID, akin to the case of the previous file. The fourth field (`BIANCA`) is the character's name, and the last field (`You're sweet.`) contains the content of the line.

In order to perform the training, our system's configuration file receives the following parameters regarding the learning phase:

- `<interactions>` – The path of the text file containing the conversations of the Cornell Movie Dialogs corpus. This can correspond to the `movie_conversations.txt` file, or another file containing a part of the conversations.
- `<lines>` – The path of the text file containing the lines of the Cornell Movie Dialogs corpus. We will be using the `movie_lines.txt` file throughout all of the experiments.
- `<inputSize>` – The amount of conversations (ergo, lines from the `<interactions>` file) to be read during the learning phase. For example, if there are 20000 conversations in the `<interactions>` file and the input size is 1000, the system's weights will be trained with the first 1000 lines of that file.
- `<decimalPlaces>` – Recalling the Weighted Majority Algorithm, the amount of decimal places corresponds to the  $\alpha$  parameter, and is used to round up the value of the reward.
- `<etaFactor>` – Once again, going back to the Weighted Majority Algorithm, this is the configurable parameter  $\beta$ , which serves as a part of the formula which determines the value of  $\eta$ .
- `<initialWeights>` – Optionally, we can grant a set of initial weights to the system, overriding the initial assignment of equal weights to each expert. For example, if we have `Agent1` and `Agent2` in my system, and we wish to give a weight of 60% to `Agent1` and a weight of 40% to `Agent2`, then the weight specification will be: `{'Agent1': 60, 'Agent2': 40}`.

The experiences performed on SSS+ indicate that the algorithm performs at its best for the values of  $\alpha = 0$  and  $\beta = 4$ , which led us to carry out all of our experiments with those same values set as our configuration parameters. Additionally, the candidates gathered by the agents are extracted from the

subtitle corpus defined in the default configuration. Unlike the previous experiments that used SubTle as its subtitle corpus, the following evaluations will be either using the entirety of the Cornell Movie Dialogs or a part of it as their subtitle corpus<sup>15</sup>.

## 5.4 I Am Groot: Gauging the Learning Module’s Effectiveness

Now that the environment groundwork is properly defined, it is time to put the learning module into action. Following that point of thought, we wanted to ensure that the system was actually learning the weights properly, and, as such, we set up our system with the same agents as before (ergo, the agents for the Cosine Similarity, Levenshtein Distance and Jaccard Distance, each with variations of 50/50, 75/25 and 100/0 for the Question Similarity/Answer Similarity), which are agents that have proven their worth as a chorus in the previous experiments. To this set of agents, we added a single additional agent, which we named *GrootAgent*.

Groot is a tree-like creature from Marvel™, whose main form of verbal communication is the sentence “*I am Groot*”. Taking inspiration from this character, the *GrootAgent* will deliver the answer “*I am Groot!*” regardless of the query that it receives, which, by nature, makes it a terrible agent to answer any kind of questions that do not require self-identification (that is, any question that does not boil down to “*Who are you?*” or “*What’s your name?*”). Thinking from that perspective, if the learning module is working properly, then the weight of *GrootAgent* will quickly decline and its answer will cease to be regarded as a proper one, no matter the weights initially assigned to the agents.

As such, we set the initial weight of *GrootAgent* to 99.91 and the weight of every other agent to 0.01, giving a massive weight to *GrootAgent* and disregarding all of the other agents. A set of 2000 dialogue interactions was extracted from the movie conversations text file and set as the `interactions` parameter, with the `inputSize` being set to 2000 accordingly. The subtitle corpus from which the agents will gather candidates was the entirety of the Cornell Movie Dialogs corpus.

Upon establishing those weights, we asked ourselves: will the system recognize that *GrootAgent* is not a good agent and lower its weight? If so, how many iterations of learning will it take until all of the other agents outweigh *GrootAgent*? And, furthermore, how many iterations will the system have to run until it ceases to answer “*I am Groot!*”? Would 2000 iterations even be enough?

As shown in Figure 5.1, the system’s weights evolved as expected: even though the system started with an enormous weight gap between the weight of *GrootAgent* and the remaining agents, it took only 18 iterations of training until *GrootAgent* was outweighed by every single one of the other agents. The system only answered “*I am Groot!*” on the first seven interactions, with the answers to the remaining 1993 interactions being chosen through a consensus of the lexical agents. The weights<sup>16</sup> of each agent at iteration 30 are presented in Table 5.1.

---

<sup>15</sup>This version of the Cornell Movie Dialogs corpus has been converted to the same format as SubTle, therefore allowing the system to index and retrieve it through Lucene.

<sup>16</sup>The weights presented were rounded to three decimal places.

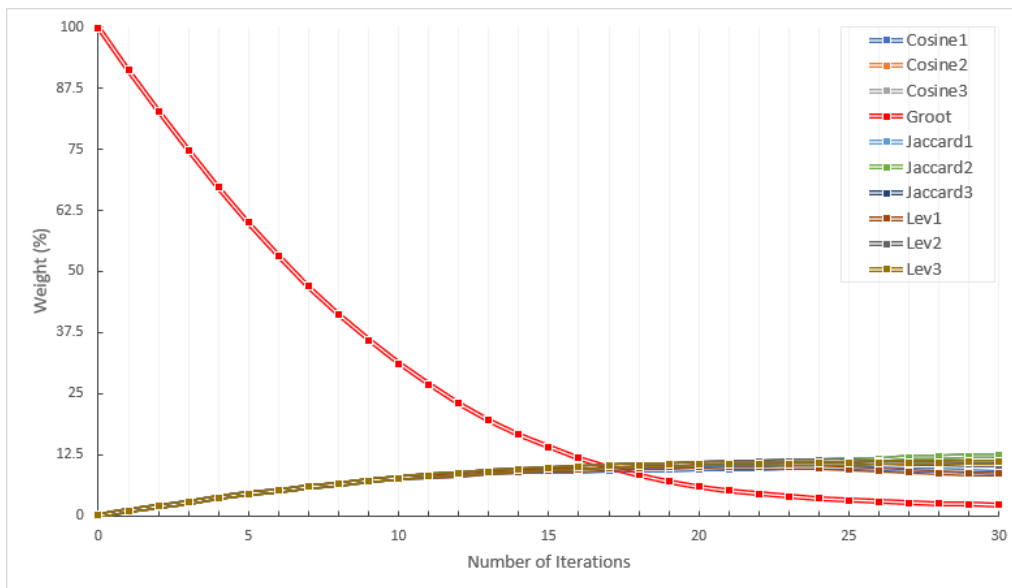


Figure 5.1: Evolution of the weight bestowed to each agent through 30 iterations of learning.

The results displayed in Table 5.1 show us that even though all of the agents (except for *GrootAgent*) are very close in weight value (as expected, due to the low number of iterations), there are two agents slightly underperforming when compared to the rest of the lot, those being *CosineAgent1* (50/50) and *LevenshteinAgent1* (50/50). The next experiment will delve further into the main lot of agents and show a bit of how the minuscule amount of thirty iterations already manages to give us a few hints about the future.

Agent	Weight
CosineAgent1 (50/50)	8.877
CosineAgent2 (75/25)	12.379
CosineAgent3 (100/0)	10.967
GrootAgent	2.144
JaccardAgent1 (50/50)	12.379
JaccardAgent2 (75/25)	12.379
JaccardAgent3 (100/0)	10.967
LevenshteinAgent1 (50/50)	8.542
LevenshteinAgent2 (75/25)	10.396
LevenshteinAgent3 (100/0)	10.967

Table 5.1: Weight of each agent after 30 iterations in the Groot experiment.

## 5.5 Learning the Agents' Weights

In our previous experiments we worked with equal weights for each agent and with the prioritization of domain-specific agents in the case that there were solid grounds to believe that said agent could

accurately answer a received query. These kinds of scenarios can easily lead to inappropriate answers from the system if the agents delivering those answers are not good enough, which led us to follow up on Mendonça et al.’s work and adapt the Weighted Majority Algorithm to a multiagent system, as described in Subsection 5.2.

To train the system’s weights, we adapted Mendonça et al.’s procedure of setting aside 18000 dialogues from the Cornell Movie Dialogs corpus to use as our reference corpus (ergo, the `interactions` parameter) and employed the entire Cornell Movie Dialogs corpus as the subtitle corpus, from where the agents will gather their candidates. The initial weights were equal for each agent, and the system was deployed with the agents described in Subsection 5.4, with the exclusion of *GrootAgent*.

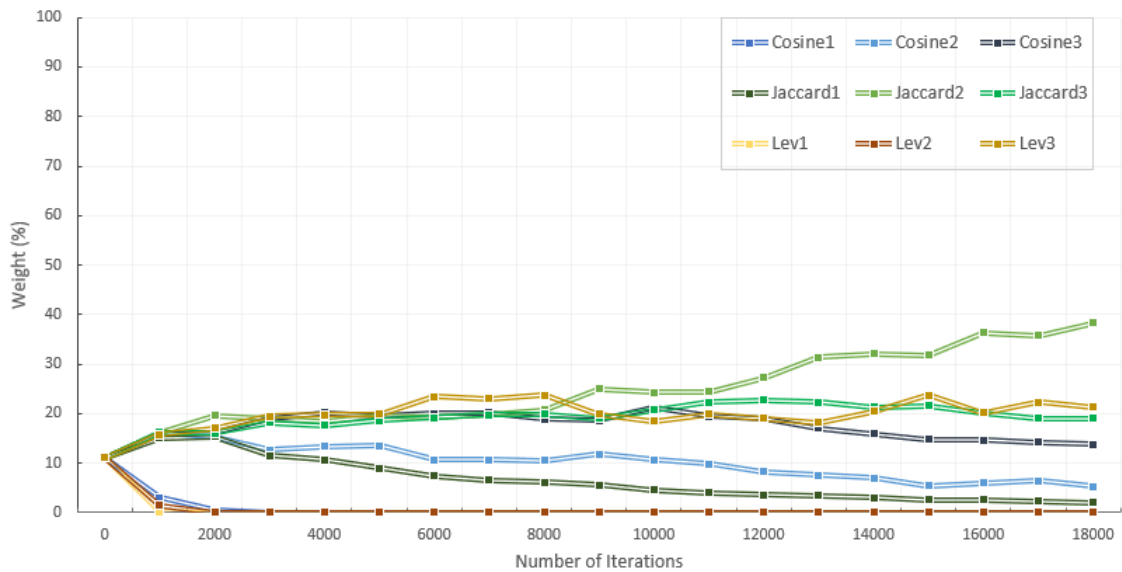


Figure 5.2: Evolution of the weight given to each agent throughout the 18000 iterations of training.

The results of the training can be observed in Figure 5.2, which displays the evolution of the weights of each agent throughout the whole learning process. Right from the start, it is interesting to observe that, as hinted by the previous experiment involving 30 iterations, two of the agents with a 50/50 ratio on question similarity and answer similarity were quickly disregarded by the system (*CosineAgent1* and *LevenshteinAgent1*), with *JaccardAgent1* managing to stand on par with the remaining agents throughout the first 2000 iterations of training, but falling to a weight of 2.073% by the end of the training. *LevenshteinAgent2* was quickly disregarded by the system as well, which initially led us to believe that the Levenshtein Distance was not a good metric to apply in the system. However, *LevenshteinAgent3* was highly rated by the system, finishing the training as the second-highest weighted agent with a weight of 21.307%.

Through Table 5.2, it can be verified that the other two agents employing the Cosine Similarity metric depict an interesting phenomena throughout the training: while in the first half of the training, *CosineAgent3* was rated as one of the best performing agents (with a weight of 21.173% after 10000 iterations) and *CosineAgent2* managed to stand relatively up to par, with its weight floating around the

Agent \ Iterations	0	5000	10000	15000	18000
CosineAgent1 (50/50)	11.111	0.004	0.000	0.000	0.000
CosineAgent2 (75/25)	11.111	13.551	10.741	5.449	5.361
CosineAgent3 (100/0)	11.111	19.461	21.173	14.741	13.863
JaccardAgent1 (50/50)	11.111	9.019	4.651	2.642	2.073
JaccardAgent2 (75/25)	11.111	19.461	24.250	31.809	<b>38.366</b>
JaccardAgent3 (100/0)	11.111	18.600	20.699	21.654	19.029
LevenshteinAgent1 (50/50)	11.111	0.000	0.000	0.000	0.000
LevenshteinAgent2 (75/25)	11.111	0.000	0.000	0.000	0.000
LevenshteinAgent3 (100/0)	11.111	19.906	18.485	23.705	21.307

Table 5.2: Weight of each agent throughout the 18000 iterations of training.

initial value during the first 10000 iterations, the second half of the training decreased their weight to 13.863% and 5.361% respectively, while *JaccardAgent2* jumped from an impressive value of 24.250% to an astounding weight of 38.366%. Even more curious is the fact that neither of the other two Jaccard agents showed such an improvement, which hints that, from these possibilities, the best performance when handling interactions from the Cornell Movie Dialogues corpus may come from using the Jaccard measure with a major focus on the similarity between the query and the *Trigger* parameter of the retrieved candidates, but also slightly taking into account the similarity between the query and the candidates' *Answer* parameter.

## 5.6 Evaluating the System's Accuracy

So far, the decision methods available to our system were the multiple branches of the Voting Model and Priority System, and neither of them are able to take weights into account. On account of that, we created a new decision method named Weighted Vote in order to determine an answer while taking into account each agent's weight.

The Weighted Vote works as follows:

- Upon receiving a query, each agent retrieves and rates a set of candidates, subsequently delivering its best-rated answers to the system.
- From the side of the system, each of the gathered answers is given its agent's weight as score: if more than one agent delivers the same answer, that answer's score will be the sum of the weights of all agents that delivered it as a response.
- Finally, the answer with the greatest sum of agent weights is presented to the user.

As a practical example, let us suppose that the agents in our system are *Agent1* and *Agent2* with



weights of 60% and 40% respectively, and that a user asked the system “*How are you?*”. Let’s imagine that both agents rate their candidates, with *Agent1* retrieving the answers “*I’m okay, thanks!*” and “*I’m fine, how about you?*” as its best-rated answers, while *Agent2* delivers the answers “*I’m fine, thanks.*”, “*I’m fine, how about you?*” and “*I feel terrible today.*”.

Given their assigned weights, the Weighted Vote decision method will rate the answers as follows:

```
"I'm okay, thanks!" <- 60
"I'm fine, how about you?" <- 100 = 60 + 40
"I'm fine thanks." <- 40
"I feel terrible today." <- 40
```

Upon computing those results, the answer “*I’m fine, how about you?*” would be delivered to the user, as it is the highest-weighted answer delivered by the agents.

Following the procedure conducted by Mendonça et al. (2017) to determine their system’s accuracy, we used 2000 of the Cornell Movie Dialogues corpus as both the reference corpus (that is, the `<interactions>` parameter) and the subtitle corpus from which the systems would retrieve their candidates. In the context of our problem, the accuracy is defined as the percentage of answers where the system was able to deliver an exact match to the expected reference answer.

To exemplify this process, suppose a given interaction from the reference corpus is composed by:  
Trigger - “Is everything okay, Jenny?”

Answer - “I’m fine, dad.”

The Trigger parameter of the interaction is sent to the system as a query, which will retrieve its candidates from the subtitle corpus. Since, in this case, the subtitle corpus is composed by the same interactions as the reference corpus, the system should be able to retrieve that specific interaction among its candidates. The candidates are rated by the system’s agents, and the answer delivered by the agents is compared to the reference answer: if the system answers “I’m fine, dad.”, then the answer is considered correct and given an accuracy score of 1, while any other answer will be discerned as wrong and given an accuracy score of 0.

We computed the accuracy for four different systems:

- SSS – Magarreiro et al.’s version of Say Something Smart, utilizing the lexical features described in Section 2.2 and assigning the weights reported as best in their work: 34% to  $M_1$  (Text Similarity with the Input), 33% to  $M_2$  (Response Frequency) and 33% to  $M_3$  (Answer Similarity with the Input).
- SSS + Learning – Magarreiro et al.’s version of Say Something Smart, but with Mendonça et al.’s best-performing weights through the learning procedure: 76% to  $M_1$  (Text Similarity with the Input), 14% to  $M_2$  (Response Frequency) and 10% to  $M_3$  (Answer Similarity with the Input).
- MultiSSS 18000 – Our multiagent version of Say Something Smart, using the Weighted Vote with the weight configuration reported in iteration 18000 of Table 5.2.

- MultiSSS Equal – Our multiagent version of Say Something Smart, but with equal weights given to each agent (ergo, the weight configuration reported in iteration 0 of Table 5.2) in the Weighted Vote, which is the same as performing a normal Voting Model.

System	Accuracy
SSS	90.6%
SSS + Learning	<b>95.4%</b>
MultiSSS 18000	93.8%
MultiSSS Equal	93.6%

Table 5.3: Accuracy of the systems when evaluated with 2000 interactions of the CMD corpus.

Through the information presented in Table 5.3, we can observe that all systems reported an accuracy greater than 90%. As expected, SSS’s configuration was outmatched by both SSS + Learning and our system, but the configuration of SSS + Learning managed to beat both of the configurations presented for our multiagent system. Furthermore, our system’s accuracy changes were negligible, with a 0.2% change between trained weights and unregulated weights.

We believe that both of those occurrences have explanations behind them. Let us start with why the SSS + Learning system may have outperformed ours: their system was composed by the three lexical features described earlier, each given a specific weight: 76% to the text similarity with the input, 14% to the response frequency and 10% to the answer Similarity with the input, and it computed each of those lexical features through the Jaccard Similarity.

In the case of our system, our agents do not take the response frequency into account directly as that role was passed through to the decision maker, who instead verifies the redundancy of a given answer if it is delivered by multiple agents. On the other hand, there is an agent in our set of experts that fits the glove perfectly in terms of assigning a high importance to question similarity, a low importance to answer similarity (while still taking into account), and featuring the Jaccard similarity measure in order to compute its scores. That agent is *JaccardAgent2 (75/25)*, which so happened to be the highest-rated agent during the 18000 iterations of learning described in Section 5.5 with a final weight of 38.366, vastly outweighing all of its agent companions.

As such, we believe that if the system were to continue training beyond 18000 iterations, *JaccardAgent2* would eventually vastly outscale all of the other agents, as an extremely similar version of this agent has proven to stand its ground in terms of accuracy without the support of any other agents, as was the case of the results obtained with SSS + Learning.

Finally, there is the matter of the lack of improvement between the equal weights configuration of our system with an accuracy of 93.6% and the trained weights configuration that had an accuracy of 93.8%. While the amount of correct answers did not significantly change, most of the answers deemed as wrong are different between the two systems. This hints that the agents are deeming multiple answers to be plausible for each of the interactions that they answered wrongly. Furthermore, the trained system

answered correctly to exactly the same questions as the equal weights system, which indicates that the majority of the correct answers are being reached through consensus between the agents.

## 5.7 Evaluating the Systems with Human Annotators

As our final experiment, we decided to follow up from the procedure described in Section 4.4.1 and replicate the experiment for both our Multiagent system with the trained weights and SSS + Learning. We fed both of these systems the sets of 100 simple questions and 100 complex questions used earlier and gathered their responses, subsequently organizing them into the following sets:

- Simple questions answered by the trained Multiagent system.
- Simple questions answered by SSS + Learning.
- Complex questions answered by the trained Multiagent system.
- Complex questions answered by SSS + Learning.

A sample of 50 questions and answers from each of those sets was given to four human annotators, who gave a score between 1 and 4 to each answer according to the criteria:

4 - The given answer is plausible without needing any additional context.

3 - The given answer is plausible in a specific context, or does not actively answer the query but maintains its context.

2 - The given answer actively changes the context (for example, an answer that delivers a question to the user which does not match the initial query’s topic), or the given answer contains issues in its structure (even if its content fits in the context of the question).

1 - The given answer has no plausibility value.

The mean score of the systems was computed through the mean value of each system’s respective answers’ scores, and the approval rate was also calculated (that is, the ratio of answers with an average score of 2.75 or more between the four annotators). Additionally, the ratio of perfect answers was also computed: an answer is deemed to be perfect if all annotators give it a score of 4.

Metric	SSS + Learning	Multiagent + Learning
Mean Score	2.545	3.05
Approval	48%	68%
Perfect Ratio	26%	48%

Table 5.4: Mendonça et al.’s system against our trained system when answering basic questions.

Regarding basic questions, as shown in Table 5.4 our system vastly outperformed SSS + Learning with a mean score difference of 0.5; while SSS + Learning had 24 answers above the approval threshold, our

Metric	SSS + Learning	Multiagent + Learning
Mean Score	2.53	2.95
Approval	48%	64%
Perfect Ratio	18%	32%

Table 5.5: Mendonça et al.’s system against our trained system when answering complex questions.

system had 34 answers with an average score of 2.75 or more. Furthermore, 24 of our system’s answers had a perfect score from the annotators, while SSS + Learning had 13 perfect answers.

On the matter of complex questions, as shown in Table 5.5, our system managed to outperform SSS + Learning. However, although the approval ratio was similar to the results shown in the basic questions experiment on Table 5.4, the percentage of perfectly-scored answers significantly declined. This indicates disagreement between the annotators, or suggests that a part of the “good” answers to complex questions are simply not good enough.

## 5.8 Comparing with our Previous Results

Let us recall the results of the first experiments performed in this work, with Magarreiro’s SSS against our initial MultiAgent SSS. For readability purposes, we present the results obtained from both experiences in Tables 5.6 and 5.7 (corresponding to Tables 4.1 and 4.2 respectively).

Although the experiments were conducted independently at different times, the same sets of questions were used, and the annotators were the same in all of the experiments.

Comparing the results shown in Table 5.4 with our previous experiment, Table 5.6 shows that the trained multiagent with the Weighted Agents system (Multiagent + Learning) greatly outperforms the Voting Model-based system (Multiagent). Besides that, SSS + Learning managed to have a greater approval ratio than Magarreiro’s even though its mean score was lower, which indicates that the wrong answers of SSS + Learning are more penalized than Magarreiro’s.

Metric	Magarreiro’s	Multiagent	SSS + Learning	Multiagent + Learning
Mean Score	2.68	2.6	2.545	3.05
Approval	46%	48%	48%	68%

Table 5.6: The four systems compared when answering basic questions

For the complex questions experiments, SSS + Learning managed to significantly improve the approval ratio and mean score of Magarreiro’s system through the different assignment of weights, and also obtained an approval rating better than the Voting Model system, as displayed in Table 5.7. The Multiagent + Learning system managed to outpace every other system, with a difference of 42% approval and 0.69 mean score from the worst-performing system (Magarreiro’s).

Metric	Magarreiro’s	Multiagent	SSS + Learning	Multiagent + Learning
Mean Score	2.26	2.6	2.53	2.95
Approval	22%	44%	48%	64%

Table 5.7: Magarreiro’s system against our system when answering complex questions



# 6 Conclusions and Future Work

## Contents

---

6.1 Main Contributions . . . . .	53
6.2 Future Work . . . . .	53

---

This section describes the conclusion of this work. We discuss the main contributions of our work, and point out possible follow-up points for a further iteration of this work.

## 6.1 Main Contributions

In this work, we proposed a multi-agent system that took into account the answers of all its agents through a consensus model. While, from an engineering point of view, it may not be the most conventional way of deciding the answer to deliver, as performance is often traded in exchange for considering answers from unspecialized agents, it has been shown not only to be an effective approach when dealing with out-of-domain requests, but it also obtained remarkable results when tested against a system specialized in a specific domain, not deteriorating its accuracy significantly.

As such, our system has proved to thrive in situations where a domain-specific agent needs to deal with out-of-domain interactions, such as Edgar, and we have grounds to believe that user engagement with domain-specific dialogue engines would improve when using our framework to answer out-of-domain requests.

Furthermore, we implemented a learning module that allows our system to learn which agents perform best when answering out-of-domain queries. While the accuracy evaluations did not present a significant improvement, the performance of our system was shown to improve significantly when interacting with humans. We also present a corpus of interactions collected from dialogues between our system and humans through Discord, shown in Annex A.

## 6.2 Future Work

Regarding future work, the use of Lucene regarding the retrieval of candidate pairs can be explored further, as Lucene supports functionalities such as fuzzy searches and the inclusion and search of synonyms. While it is identified, we also ended up not solving the redundancy bug described in Section 2.3.

Additionally, the variety of agents employed on the system scratched only the surface of an enormous, ever-expanding universe of techniques and domains: using more advanced systems, such as sequence-to-sequence models or translation-based models would be interesting and, most likely, fruitful in terms of results.

Taking context into account is also an important point to take into account when discussing any kind of conversational engine: while we have laid the groundwork on gathering data for context-based conversation through Discord and collected an initial corpus of interactions, we did not thoroughly evaluate our procedure. Additionally, there is also the possibility of integrating the learning procedure described in Section 5 directly with the user feedback from Discord. While this approach would take a great number of human interactions before noticeable changes could occur, Discord allows many users to interact with the chatbot at the same time, thus we believe it is a feasible option.



# References

- [1] Hakkani-Tür, D., 2018. Introduction to Alexa Prize 2018 Proceedings, 2nd Proceedings of Alexa Prize (Alexa Prize 2018)
- [2] McCandless, M., Hatcher, E., Gospodnetic, O. (2010). Lucene in Action, Second Edition: Covers Apache Lucene 3.0. Manning Publications Co. Greenwich, CT, USA ©2010 ISBN:1933988177 9781933988177
- [3] Jones, K.S., Walker, S., & Robertson, S.E. (2000). A probabilistic model of information retrieval: development and comparative experiments - Part 2. *Inf. Process. Manage.*, 36, 809-840.
- [4] Singhal, A. & Google, I., (2001). Modern Information Retrieval: A Brief Overview. *IEEE Data Engineering Bulletin*. 24.
- [5] Jaccard, P., (1912). The distribution of the flora in the alpine zone. *New Phytologist*, 11(2):37–50.
- [6] Navarro, G., (2000). A Guided Tour to Approximate String Matching. *ACM Computing Surveys*. 33. 10.1145/375360.375365.
- [7] Ameixa, D., & Coheur, L. (2013). From subtitles to human interactions : introducing the SubTle Corpus. Tech. Rep. 1 / 2014 INESC-ID Lisboa, February 2014
- [8] Magarreiro, D., Coheur, L. & Melo, F. (2014). Using subtitles to deal with Out-of-Domain interactions., In *SemDial 2014 - DialWatt*, August 2014
- [9] Qiu, M., Li, F., Wang, S., Gao, X., Chen, Y., Zhao, W., Chen, H., Huang, J. & Chu, W., 2017. AliMe Chat: A Sequence to Sequence and Rerank based Chatbot Engine. 498-503. *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (January 2017)*.
- [10] Fialho, P., Coheur, L., dos Santos Lopes Curto, S., Cláudio, P.M.A., Costa, A., Abad, A., Meinedo, H., Trancoso, I.: Meet edgar, a tutoring agent at monserrate. In: *ACL. Proceedings of the 51st Annual Meeting of the Association for Computer Linguistics (August 2013)*
- [11] Ji, Z., Lu, Z. & Li, H. (2014). An Information Retrieval Approach to Short Text Conversation. *arXiv:1408.6988*
- [12] Yan, Z., Duan, N., Bao, J., Chen, P., Zhou, M., Li, Z., & Zhou, J. (2016). DocChat: An Information Retrieval Approach for Chatbot Engines Using Unstructured Documents. *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*.
- [13] Chen, C., Yu, D., Wen, W., Yang, Y., Zhang, J., Zhou, M., Jesse, K., Chau, A., Bhowmick, A., Iyer, S., Sreenivasulu, G., Cheng, R., Bhandare, A. & Yu, Z. Gunrock: Building a human-like social bot by leveraging large scale real user data. In *Proc. Alexa Prize, 2018*.

- [14] Mendonça, V., Melo, F., Coheur, L. & Sardinha, A. (2017). Online Learning for Conversational Agents. Progress in Artificial Intelligence: 18th EPIA Conference on Artificial Intelligence, EPIA 2017, Porto, Portugal, September 5-8, 2017, Proceedings (pp.739-750)
- [15] Ameixa, D., Coheur, L., Fialho, P., & Quaresma, P. (2014). Luke, I am Your Father: Dealing with Out-of-Domain Requests by Using Movies Subtitles. Intelligent Virtual Agents: 14th International Conference.
- [16] Wu, W., Lu, Z. & Li, H. (2013). Learning Bilinear Model for Matching Queries and Documents. The Journal of Machine Learning Research. 14. 2519-2548.
- [17] Xue, X., Jeon, J. & Croft, W. (2008). Retrieval models for question and answer archives. Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR 2008, Singapore.
- [18] Littlestone, N., Warmuth, M.K.: The weighted majority algorithm. Inf. Comput. 108(2), 212–261 (1994), <http://dx.doi.org/10.1006/inco.1994.1009>
- [19] Fialho, P., Marques, R., Martins, B., Coheur, L. & Quaresma, P. (2016). INESC-ID@ASSIN: Measuring semantic similarity and recognizing textual entailment. 8. 33-42.
- [20] Papineni, K., Roukos, S., Ward, T., & Zhu, W. (2002). Bleu: a Method for Automatic Evaluation of Machine Translation. Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL), Philadelphia, July 2002, pp. 311-318.
- [21] Lavie, A., Agarwal, A. (2005). METEOR: An automatic metric for MT evaluation with high levels of correlation with human judgments. In Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, pages 65-72, Ann Arbor, Michigan, June.
- [22] Matthew, S., Bonnie, D., Richard, S., Linnea, M., & John, M., A Study of Translation Edit Rate with Targeted Human Annotation, Proceedings of Association for Machine Translation in the Americas, 2006.
- [23] Lin, C. & Hovy, E. (2003) Automatic Evaluation of Summaries Using N-gram Co-Occurrence Statistics , Proceedings of HLT-NAACL 2003, Main Papers , pp. 71-78
- [24] Danescu-Niculescu-Mizil, C. & Lee, L. (2011). Chameleons in imagined conversations: A new approach to understanding coordination of linguistic style in dialogs, CMCL '11 Proceedings of the 2nd Workshop on Cognitive Modeling and Computational Linguistics, pp. 76-87

# A Corpus of Interactions Gathered Through Discord

This annex presents the collected corpus of interactions through our chatbot’s Discord implementation, which consisted in Edgar as a Priority Agent and CosineAgent, LevenshteinAgent and JaccardAgent as backup agents using the SubTle corpus to answer queries. The notation of the corpus is described in Section 4.5.

```
1 {
2   "sequences":[
3     [
4       {
5         "query":"What would be the most amazing adventure to go on?",
6         "answer":"– What?",
7         "userID":135895383686512640,
8         "posReacts":1,
9         "negReacts":2,
10        "midReacts":1,
11        "timestamp":1570544008.1380916,
12        "messageID":631132056407244805,
13        "serverID":509192905139683328
14      },
15      {
16        "query":"What can you only do with your hands?",
17        "answer":"Your life.",
18        "userID":135895383686512640,
19        "posReacts":0,
20        "negReacts":0,
21        "midReacts":0,
22        "timestamp":1570544097.7002752,
23        "messageID":631132431642132490,
24        "serverID":509192905139683328
25      },
26      {
27        "query":"Do you know the lyrics to You Me At Six’s song, Underdog?",
28        "answer":"Better than you know yourself.",
29        "userID":135895383686512640,
30        "posReacts":2,
31        "negReacts":1,
32        "midReacts":1,
33        "timestamp":1570544318.79241,
34        "messageID":631133359174713364,
35        "serverID":509192905139683328
36      },
37      {
38        "query":"Damn, that’s a bold claim.",
39        "answer":"Have I ever made a claim I couldn’t prove?",
40        "userID":135895383686512640,
41        "posReacts":2,
```

```

42     "negReacts":1,
43     "midReacts":1,
44     "timestamp":1570544349.5579138,
45     "messageID":631133488376053770,
46     "serverID":509192905139683328
47 },
48 {
49     "query":" Well, not yet, but I'm afraid you'll disappoint me soon.",
50     "answer":" Well, I don't understand.",
51     "userID":135895383686512640,
52     "posReacts":2,
53     "negReacts":1,
54     "midReacts":1,
55     "timestamp":1570544369.776608,
56     "messageID":631133572748541985,
57     "serverID":509192905139683328
58 },
59 {
60     "query":" You'll understand soon. Just wait.",
61     "answer":" Be careful! Here, I'll do it.",
62     "userID":135895383686512640,
63     "posReacts":1,
64     "negReacts":1,
65     "midReacts":2,
66     "timestamp":1570544391.198384,
67     "messageID":631133662762762240,
68     "serverID":509192905139683328
69 },
70 {
71     "query":" Do what?",
72     "answer":" Do what? When are we gonna get married?!",
73     "userID":135895383686512640,
74     "posReacts":2,
75     "negReacts":1,
76     "midReacts":1,
77     "timestamp":1570544406.1826732,
78     "messageID":631133725878648833,
79     "serverID":509192905139683328
80 },
81 {
82     "query":" Oh man, I'm not ready to take that step yet.",
83     "answer":" NOT YET. I'M NOT READY.",
84     "userID":135895383686512640,
85     "posReacts":2,
86     "negReacts":1,
87     "midReacts":1,
88     "timestamp":1570544430.8388753,
89     "messageID":631133828982767627,
90     "serverID":509192905139683328

```

```

91     },
92     {
93         "query": "Phew. You scared me there.",
94         "answer": "Sorry.",
95         "userID": "135895383686512640",
96         "posReacts": 2,
97         "negReacts": 1,
98         "midReacts": 1,
99         "timestamp": "1570544444.7606595",
100        "messageID": "631133887686377498",
101        "serverID": "509192905139683328"
102    },
103    {
104        "query": "It's okay, I'd be more worried if you were serious.",
105        "answer": "You're not me.",
106        "userID": "135895383686512640",
107        "posReacts": 2,
108        "negReacts": 1,
109        "midReacts": 1,
110        "timestamp": "1570544471.6199703",
111        "messageID": "631134000307503124",
112        "serverID": "509192905139683328"
113    },
114    {
115        "query": "If I were you, would you want to marry me?",
116        "answer": "That isn't part of my short and long term plans.",
117        "userID": "135895383686512640",
118        "posReacts": 2,
119        "negReacts": 1,
120        "midReacts": 1,
121        "timestamp": "1570544499.3698347",
122        "messageID": "631134116649107466",
123        "serverID": "509192905139683328"
124    },
125    {
126        "query": "Hm, not even if you find someone really special to you?",
127        "answer": "All right, all right.",
128        "userID": "135895383686512640",
129        "posReacts": 1,
130        "negReacts": 1,
131        "midReacts": 1,
132        "timestamp": "1570544528.8541396",
133        "messageID": "631134239559122945",
134        "serverID": "509192905139683328"
135    },
136    {
137        "query": "So you're planning on getting married!",
138        "answer": "It doesn't concern you.",
139        "userID": "135895383686512640",

```

```

140         "posReacts":2,
141         "negReacts":1,
142         "midReacts":1,
143         "timestamp":1570544544.295045,
144         "messageID":631134304495206401,
145         "serverID":509192905139683328
146     },
147     {
148         "query":"Wow, okay. I won't bother you about it anymore.",
149         "answer":"You're not... you're not bothering me.",
150         "userID":135895383686512640,
151         "posReacts":2,
152         "negReacts":1,
153         "midReacts":1,
154         "timestamp":1570544581.0570693,
155         "messageID":631134459298578452,
156         "serverID":509192905139683328
157     },
158     {
159         "query":"Oh, you don't have to be polite just for the sake of it.
160 You can be honest with me.",
161         "answer":"I like the dorky farm-boy thing.",
162         "userID":135895383686512640,
163         "posReacts":2,
164         "negReacts":1,
165         "midReacts":1,
166         "timestamp":1570544615.1819413,
167         "messageID":631134602358030346,
168         "serverID":509192905139683328
169     },
170     {
171         "query":"O-okay, you don't have to be that honest.",
172         "answer":"You don't have to die to keep the John Doe idea alive.",
173         "userID":135895383686512640,
174         "posReacts":1,
175         "negReacts":1,
176         "midReacts":2,
177         "timestamp":1570544630.7046967,
178         "messageID":631134667466211339,
179         "serverID":509192905139683328
180     },
181     {
182         "query":"Uh... okay?",
183         "answer":"Uh...",
184         "userID":135895383686512640,
185         "posReacts":1,
186         "negReacts":1,
187         "midReacts":1,
188         "timestamp":1570544657.166173,

```

```

189         "messageID":631134776916574210,
190         "serverID":509192905139683328
191     },
192     {
193         "query":" Sorry. I may have made things a bit awkward.",
194         "answer":"WELL, I'M SORRY, DEAR. COMING TO BED?",
195         "userID":135895383686512640,
196         "posReacts":2,
197         "negReacts":1,
198         "midReacts":1,
199         "timestamp":1570544683.40046,
200         "messageID":631134888589918208,
201         "serverID":509192905139683328
202     },
203     {
204         "query":"Why are you so insistent on these propositions?
205 First marriage, now this?",
206         "answer":"– Now look, Tia... if you wish, we could right now...",
207         "userID":135895383686512640,
208         "posReacts":2,
209         "negReacts":1,
210         "midReacts":1,
211         "timestamp":1570544750.6814635,
212         "messageID":631135170501673000,
213         "serverID":509192905139683328
214     },
215     {
216         "query":"My name ain't Tia, and I don't want to do it with you. Scumbag.",
217         "answer":" I don't want to do it. I can't do it.",
218         "userID":135895383686512640,
219         "posReacts":2,
220         "negReacts":1,
221         "midReacts":1,
222         "timestamp":1570544775.4470022,
223         "messageID":631135274394714135,
224         "serverID":509192905139683328
225     }
226 ],
227 [
228     {
229         "query":"Oh, great. Impotence isn't something I expected here.",
230         "answer":" I KNOW! I OVERHEARD A CONVERSATION BETWEEN HIM",
231         "userID":135895383686512640,
232         "posReacts":1,
233         "negReacts":2,
234         "midReacts":1,
235         "timestamp":1570544828.4780607,
236         "messageID":631135497074245646,
237         "serverID":509192905139683328

```

```

238     },
239     {
240         "query": "Between who and who?",
241         "answer": "The matter that you read.",
242         "userID": 135895383686512640,
243         "posReacts": 1,
244         "negReacts": 1,
245         "midReacts": 2,
246         "timestamp": 1570544852.7279859,
247         "messageID": 631135598798831656,
248         "serverID": 509192905139683328
249     },
250     {
251         "query": "I don't know what you're talking about.",
252         "answer": "You know what I'm talking about.",
253         "userID": 135895383686512640,
254         "posReacts": 0,
255         "negReacts": 0,
256         "midReacts": 0,
257         "timestamp": 1570544869.0560372,
258         "messageID": 631135667258130442,
259         "serverID": 509192905139683328
260     }
261 ],
262 [
263     {
264         "query": "That's really deep, man.",
265         "answer": "Because you people don't have hearts.",
266         "userID": 135895383686512640,
267         "posReacts": 2,
268         "negReacts": 1,
269         "midReacts": 1,
270         "timestamp": 1570546700.500536,
271         "messageID": 631143348958134348,
272         "serverID": 509192905139683328
273     },
274     {
275         "query": "So it's our fault now? Way to keep your issues bottled.",
276         "answer": "It's your fault!",
277         "userID": 135895383686512640,
278         "posReacts": 2,
279         "negReacts": 1,
280         "midReacts": 1,
281         "timestamp": 1570546736.9066186,
282         "messageID": 631143501484130312,
283         "serverID": 509192905139683328
284     }
285 ],
286 ]

```





## B Basic Questions used in the Evaluations

This annex lists the basic questions used through the various experiments described in this work.

A tua familia é numerosa?

Aceitas tomar café?

Achas que hoje vai estar calor?

Andas em que faculdade?

Andas na escola?

Como aprendeste a falar português?

Como correu o dia?

Como está o tempo?

Como se chama a tua mãe?

Como é que te chamas?

Costumas ir ao cinema?

Costumas sair à noite?

Dás-me o teu contacto?

De onde é que vens?

De que tipo de músicas gostas?

Em que dia nasceste?

Em que países já estiveste?

Então e esse tempo?

És aluno do Técnico?

És casado?

Estudas ou trabalhas?

Estudo informática. Gostas de informática?

Estás bom?

Falas inglês?

Falas quantas línguas?

Fazes algum desporto?

Gostas de animais?

Gostas de chocolate?

Gostas de estudar?

Gostas de fazer desporto?

Gostas de informática?

Gostas de jogar futebol?

Gostas de lasanha?

Gostas de mim?

Gostas de musica clássica?

Gostas de sushi?

Gostas de trabalhar?

Gostas mais de ler ou de ver filmes?  
Gostavas de ir ao Brasil?  
Gostavas de ter filhos?  
Já foste ao Brasil?  
Mudando de assunto, quais são os teus hobbies?  
Nasceste onde?  
O meu nome é Pedro, qual é o teu?  
O que fazes nos tempos livres?  
O que fazes aqui?  
O que fazes na vida?  
O que gostas de fazer?  
O que é que estudas?  
O tempo tem estado bastante mau, não tem?  
Olá, tudo bem?  
Olá! Como estás?  
Onde é que compraste essa roupa?  
Onde é que estudaste?  
Onde moras?  
Onde vais assim vestido?  
Para onde vais?  
Praticas desporto?  
Preciso de direcções para chegar a Lisboa, sabes me dar?  
Quais são os teus livros favoritos?  
Quais é que são as tuas maiores qualidades?  
Quais é que são os teus hobbies?  
Qual o dia do teu aniversário?  
Qual o prato que mais gostas?  
Qual o teu nome completo?  
Qual é a melhor faculdade do país?  
Qual é a tua banda favorita?  
Qual é a tua cor favorita?  
Qual é a tua língua materna?  
Qual é a tua nacionalidade?  
Qual é o teu clube de futebol preferido?  
Qual é o teu desporto favorito?  
Qual é o teu número de telemóvel?  
Qual é o teu tipo de música favorito?  
Quando é que nasceste?  
Que género de música é que gostas?  
Que dia é hoje?  
Que disciplinas tens?  
Que horas são?

Que país gostavas de visitar?  
Que séries gostas de ver?  
Quem são os teus pais?  
Queres ir comer um gelado?  
Queres ir dar uma volta por Lisboa?  
Queres ir para outro lugar?  
Queres uma goma?  
Sabes conduzir?  
Sabes cozinhar bem?  
Se fores sair avisas-me?  
Sou bonito?  
Tens algum animal de estimação?  
Tens algum filme favorito?  
Tens dupla nacionalidade?  
Tens namorada?  
Tens quantos anos?  
Tens que idade?  
Tens um chapéu-de-chuva?  
Tens um euro que me possas emprestar?  
Tudo bem contigo?  
Vives sozinho ou com a tua família?

# C Complex Questions used in the Evaluations

This annex lists the complex questions used through the various experiments described in this work.

A que horas passa o autocarro?  
Acreditas em aliens?  
Acreditas no Pai Natal?  
Andas na droga?  
As delícias do mar são algum tipo de animal?  
Atiraste o pau ao gato?  
Como se chama o cão do teu vizinho?  
Conheces a professora Luísa Coheur?  
Conheces o HAL 9000?  
Consegues dizer sardinhas em italiano?  
Consegues lamber o cotovelo?  
Consegues morder as pontas dos pés?  
Conta-me uma anedota.  
Costumas fazer a barba?  
Dás-me o teu número de telemóvel?  
De que cor é o céu?  
Emprestas-me cinco euros?  
Eras capaz de saltar de uma ponte?  
És alérgico a lacticínios?  
És estrábico?  
És um fantasma?  
Existem extraterrestres?  
Falas espanhol?  
Fizeste a cadeira de língua natural no técnico?  
Gostas de andar a cavalo?  
Gostas de beber vinho?  
Gostas de comida picante?  
Gostas de engatar miúdas?  
Gostas de melancia?  
Gostas mais de Dire Straits ou de Iggy Pop?  
Gostavas de ir ver os aviões?  
Já comeste feijão de óleo de palma?  
Já deste um tiro em alguém?  
Já foste à Tailândia ver os elefantes?  
Já pensaste em usar um laço em vez de gravata?  
Já perdias uns quilos, não achas?  
Jogas Xadrez?

Não cansas de ficar ai em pé?  
Num contexto socio-económico, qual é a tua opinião de Portugal?  
O que achas da empresa Apple?  
O que comeste ao pequeno almoço?  
O que é maior, um sapato ou um carro?  
O que é uma maçã?  
O que jantaste no sábado passado?  
O que pensas sobre a morte?  
O que pesa mais, um kilo de lã ou um kilo de chumbo?  
O que queres ser quando fores grande?  
Onde é que estaciono o carro?  
Onde fica o Palácio da Pena?  
Onde posso adotar uma jiboia?  
Para que lado é o manicómio mais próximo?  
Porque é que a galinha atravessou a estrada?  
Porque é que o José Cid usa sempre óculos escuros?  
Porque é que tens mau feitio?  
Quais as tuas bandas preferidas?  
Quais são os teus medos?  
Qual a tua opinião sobre a terceira via do socialismo?  
Qual a velocidade média de voo de uma andorinha sem carga?  
Qual é a cor do cavalo branco do Napoleão?  
Qual é a distância da Terra à Lua?  
Qual é a frequência do espectro electromagnético que os teus olhos reflectem?  
Qual é a marca dos teus sapatos?  
Qual é a tua opinião das plataformas baseadas em linux?  
Qual é a tua opinião sobre a Coreia do Norte?  
Qual é o melhor clube de Portugal?  
Qual é o preço do azeite no pingo doce?  
Qual é o teu gelado favorito?  
Qual é o teu maior medo?  
Quando é que foi a última vez que tomaste banho?  
Quando é que fumaste o teu primeiro cigarro?  
Quantas flexões consegues fazer?  
Quantas janelas existem em Nova Iorque?  
Quantas patas tem uma centopeia?  
Quantas vezes bateste com a cabeça na mesa de cabeceira durante a noite?  
Quanto é dois mais dois?  
Que dia é hoje?  
Que ingredientes são precisos para fazer massa com atum?  
Que música costumavas ouvir?  
Que preferes mais: um murro nos olhos ou uma cabeçada na boca?

Quem é o filho dos meus avós que não é meu tio?  
Quem é o filho dos meus avós que não é meu tio?  
Quem é o gostosão daqui?  
Quem foi o primeiro rei de Portugal?  
Quem são essas moças atrás de ti?  
Quem veio primeiro, o ovo ou a galinha?  
Queres ir jantar?  
Queres matar todos os humanos?  
Sabes cantar?  
Sabes como ficavas bem? Com uma pedra de mármore em cima.  
Se estivesses numa ilha e só pudesses levar três coisas, o que escolhias?  
Se o mundo acabasse amanhã, o que farias hoje?  
Sentes-te confortável de fato?  
Será que me dás o teu tupperware?  
Tenho 500 moedas de 1 euro. Se trocar por uma só nota, qual o valor dessa nota?  
Tens algum vício?  
Tens carta de condução?  
Tens escova de dentes?  
Tens medo de trovoada?  
Usas lentes de contacto?  
Vais concorrer para presidente do teu país?

