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

João Santos
joao.l.santos@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

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

1. Introduction

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.

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.

2. Background

This section describes the background work originally done for the development of Say Something Smart [10]. We specify the creation of the SubTle Corpus [6], SSS's previous architecture and the work developed on Edgar [8].

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.

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 [2], 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** (M_1), which is measured through the calculation of the Jaccard similarity [4] 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), which evaluates the redundancy of each extracted entry by computing the Jaccard similarity between each pair of extracted answers; **Answer similarity with input** (M_3), that verifies each answer's similarity to the user query, once again, using the Jaccard similarity formula; and the **time difference between trigger and answer** (M_4), which gives a inversely proportional score to each entry according to the time elapsed between a T-A pair.

The final score of an Answer A_i given the user query u used to retrieve i,t 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:

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

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.

2.3. 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?".

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.

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 eachother.

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. Upgrading SSS into a Multiagent System

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.

3.1. 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 1.

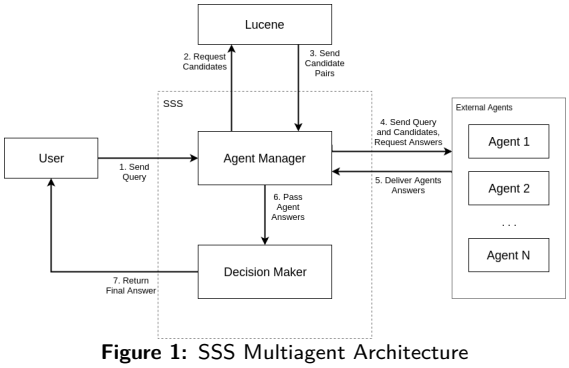

Figure 1: SSS Multiagent Architecture

Following the protocol set by SubTle, 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."*.

3.2. 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.

3.3. 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. Upon receiving an user query, this module 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.

3.4. 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.

3.4.1 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. Based on the results gathered by Fialho et al. (2016) [12], we chose three distinct lexical features and built agents based on them: *Cosine Similarity* [3], *Jaccard Similarity* [4] and *Levenshtein Distance* [5].

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.

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. 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, varying between 100/0, 75/25 and 50/50. 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, two sets of out-of-domain questions (simple and complex) were in turn evaluated and answered by both our system and SSS. Four annotators were given these questions and answers and annotated them according to their plausibility.

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 [7], 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, and so we computed the Approval rating of each system.

Regarding basic questions, our multiagent system had practically the same results as Magarreiro's, as shown in Table 1.

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

Table 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 2. 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 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 it com-

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

3.4.2 Case study B – David vs. Golias

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.

In a scenario where our system could guarantee that a certain agent's answer would be accurate, 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.

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 order to address the previously mentioned 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 backup, 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. 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. The first set was composed by out-of-domain requests gathered from past interactions with Edgar in the Monserrate Palace, and the second set was composed entirely by questions about the Monserrate Palace, as to test Edgar's accuracy on his domain.

Similarly to the previous case study, four annotators were given a sample of 50 questions and answers from each of the sets, respectively answered by Edgar on his own and by our system.

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. As shown in Table 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 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, as presented in Table 4.

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

Table 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.

3.5. Integrating the Chatbot on Discord

Discord[1] 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

[1] `https://discordapp.com/`

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.

For our Discord chatbot, we set Edgar as a Priority Agent 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.

4. Upgrading the Multiagent System with a Learning Module

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

4.1. Online Learning for Conversational Agents

In 2017, Mendonça et al. [9] 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 (EWAF) algorithm [11] 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 [13], 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.

4.2. Weighted Majority Algorithm

The implementation of the Weighted Majority 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 3. 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 α decimal places (α 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, $R^k(1, ..., t)$ as shown in the equation:

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

The variable η depends on the number of experts K, the expected number of iterations U, and a configurable parameter β, 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.

4.3. 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: 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.

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.

4.4. I Am Groot: Gauging the Learning Module's Effectiveness

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 and added a single additional agent, which we named *GrootAgent*.

GrootAgent is an agent that 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. 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.

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[2] of each agent at iteration 30 are presented in Table 5.

The results displayed in Table 5 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: Weight of each agent after 30 iterations in the Groot experiment.

4.5. 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 4.2.

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 4.4, with the exclusion of *GrootAgent*.

The results of the training can be observed in Figure 2, which displays the evolution of the weights of each agent throughout the whole learning process.

Through Table 6, 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 and *CosineAgent2* managed to stand relatively up to par, with its weight floating around the initial value during the first 10000 iterations, the second half of the training decreased their weight significantly, while *JaccardAgent2* became the most-weighted agent. Even more curious is the fact that neither of the other two Jaccard agents showed such

[2] The weights presented were rounded to three decimal places.

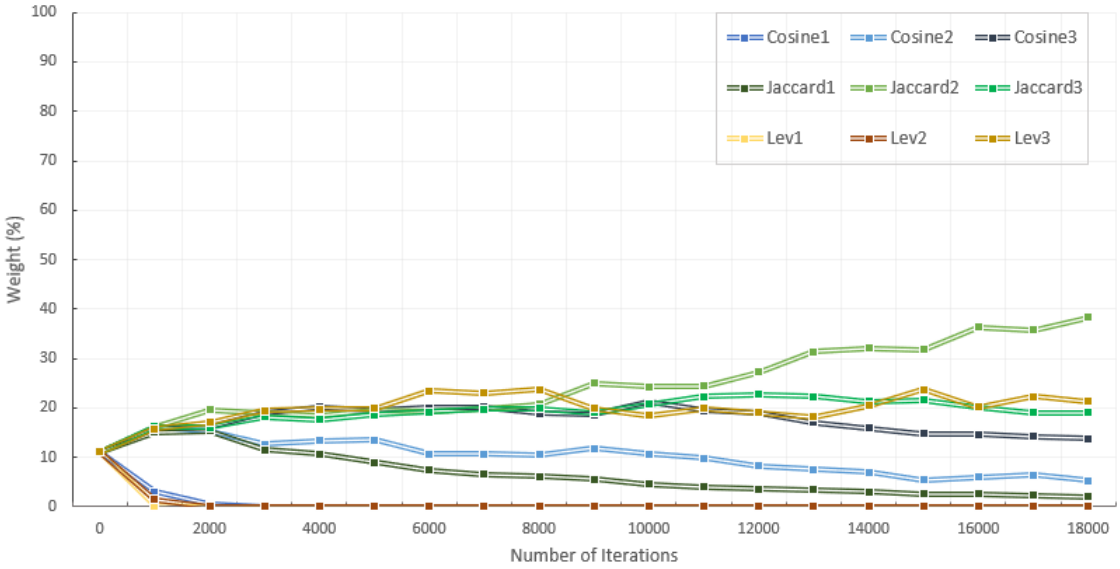

Figure 2: Evolution of the weight given to each agent throughout the 18000 iterations of training.

an improvement, which hints that, from these possibilites, 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.

Agent \ Iterations	5000	18000
CosineAgent1 (50/50)	0.004	0.000
CosineAgent2 (75/25)	13.551	5.361
CosineAgent3 (100/0)	19.461	13.863
JaccardAgent1 (50/50)	9.019	2.073
JaccardAgent2 (75/25)	19.461	**38.366**
JaccardAgent3 (100/0)	18.600	19.029
LevenshteinAgent1 (50/50)	0.000	0.000
LevenshteinAgent2 (75/25)	0.000	0.000
LevenshteinAgent3 (100/0)	19.906	21.307

Table 6: Weight of each agent throughout the 18000 iterations of training.

4.6. 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.

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.

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 6.
- 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 6) in the Weighted Vote, which is the same as performing a normal Voting Model.

System	Accuracy
SSS	90.6%
SSS + Learning	**95.4**%
MultiSSS 18000	93.8%
MultiSSS Equal	93.6%

Table 7: Accuracy of the systems when evaluated with 2000 interactions of the CMD corpus.

Through the information presented in Table 7, 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 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.

4.7. Evaluating the Systems with Human Annotators

As our final experiment, we decided to follow up from the procedure described in Section 3.4.1 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 delivering them to humans who annotated their answers according to their plausibility.

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

Table 8: Mendonça et al.'s system against our trained system when answering basic questions.

Regarding basic questions, as shown in Table 8 our system vastly outperformed SSS + Learning. On the matter of complex questions, as shown in Table 9, 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 8, 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.

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

Table 9: Mendonça et al.'s system against our trained system when answering complex questions.

5. Conclusions and Future Work

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.

5.1. 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.

5.2. 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.

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 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] Singhal, A. & Google, I., (2001). Modern Information Retrieval: A Brief Overview. IEEE Data Engineering Bulletin. 24.

[4] Jaccard, P., (1912). The distribution of the flora in the alpine zone. New Phytologist, 11(2):37–50.

[5] Navarro, G., (2000). A Guided Tour to Approximate String Matching. ACM Computing Surveys. 33. 10.1145/375360.375365.

[6] Ameixa, D., & Coheur, L. (2013). From subtitles to human interactions : introducing the SubTle Corpus. Tech. Rep. 1 / 2014 INESC-ID Lisboa, February 2014

[7] 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).

[8] 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)

[9] 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)

[10] 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.

[11] 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

[12] Fialho, P., Marques, R., Martins, B., Coheur, L. & Quaresma, P. (2016). INESC-ID@ASSIN: Measuring semantic similarity and recognizing textual entailment. 8. 33-42.

[13] 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