UNIVERSITAT
ROVIRA i VIRGILI

Introduction to Multi-Agent Systems Implementation - Second Delivery

Team 5

Victor Gimenez Abalos - victor.gimenez.abalos@est.fib.upc.edu
Daniel Felipe Ordonez Apraez - daniel.ordonez@est.fib.upc.edu
Albert Rial Farràs - albert.rial@est.fib.upc.edu
Jordi Armengol Estapé - jordi.armengol.estape@est.fib.upc.edu
Joan Llop Palao - joan.llop@est.fib.upc.edu

January 15, 2020

Contents

1 Introduction

The aim of this project is to create an agent-based decision support system (A-DSS). This system is able to interact with a human user, receive a configuration, train with a dataset specified on that configuration and finally predict.

For the theoretical understanding of multi-agent systems, we base our work on [1], [2] and [3]. For the practical part, apart from the JADE guide found in the Moodle of this course, we have used [4] and [5].

This proposed A-DSS system includes three different agents: one User, one Manager and a set of Classifiers.

- **User agent**
  This agent operates as the main interface with the system user, by providing the capability of executing actions on command, e.g. the system user can request a *training* or *prediction* action that will affect the classifier agents.
- **Manager agent**
  The main task of this agent is to control and coordinate the operation of the classifier agents in order to fulfill the commands of the system user. Its objective is to mediate and to behave as a strong learner of an ensemble machine learning method, merging information from other weak learners. Regarding the commands it can perform, we have training and prediction actions. In the case of a *training action* command the manager should coordinate the training of all classifier agents with its correspondent hyper-parameters. Depending on the resultant accuracy of each of them decide on a metric to agglomerate/combine the multiple output, once a *prediction action* command is received.
- **Classifier agents**
  The classifier agents will behave like the weak learners in a metalearning algorithm. They work by encapsulating supervised machine-learning models which have the capability to be trained on an specific data-set and predict with a certain accuracy the nature of unseen data.

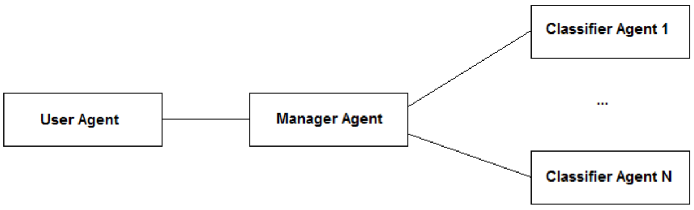

Figure 1: System architecture displaying the interaction between the proposed agents. Source: Assignment statement.

1.1 Document structure

The document is organized as follows: Section 2 presents all the implementation details. Section 3 provides the results of different methodology along with all experiments performed with the proposed models. Finally, conclusions and a description on future work is provided in the Section 4. At the end of the document we have also added, as appendices, our E-Portfolio and the instructions on how to compile and run the code.

2 Implementation details

The implementation of this project has been done with Java, a programming language for object-oriented programming.

To implement the agents we have used JADE (Java Agent Development Framework), a software framework for the development of intelligent agents and implemented in JAVA. This framework allows the coordination of the agents following the FIPA protocols and the use of standard FIPA communication language.

The other resource used in this project has been Weka, a collection of open source machine learning algorithms for data mining tasks.

Next we will see the details of communication between agents and the logic inside them. We can advance that in our system the logic resides in the agents and all the communication is done by the behaviours.

2.1 Communication

The agent communication paradigm in this project is quite simple. We base ourselves on the protocol specifications from FIPA, and structure a hierarchy of agents such that they have very well defined roles in the dialog.

Particularly, we use the FIPA Request protocol[1], which we have implemented using three separate and reusable behaviours: one for the receiver (or participant), one for the initiator and the other one for the agent that is both (receiver and initiator). The idea was to only have two behaviors, one for the initiator and other for the receiver. However, we were unable to do such approach as we did not find the way to block only one behaviour (not all the behaviours of the agent). The FIPA Request protocol's sequence diagram can be seen in Fig. 2.

[1] http://www.fipa.org/specs/fipa00026/SC00026H.html

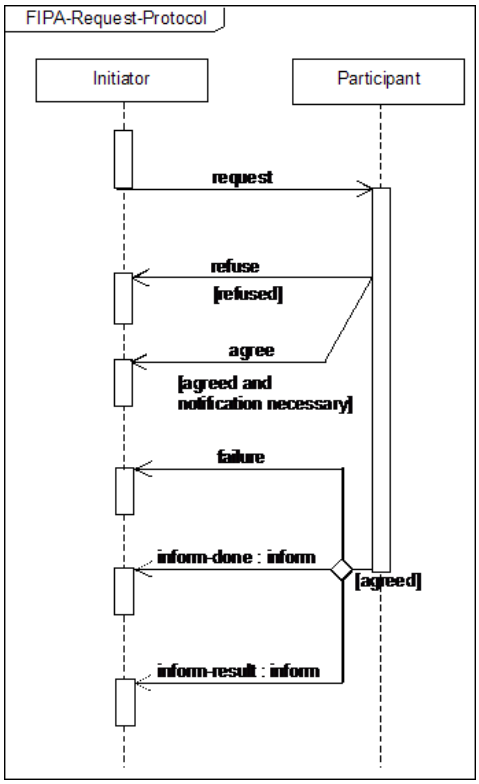

Figure 2: Official FIPA-Request Protocol specification. Source: FIPA (`http://www.fipa.org/specs/fipa00026/SC00026H.html`).

In our problem, we will have communication between the user and the manager agents (in the roles of initiator and receiver, respectively), and between the manager and the classifier agents (again, as initiator and receiver respectively).

We will detail the particular details of the communication mechanisms (*i.e.* the actions performed at each step, as well as the particulars of each behaviour applied to the agents) in each agent's corresponding subsection.

2.2 Agent details

In this section we comment the logic and specific communication/states of each of the agents.

2.2.1 User Agent

As we have explained in the previous section, the User Agent is the one who is capable of reading the user input. When the action of the user arrives it sends a request to the Manager agent in order to perform it.

To do so, it has two behaviours associated:

- *ReadUserInputBehaviour*: is a cyclic behaviour that, as its name implies, is continuously reading the user input. When the user enters a command, it parses the command to get the action and, in case of training, reads the configuration file specified. When it has all the input it tells the

agent to start the action. Below, in Figure 3, we can see a sequence diagram explaining the main actions of the behaviour.

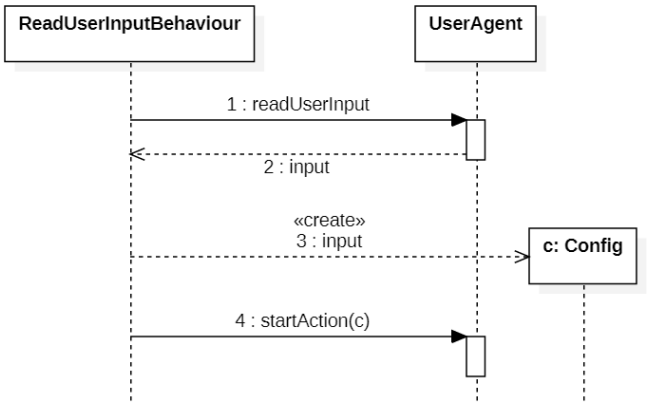

Figure 3: Sequence diagram of the ReadUserInputBehaviour. Source: Own elaboration.

- *FIPAInitiatorBehaviour*: is also a cyclic behaviour that is in charge of initiating the FIPA Request communication protocol. Basically, it gets the configuration and the action obtained by the previous behaviour and passes it as a serializable object to the Manager. In fact, the one that will receive the message will be the behaviour FIPAInitiatorReceiverBehavior associated to the Manager agent.
  After that, the behaviour will change its state to *Waiting* (Figure 5). In this state, the behaviour will block the agent until it receives a response, using *blockingReceive*. When a response arrives to the behaviour, it will analyze its performative and call the corresponding function of the User agent. If the response is a *refuse* or a *failure*, the User agent will show the error associated with it. If is an inform, it will show the result. In all of the previous cases, it will return then to the *Idle* state, where the agent is not blocked and is able to read user input. However, if we receive an agreed, the state will remain the same, waiting for a *failure* or an *inform*.

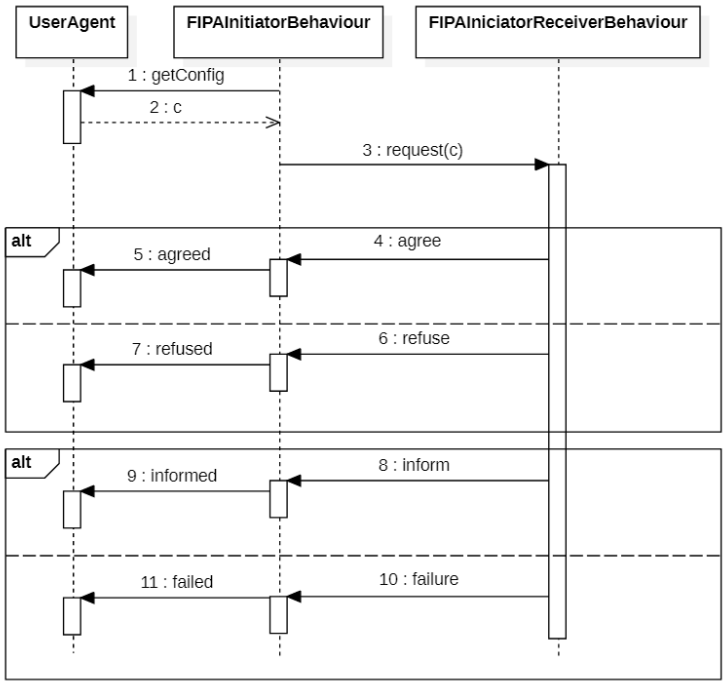

Figure 4: Sequence diagram of the FIPAInitiatorBehaviour. Source: Own elaboration.

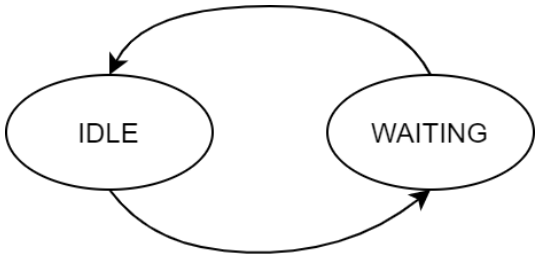

Figure 5: State flow of the FIPAInitiatorBehaviour

2.2.2 Manager Agent

The manager agent is the one responsible for the classifiers coordination and interfacing with the user. It does so through having the Fipa Initiator/Receiver Behaviour, which receives a request from the User agent and communicates and interacts with the Classifier agent. Fig. 6 and 7 describe the two phases of this behaviour.

How the Initiator/Receiver Behaviour works is the following: first, the manager receives a request (from the user agent, the initiator behaviour). In order to be able to accept it, the manager must be in the *Idle* state, and it must also check that the action received is valid for the manager. If any of the above do not check out, the manager rejects the request. If it doesn't, it will prepare the ground for sending the requests to classifiers. Once these actions have been performed, the behaviour sends all of the requests, keeping track of how many requests have been sent, and switches to the *Waiting* state. If any of the above, fail, it will send a *Reject* message to the requester agent. This process can be seen in Fig. 6 and 8.

When we talk about checking the action, we refer to checking whether the system is in a state in which the action can be performed. Particularly, we will reject the action of predicting if we have not trained yet. It will also reject it if we find any exception during the processing of the message, or if the message content is not well formatted. On the other hand, when we refer to preparing the ground, we are talking about possible classifier agent instantiation (when the action is train): it will check how many classifier agents already exist and, if the request sent by the user demands more, it will create them before proceeding. It will not, however, kill extra classifiers if the action demands less. Instead, it will not send them requests, but will keep them for future use in case more are demanded. This is a design decision for avoiding possible overhead from agent creation.

The requests sent to the classifiers contain the required data for training or predicting, depending on the original user request. The manager is in charge of distributing the required amount of instances to each of the classifiers, randomly picked from the dataset. In this way, the classifiers do not have access to the dataset, and therefore are not capable of looking at other instances. Once the requests are sent, we wait for the results. While we are waiting, we count the number of responders (*i.e.* the classifiers that agree to our request) and of informs (*i.e.* the number of classifiers which are reportedly done with their request). Once the number of informs, responders and requested classifiers match, the manager changes state to *Success* and starts to process the results, which is our prediction aggregation stage. If we receive a *Failure* or *Reject* message from a classifier, we will abort the operation, returning the error to the user agent. We also note that we assign a 5 second timeout for receiving the informs from the classifiers, in order to ensure that the system does not freeze. All of these are summarised in Fig. 7.

Regarding result aggregation for the prediction stage, we have implemented a simple voting mechanism. We debated several times about the possibility of implementing more sophisticated systems, principally auction mechanisms as described by previous deliveries. However, in the current stage of the system, this mechanism would simply cause overhead, and is not as supported by statistics, therefore we believe it would worsen our results. We talk about the more sophisticated alternatives in Section 4.

The voting mechanism works as follows: we compute a weighted sum for each possible prediction. Formally, $p(class) = \sum_{C \in classifiersthatvotedclass} I(C) / \sum_{C \in classifiers} I(C)$, where $I(C)$ is the number of instances that the classifier receives. This voting mechanism comes from the fact that the number of instances the classifier has trained with lowers its bias, and therefore should be trusted more. However, this could also cause overfitting, and we thus check with the other weak learners. If the weak learners disagree with classifiers that have high instances, we are able to detect this possible overfitting and correct it. There is a lot of literature supporting this claim, mainly the Ensemble methods of ML. The agreed prediction is the one maximizing $p(class)$. In case of a tie, we return either prediction, as we cannot decide which is the correct one.

Once the results are aggregated, the result of the aggregation, as well as the accuracies of each individual classifier, are sent to the user agent in the final reply of the manager to the initial query.

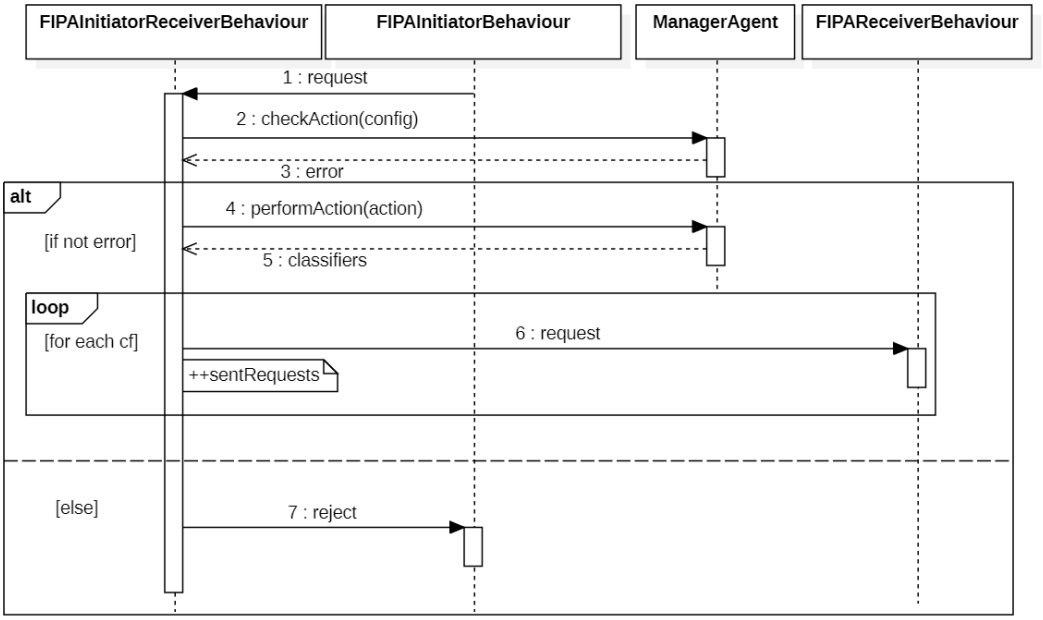

Figure 6: Sequence diagram of the FIPAInitiatorReceiverBehaviour, part 1. Source: Own elaboration.

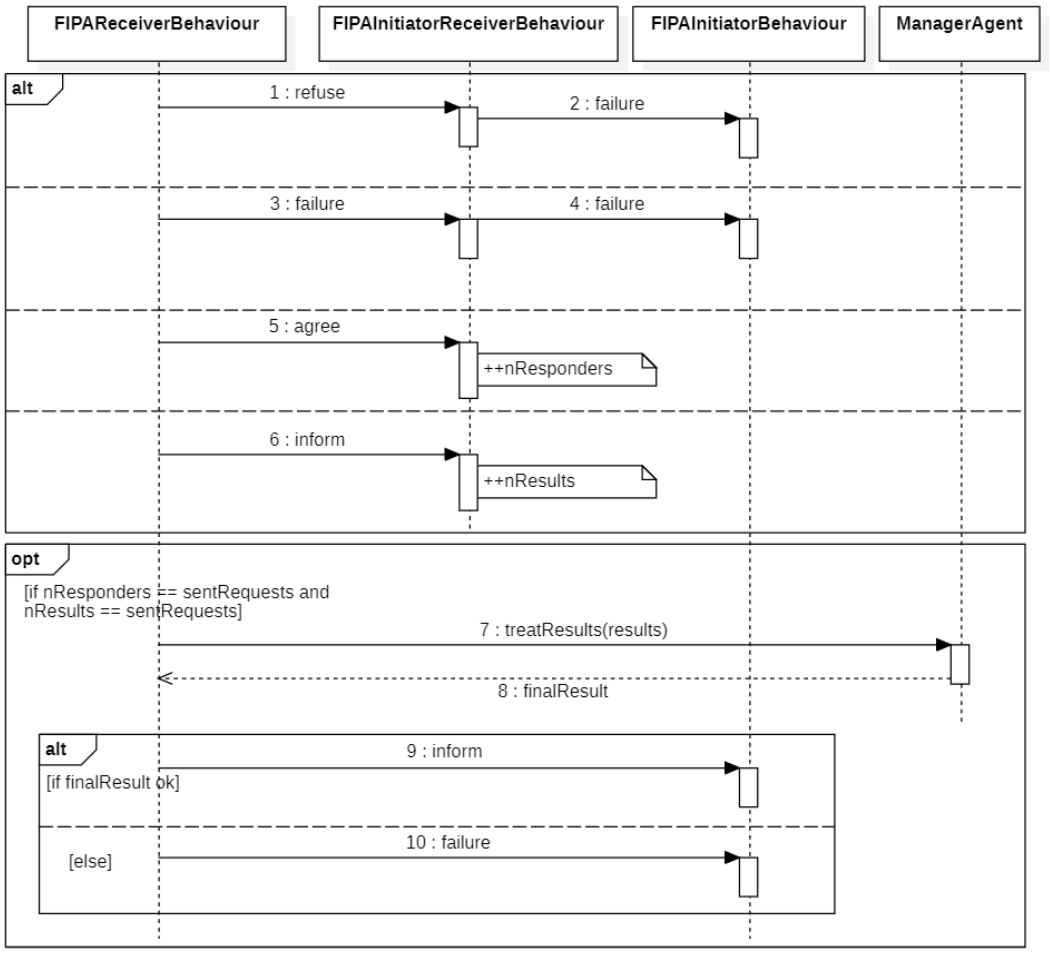

Figure 7: Sequence diagram of the FIPAInitiatorReceiverBehaviour, part 2. Source: Own elaboration.

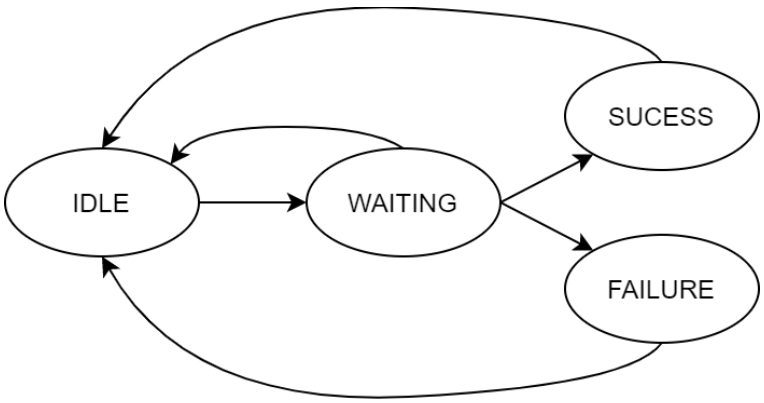

Figure 8: State flow of the FIPAInitiatorReceiverBehaviour. Source: Own elaboration.

2.2.3 Classifier Agent

The classifier agent performs most of the machine learning logic of the model. It is a simple agent that is in an idle state most of the time, only acting upon receiving requests from its manager. It does so by having our custom *FIPAReceiverBehaviour*, which checks and performs the actions received in incoming messages.

Once a message arrives, the classifier checks whether it is correctly formatted (*i.e.* the performative is a *request*, the first character of its content is 'T' or 'P' and the second is an underscore. If all of the above apply, we also parse the rest of the content and dump it into the agent's own parameters: in the case of a train message, it stores the classifier to use and the data for training, whereas in the case of predict it stores the data for predicting. After this is done, the classifier agent replies to the message with an *agree* performative. In case the performative is not *request*, or the content could not be parsed, the classifier agent replies with a *refuse* performative. Originally, we thought of using the *not understood* performative, since it seems to fit better, but we changed it to refuse to better comply with the FIPA Request protocol: both should be valid in any case, but we doubt on whether we should use one or the other.

Once the reply is sent, it performs the training and/or prediction action depending on the message. Once the training/prediction finishes, the classifier moves from the *Idle* to the *Failed*/*Success* states, depending mainly on whether the action could be performed or not. The reasons why it could return an error are mainly exceptions, but we also leave open the possibility of an 'Algorithm not supported' clause, where the manager sends a request to train with an algorithm the classifier does not know. Particularly, the classifier can use three algorithms: J48, IBk and a multilayer perceptron. We could return a refuse in the previous stage, since we know at the moment of receiving the message whether we support the requested algorithm or not, but in this way it allowed us to test our communication protocols more in depth.

Finally, we reply to the message based on the current state the classifier is in. If it is failed, it sends a reply to the request message with the failure performative, whereas if it is success it will send the inform performative. Moreover, the content of the inform will be 'Classifier trained' for the training request, and the results of the classifier's prediction on the test set (as an arraylist of strings serialised).

Once the final reply is sent, the classifier returns to the *Idle state*, to receive more messages.

A more detailed snippet on the sequence of actions and the relationship between the behaviour and the agent class can be seen in Fig. 9. Also, the state authomaton of the behaviour can be seen in Fig. 10.

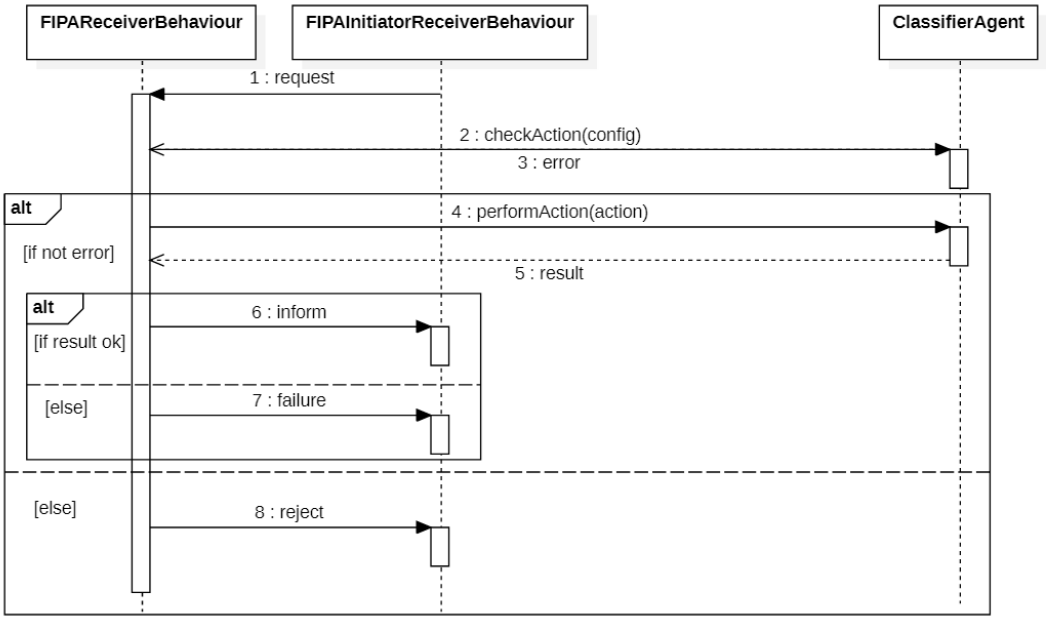

Figure 9: Sequence diagram of the FIPAReceiverBehaviour. Source: Own elaboration.

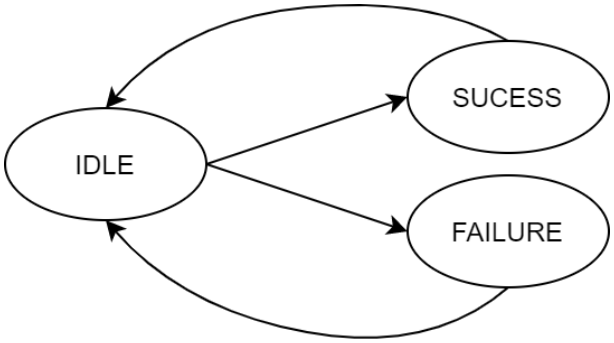

Figure 10: State flow of the FIPAReceiverBehaviour. Source: Own elaboration.

2.3 Changes from our initial expectations

Initially, we expected the distributiveness implementation requirement of this project to result in an artificially imposed constraint that limited its potential. Nevertheless, we ended up understanding the usefulness of developing our system with agents. In particular, agents proved to be a natural fit at least for the manager and the classifiers. Moreover, we believe that, with the knowledge gained from the subject, we would be able to further develop this system into something more interesting: having the manager be a mobile agent that goes to gather information from different classifiers could be an example of such. Other things we believed would be interesting are implementing auction based scores for trusting the classifiers (as detailed in the second activity), or making the classifier creation more dynamic.

As far as the implementation is concerned, we started with a mostly pseudo-monolithic architecture for the reason stated before, but as soon as our understanding of JADE improved, our code started

to be more truly agent-oriented. This improved understanding also allowed us to refactor our project such that behaviours were extracted from the agents and made more generic for re-usability purposes.

3 Results

3.1 Experiment 1

We started by executing the first required experiment: a full execution with 2 classifiers. The specific configuration file is fully specified in Figure 11.

Figure 12 shows the JADE GUI with two classifier agents. Then, in Figure 13, we can observe the communication between agents thanks to the Sniffer agent. Finally, Figure 14 shows the user commands and the system output.

As we can observe, the system successfully handles this simple, yet complete, use case. The resulting accuracy is high, which is no surprise taking into account the datasets used in this project. Some of the classifiers do not obtain the maximum accuracy, but the overall accuracy is 100%. The output informs of both the statistics of each classifier and the final decision. Notice that the two classifiers give different predictions for the first instance, which results in a tie. In this case, they have been trained with the same number of instances, so we resolve the tie by taking the prediction of the first classifier.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<SimulationSettings>
<title>IMAS_basic_simulation</title>
<algorithm>J48</algorithm>
<classifiers>2</classifiers>
<trainingSettings>80,80</trainingSettings>
<classifyInstances>5</classifyInstances>
<file>fertility_Diagnosis_train.arff</file>
</SimulationSettings>
```

Figure 11: Example 1: Configuration.

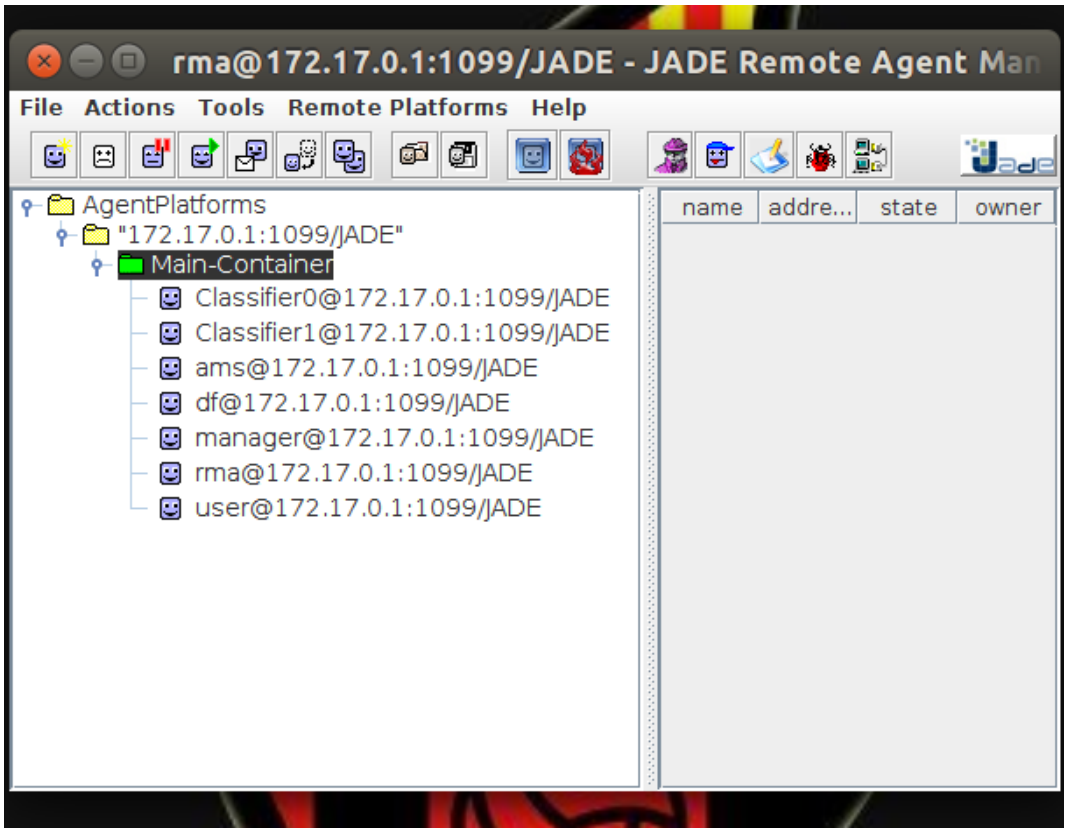

Figure 12: Example 1: JADE GUI showing the Agents container.

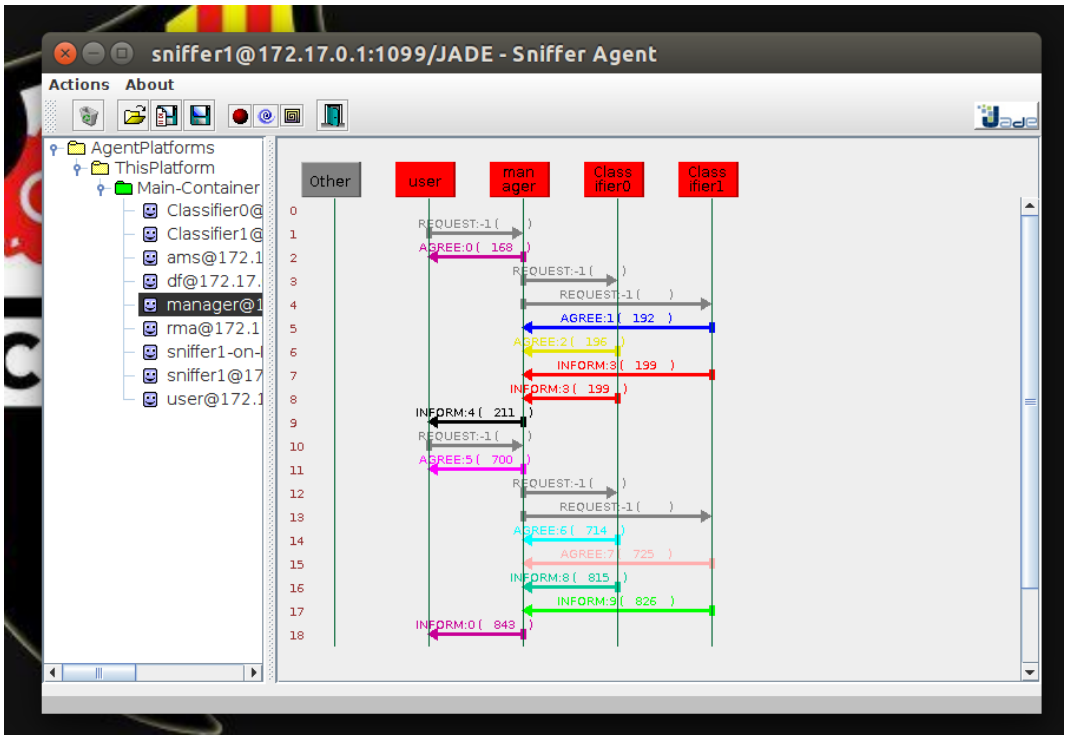

Figure 13: Example 1: Communications intercepted by the Sniffer agent.

```
T
Jan 15, 2020 3:45:34 AM agents.UserAgent receivedInform
INFO: Classifiers have been trained
P
Jan 15, 2020 3:45:36 AM agents.UserAgent receivedInform
INFO:  Table with the results
  Instance                    Data                    Agent 0          Precission Agent 0          Agent 1          Precission Agent 1      Final Decision
     0            -1,0.78,1,0,1,0,1,-1,0.25,N              N                    80.00 %                0                    80.00 %                 N
     1             -0.33,0.5,1,0,0,0,1,-1,0.5,N            N                    80.00 %                N                    80.00 %                 N
     2           -0.33,0.86,1,1,1,1,1,-1,0.25,N            N                    80.00 %                N                    80.00 %                 N
     3           -0.33,0.83,1,1,1,0,1,-1,0.31,N            N                    80.00 %                N                    80.00 %                 N
     4             -1,0.67,0,0,1,0,0.6,0,0.5,O             O                    80.00 %                O                    80.00 %                 O
                                                  Accuracy classifier 0: 100.00 %          Accuracy classifier 1: 80.00 %       Accuracy: 100.00 %
```

Figure 14: Example 1: User inputs and resulting output.

3.2 Experiment 2

For the next execution, we were asked to use 5 classifiers. Figure 15 shows the specific configuration file that we used. Then, in Figure 16, we present the resulting input, given the user commands. The overall accuracy is, again, 100%, but with another dataset. In more challenging datasets, we expect that increasing the number of classifiers should improve the global accuracy.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<SimulationSettings>
<title>IMAS_complete_simulation</title>
<algorithm>J48</algorithm>
<classifiers>5</classifiers>
<trainingSettings>100,200,300,400,500</trainingSettings>
<classifyInstances>10</classifyInstances>
<file>segment-test.arff</file>
</SimulationSettings>
```

Figure 15: Example 2: Configuration.

```
Example 2 settings
Jan 15, 2020 3:58:46 AM agents.UserAgent receivedInform
INFO: Classifiers have been trained
p
Jan 15, 2020 3:58:47 AM agents.UserAgent receivedInform
INFO: Table with the results
   Instance       Data  Agent 0  Precission Agent 0  Agent 1  Precission Agent 1  Agent 2  Precission Agent 2  Agent 3  Precission Agent 3  Agent 4  Precission Agent 4  Final Decision
0         0  brickface  brickface             12.35 %  brickface             24.69 %  brickface             37.04 %  brickface             49.38 %  brickface             61.73 %  brickface
1         1     window     window             12.35 %    window             24.69 %    cement             37.04 %    cement             49.38 %    window             61.73 %    window
2         2     cement     cement             12.35 %    cement             24.69 %    cement             37.04 %    cement             49.38 %    cement             61.73 %    cement
3         3  foliage    foliage             12.35 %  foliage             24.69 %  foliage             37.04 %  foliage             49.38 %  foliage             61.73 %  foliage
4         4     grass     grass             12.35 %     grass             24.69 %     grass             37.04 %     grass             49.38 %     grass             61.73 %     grass
5         5     grass     grass             12.35 %     grass             24.69 %     grass             37.04 %     grass             49.38 %     grass             61.73 %     grass
6         6     cement     cement             12.35 %    cement             24.69 %    cement             37.04 %    cement             49.38 %    cement             61.73 %    cement
7         7        sky        sky             12.35 %       sky             24.69 %       sky             37.04 %       sky             49.38 %       sky             61.73 %       sky
8         8     grass     grass             12.35 %     grass             24.69 %  foliage             37.04 %     cement             49.38 %     grass             61.73 %     grass
9         9     window     window             12.35 %    window             24.69 %    cement             37.04 %    cement             49.38 %    window             61.73 %    window
                                Accuracy classifier 0: 100.00 %      Accuracy classifier 1: 100.00 %      Accuracy classifier 2: 70.00 %      Accuracy classifier 3: 70.00 %      Accuracy classifier 4: 100.00 %      Accuracy: 100.00 %
```

Figure 16: Example 2: User inputs and resulting output.

3.3 Experiment 3

To test our system in other scenarios, we decided to experiment with additional configurations.

In the third experiment, we run the same configuration three times, but with different classification algorithms:

- IBK: Instance-based learning (K-NN). Figure 17 shows the corresponding configuration, while Figure 20 shows the resulting outputs.
- J48: The default classifier. In figures 17 and 21 we present the used configuration file and the resulting output, respectively.
- MLP: Multi-Layer Perceptron. Figures 19 and 22 show the used configuration file and the corresponding output, respectively.

Our additional implementation of being able to use different classification algorithms (specified by the user in the configuration file) works as expected. Since the used dataset, Segment, is not particularly challenging, and we are augmenting the algorithms with ensemble learning (voting), all of them obtain very high global accuracies and therefore are difficult to compare. MLPs present a slightly worse performance, perhaps because this algorithm needs more data to generalize properly, but the difference is almost negligible.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<SimulationSettings>
<title>IMAS_model_comparison_IBk</title>
<algorithm>IBk</algorithm>
<classifiers>8</classifiers>
<trainingSettings>100,150,200,250,300,400,450,500</trainingSettings>
<classifyInstances>20</classifyInstances>
<file>segment-test.arff</file>
</SimulationSettings>
```

Figure 17: Example 3: IBK execution configuration file.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<SimulationSettings>
<title>IMAS_model_comparison_J48</title>
<algorithm>J48</algorithm>
<classifiers>8</classifiers>
<trainingSettings>100,150,200,250,300,400,450,500</trainingSettings>
<classifyInstances>20</classifyInstances>
<file>segment-test.arff</file>
</SimulationSettings>
```

Figure 18: Example 3: J48 execution configuration file.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<SimulationSettings>
<title>IMAS_model_comparison_MLP</title>
<algorithm>MLP</algorithm>
<classifiers>8</classifiers>
<trainingSettings>100,150,200,250,300,400,450,500</trainingSettings>
<classifyInstances>20</classifyInstances>
<file>segment-test.arff</file>
</SimulationSettings>
```

Figure 19: Example 3: MLP execution configuration file.

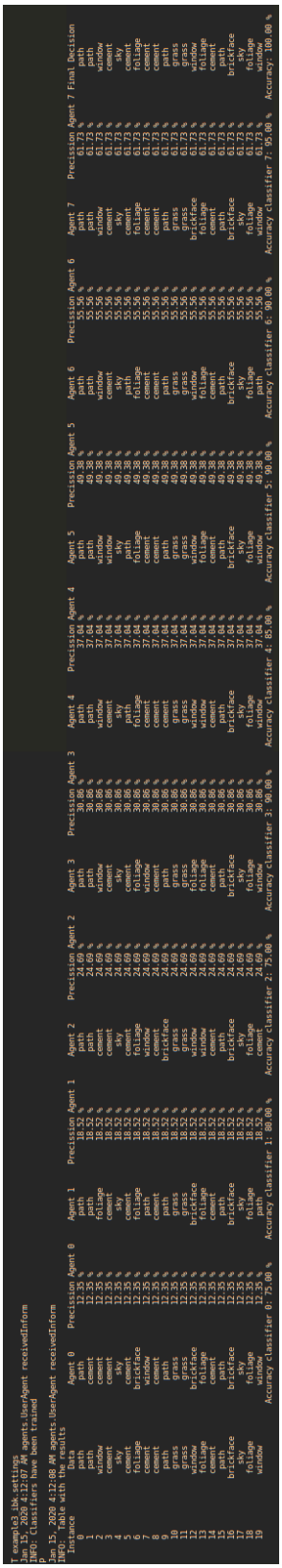

Figure 20: Example 3: User inputs and resulting output with IBK classifiers.

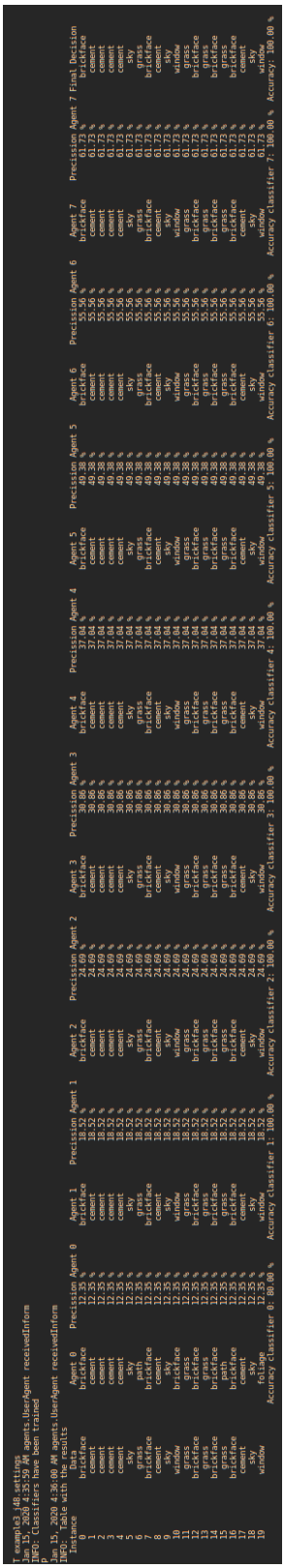

Figure 21: Example 3: User inputs and resulting output with J48 classifiers.

Figure 22: Example 3: User inputs and resulting output with MLP classifiers.

3.4 Experiment 4

The previous experiments showed that our system works in simple cases, but do not answer the question of whether it scales to a considerably larger number of classifiers.

With as many as 50 classifiers, as specified in the configuration file shown in Figure 23, our system keeps working as expected. We believe that we have evidence to believe that our system is robust and scales to high numbers of classifiers.

Table 3.4 shows the accuracy of each classifier, and the resulting overall accuracy is obviously 100%. In this case, it is quite obvious that we do not need so many classifiers, but the purpose of this test was assessing the behaviour of our system with a high number of concurrent classifiers.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<SimulationSettings>
<title>IMAS_large_nb_classifiers</title>
<algorithm>J48</algorithm>
<classifiers>50</classifiers>
<trainingSettings>500,501,518,527,556,570,590,609,615,632,644,655,
<classifyInstances>20</classifyInstances>
<file>segment-challenge.arff</file>
</SimulationSettings>
```

Figure 23: Example 4: Configuration file.

Classifier	Accuracy
1	95%
2	100%
3	95%
4	95%
5	95%
6	100%
7	100%
8	100%
9	100%
10	95%
11	100%
12	95%
13	100%
14	95%
15	100%
16	95%
17	90%
18	100%
19	95%
20	95%
21	100%
22	95%
23	100%
24	100%
25	100%

Classifier	Accuracy
26	100%
27	95%
28	95%
29	100%
30	100%
31	100%
32	100%
33	100%
34	100%
35	100%
36	95%
37	95%
38	100%
39	100%
40	100%
41	100%
42	100%
43	100%
44	100%
45	100%
46	95%
47	100%
48	100%
49	100%
50	100%

Table 1: Example 4: Results.

3.5 Experiment 5

Finally, we wanted to force our system to fail. In particular, we introduced an unsupported classification algorithm, J49, in the configuration file, as shown in Figure 24.

Figure 25 shows the system output. It is important to note that the failure message that says "Algorithm not supported" is created by the classifiers, and propagated to the User agent. The same happens with the message "Classifiers not ready", that is created by the Manager agent and sent to the User, that is the only agent that prints messages.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<SimulationSettings>
<title>IMAS_basic_simulation</title>
<algorithm>J49</algorithm>
<classifiers>2</classifiers>
<trainingSettings>80,80</trainingSettings>
<classifyInstances>5</classifyInstances>
<file>fertility_Diagnosis_train.arff</file>
</SimulationSettings>
```

Figure 24: Example 5: Configuration.

```
T example5.settings
Jan 15, 2020 5:09:26 AM agents.UserAgent receivedFailure
WARNING: FAILURE
Algorithm not supported
P
Jan 15, 2020 5:09:28 AM agents.UserAgent receivedRefuse
WARNING: REFUSED
Classifiers not ready
```

Figure 25: Example 5: Error messages.

4 Conclusions

From our results and the extensibility of this project, we can appreciate the usefulness of agents beyond the current computational paradigm. Although this project could be perfectly coded by any ML library and language without using agents, the current system architecture allows for much more complex and interesting applications. We envision the applicability of this project as a MAS with several owners, having their own agents, and several datasets, in which the manager is a mobile agent that can locate useful classifier agents throughout the net (rating their usefulness via knowing accuracy or trustworthiness of the classifiers). This would result in a distributed network of leasable classifiers with different capabilities (*i.e.* knowledge of different domains, and implementing different classifiers). This could, in turn, become a platform for private machine learning network, in which each owner devises their agents in a way that they earn reputation (via exploiting new machine learning methods) or even currency (real or in the system). Through these means, we could transform our manager agents into the more proactive *Information* agents, which behave similarly to the BDI paradigm (belief of the classifier's trustworthiness, desire to improve reports and to be more precise in beliefs, intention of testing classifiers and/or providing the answer to specific user queries).

We note that, despite the current implementation not being capable of doing this, the design of the system sets a hopeful expectations toward the expansion into such intriguing applications.

Our results on the dataset show what we expected: an ensemble of weak learners (classifiers) becomes a stronger learner (manager) that is able to better support decisions of the owner (user). This basic

multiagent system is also extensible to more complex requirements, and has been programmed such that it is reusable in other MAS.

Personally, we have learnt about the agent paradigm through this project: how to communicate information across a MAS, how to program behaviours and encode agent states, and what the limitations are. We are excited to see the possibilities the agent paradigm opens with respect to traditional OOB, particularly in the proactiveness and distributableness of the agents in order to achieve objectives.

A System execution

In this sections we detail the instructions for executing our system in Windows and Linux.

A.1 Instructions for executing our system in Windows:

1. Compile the agents:

   ```
   javac -classpath lib\jade.jar;lib\weka.jar -d src\out\production\IMAS_test\ src\src\agents\*.jav
   src\src\behaviours\*.java src\src\utils\*.java
   ```

2. Execute Jade with User Agent (the rest will be created dynamically):

   ```
   java -cp lib\jade.jar;lib\weka.jar;src\out\production\IMAS_test\ jade.Boot -gui -agents
   user:agents.UserAgent
   ```

3. Use the command line to train or predict:

   ```
   USAGE: T <config_file> | P
   Example: T imas.settings
   ```

A.2 Instructions for executing our system in Linux:

1. Compile the agents:

   ```
   javac -cp lib/jade.jar:lib/weka.jar -d src/out/production/IMAS_test/ src/src/agents/*.java
   src/src/behaviours/*.java src/src/utils/*.java
   ```

2. Execute Jade with User Agent (the rest will be created dynamically):

   ```
   java -cp lib/jade.jar:lib/weka.jar:src/out/production/IMAS_test/ jade.Boot -gui -agents
   user:agents.UserAgent
   ```

3. Use the command line to train or predict:

   ```
   USAGE: T <config_file> | P
   Example: T imas.settings
   ```

B E-Portfolio

B.1 Task distribution

As we said in previous deliveries, managing a group of 5 members is a non-trivial task. For this reason, we have carefully divided work into subtasks and assigned them to 5 members. However, the general tasks like writing the report, preparing the slides and testing the system has been done by all members. Specifically, the task distribution has been the following:

- Víctor developed the Classifier agent.
- Daniel and Albert developed the User agent and did the refactoring to separate Behaviours and have a cleaner code.

- Joan and Jordi developed the Manager agent and, by consequence, the voting mechanism to select the class of an instance based on the predictions of each Classifier.

However, it is very important to notice the following considerations:

- Even though the tasks done can be reviewed in the commit history, it is worth noting that when working in group, it could happen that one member of the sub-group did not appear in the commits (*e.g.* in cases in which the sub-group met physically and the commits were done by one of the members of the sub-group).
- At the end of the day, since the agents implementation was not completely independent, everyone ended up a bit of other agents, apart from the assigned one. This is particularly true for the refactoring of behaviours which involved several agents.
- All the members also ended up participating on the refactoring of the code to make behaviours reusable.

For holding the tasks we use also the repository in Github. The repository holds organizational information for each of the agent types considered, and for all of the features, bugs and enhancements during the process of design and development.

Figures 26 and 27 depict the open and closed issues in the repository, indicating abstractly which features were already addressed and which were still missing.

Finally, Figures 29, 30 and 31 show the status of development of each of the agent types considered. The Figure 28 shows the status of the general tasks, like writing the report or doing the refactoring and testing.

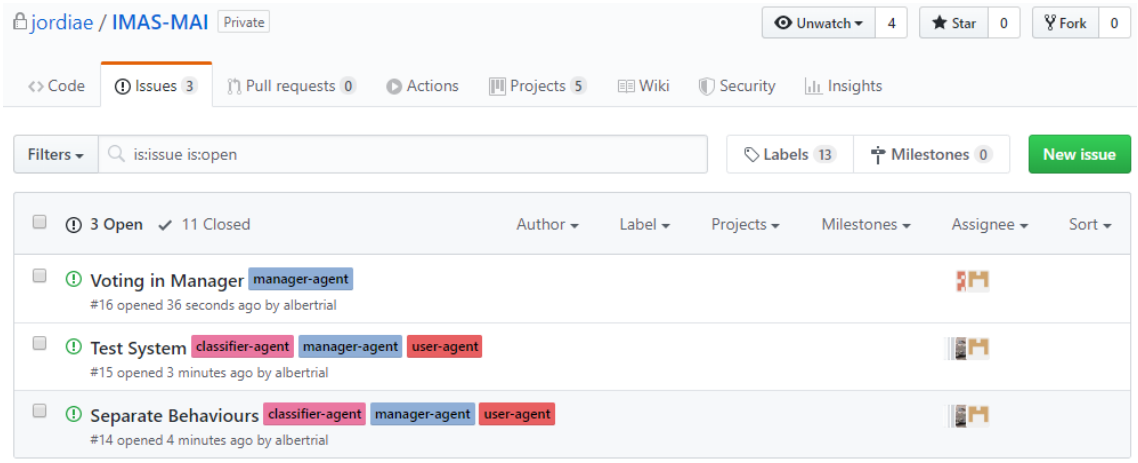

Figure 26: Open issues in project repository

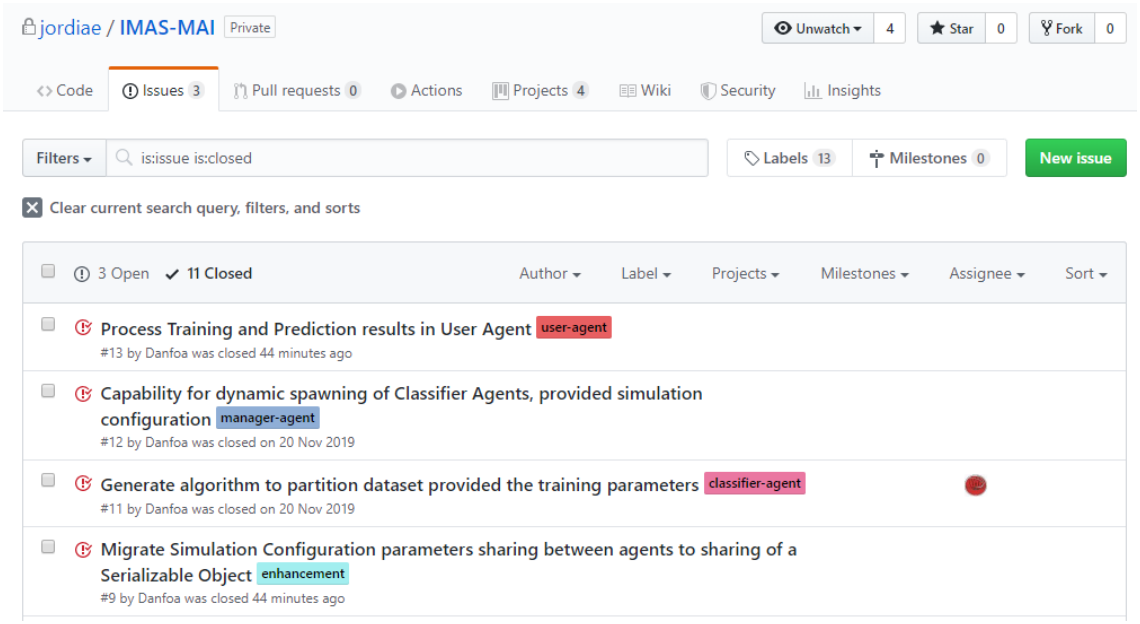

Figure 27: Closed issues in project repository

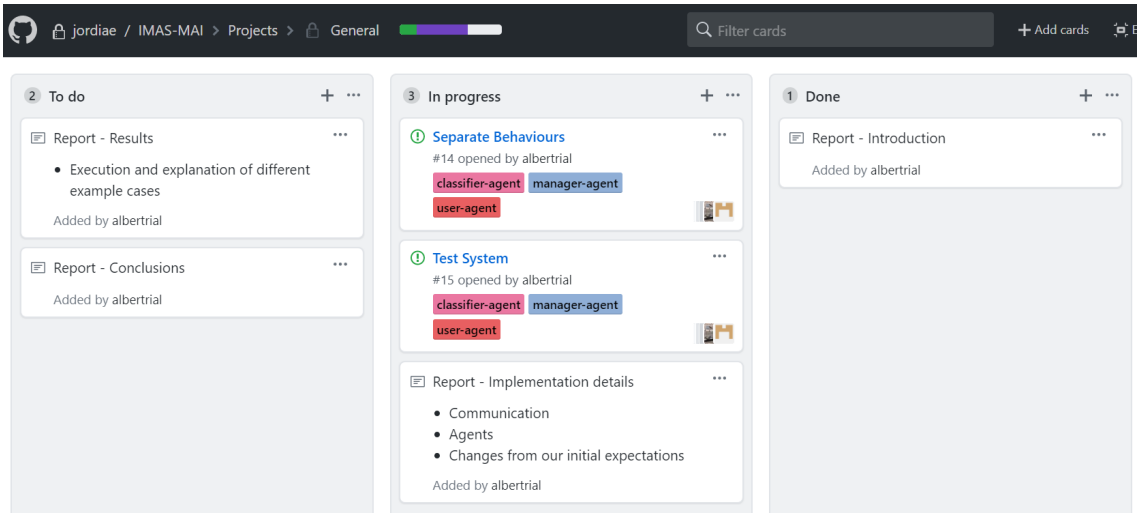

Figure 28: General sub-project status

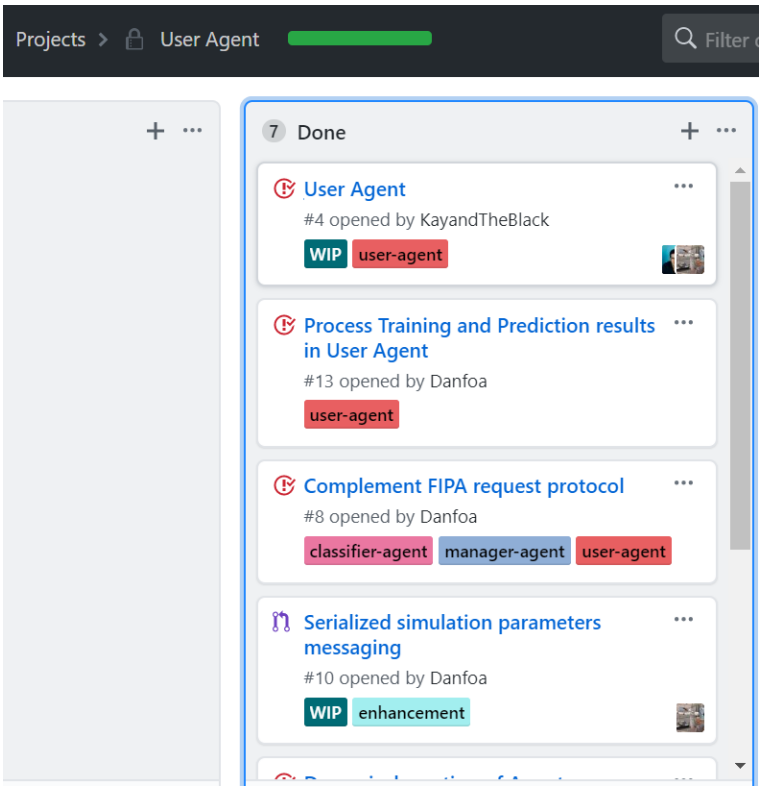

Figure 29: User agent sub-project status

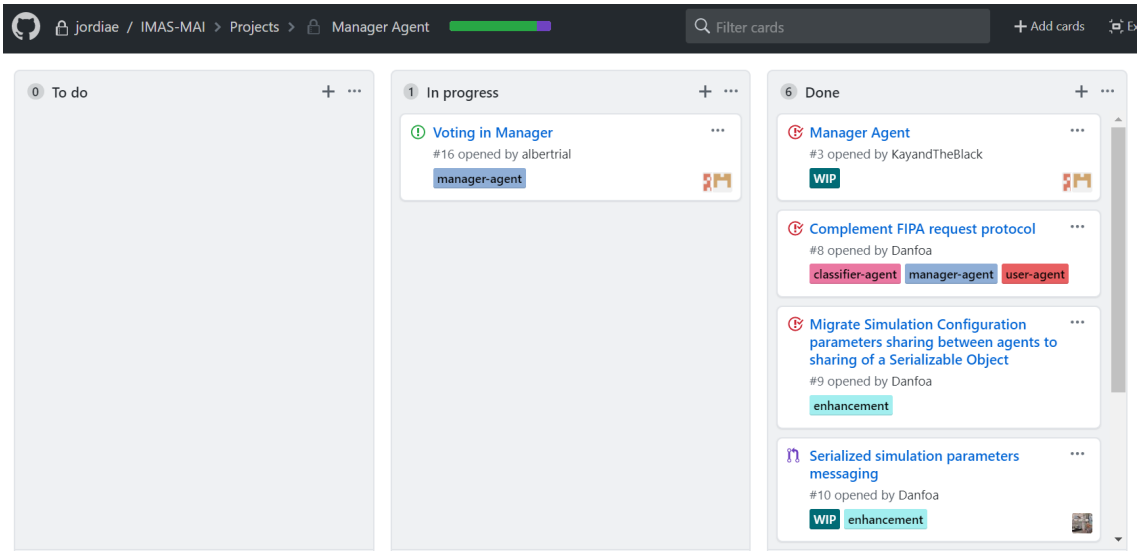

Figure 30: Manager agent sub-project status

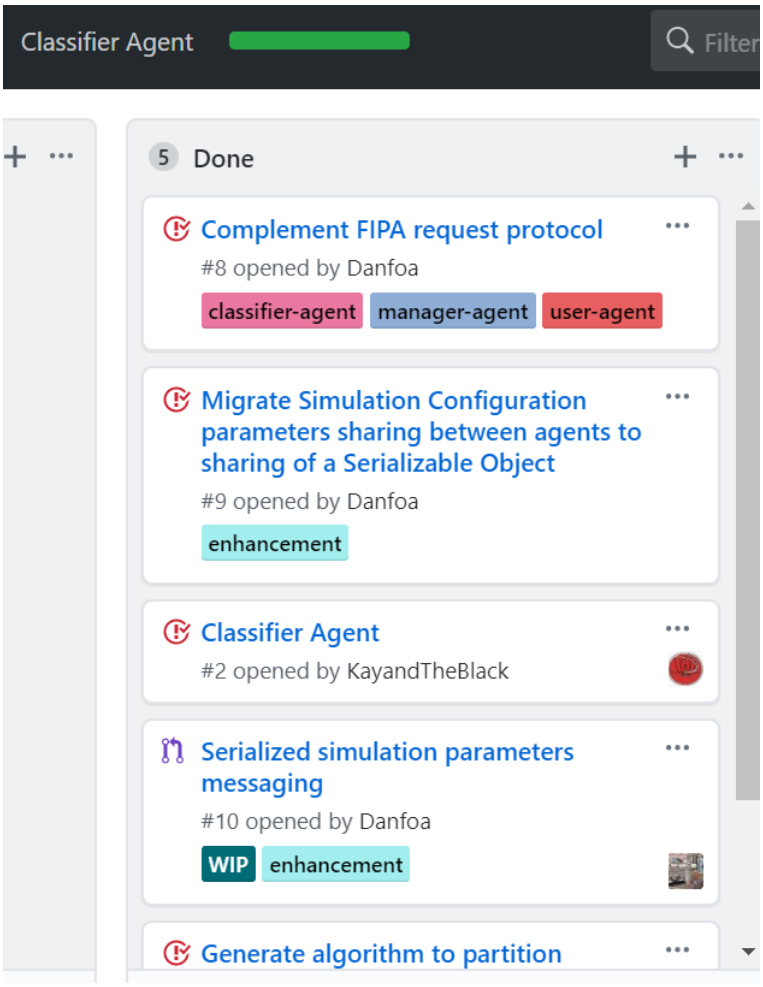

Figure 31: Classifier agent sub-project status

B.2 Project meetings

In order to track our meetings and write the topics discussed and the decisions made, we use the Github Wiki (https://github.com/jordiae/IMAS-MAI/wiki).

In this section we describe the information about all the meetings done since the start of the project:

B.2.1 Meeting 1 (Activity 1)

- Date: 09/10/19
- Duration: 2 hour
- Assistants: All team members
- Topics:
  - Statement comments
  - First delivery decisions

- Decisions:
  - Making a wiki page for the initial delivery.
  - Splitting the work into subtasks, working remotely and checking on each other's work.

B.2.2 Meeting 2 (First implementation)

- Date: 01/11/19
- Duration: 1 hour
- Assistants: All team members
- Topics:
  - Implementation general ideas
  - Communication framework draft
  - Work splitting
- Decisions:
  - Making a wiki page in our Github for the communication protocols of the agents, to be expanded as we require more.
  - Splitting the work into subtasks: mostly each type of agent. We will pool them and distribute them as required.

B.2.3 Meeting 3 (First implementation)

- Date: 08/11/19
- Duration: 3 hours
- Assistants: All team members
- Summary: After individually studying the task assigned to each of us, we shared knowledge and went deep on our discussions.
- Topics:
  - Communication protocol between Agents
  - Discussion about the dynamical creation of Agents
  - Discussion about the proper way of reading and sending the configuration file properties
- Decisions:
  - For the communication protocol we decided to use the FIPA Request Interaction Protocol.
  - For reading and sending the configuration file, we will use an XML Parser and construct manually a string with the parameters, divided by some kind of character like "@". We decided to study on the future a way to send a Serializable object instead of doing it manually, but it is not our priority.
  - We left also the dynamical creation of Agents for later. First, we will instantiate them via command line.

B.2.4 Meeting 4 (First implementation)

- Date: 15/11/19
- Duration: 2 hours
- Assistants: All team members
- Summary: We reviewed the code that we had and commented our thoughts about what we should improve.
- Topics:
  - Classifier Agents future work
  - Discussion about the dynamical creation of Agents
  - Discussion about what was left for the first implementation
- Decisions:
  - We decided that the Classifier Agent that we had was good enough for this first implementation.
  - The dynamical creation of Classifiers and Manager should be done for the first implementation. User should instantiate the Manager and this one the Classifiers.
  - We decided that the Manager should send the training request to Classifiers. However, we decided that the prediction was not going to be part of this first approach.

B.2.5 Meeting 5 (Activity 2)

- Date: 28/11/19
- Duration: 2 hours
- Assistants: All team members
- Summary: We reviewed the statement of the activity and we split the tasks.
- Topics:
  - Tasks to be done
  - Discussion about the dynamical creation of Agents
  - Discussion about our first comments and thoughts about the different coordination mechanisms
- Decisions:
  - Splitting the work into subtasks, working remotely and checking on each other's work.
  - Making a Google Docs in order to keep all the team updated with the last version.

B.2.6 Meeting 6 (Activity 2)

- Date: 02/12/19
- Duration: 1.5 hours
- Assistants: All team members
- Summary: We reviewed the status of each task and decided which coordination mechanisms are appropriate for our project.
- Topics:
  - Revision of the analysis, advantages and disadvantages of each coordination mechanisms.
  - Discussion about the appropriate coordination mechanisms.
- Decisions:
  - We decided that the best ones we could implement are voting and organisational structures (hierarchical).
  - We decided how we would do the presentation of the activity.

B.2.7 Meeting 7 (Final delivery)

- Date: 14/12/19
- Duration: 2 hours
- Assistants: All team members
- Summary: We reviewed what was left before the final delivery of the project.
- Topics:
  - Revision of the implementation and the refactoring needed in order to satisfy all criteria of the rubric.
  - Discussion on how to prepare the presentation.
- Decisions:
  - We split the work into subtasks in order to start working on the implementation and on the final report and presentation.
  - We discussed on how to implement some details of the implementation (how to separate behaviours, voting mechanism, ...)

B.2.8 Meeting 8 (Final delivery)

- Date: 04/01/20
- Duration: 6 hours
- Assistants: All team members
- Summary: Revision of the entire project and preparation of test cases.
- Topics:

  - We reviewed the work done and we prepared different executions to test our system.

- Decisions:
  - We reviewed all the code and we made some refactoring.
  - We tested different executions and special cases to find bugs and solve them.
  - We prepared the example cases to include them in the report and presentation.

B.2.9 Meeting 9 (Final delivery)

- Date: 11/01/20
- Duration: 5 hours
- Assistants: All team members
- Summary: Meeting to review the report and presentation
- Topics:
  - Revision of the document with the details of the implementation, results and conclusions.
- Decisions:
  - We assigned to each team member a part of the report that was left.
  - We decided that for the presentation, apart from the slides, we would do a video to show the execution of special cases.

References

[1] Russell, S. J., & Norvig, P. (2016). Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited,.

[2] Wikibooks contributors. (2019, 11 January). Artificial Intelligence: Agents and their Environments. In Wikibooks. Retrieved 22:35, October 16, 2019, from `https://en.wikibooks.org/wiki/Artificial_Intelligence/AI_Agents_and_their_Environments`

[3] Nowrouzi, Ehsan (2012). Artificial Intelligence Chapter Two: Agents Slides. In Slideshare. Retrieved 16:54, October 16, 2019, from `https://es.slideshare.net/EhsanNowrouzi/artificial-intelligence-chapter-two-agents`

[4] Lang, S. (2014). Web Development with Jadeh. Packt Publishing.

[5] Darwin, I. F. (2014). Java Cookbook: Solutions and Examples for Java Developers. O'Reilly Media.