AGENT-BASED DECISION SUPPORT SYSTEM (A-DSS)

MULTI AGENT SYSTEMS

Aïda Valls Mateu and Jordi Pascual Fontanilles

Master in Artificial Intelligence

UNIVERSITAT ROVIRA I VIRGILI

Tarragona

2021

Laia Borrell Araunabeña

Sergi Cirera Rocosa

Mario Lozano Cortés

Iago Águila Cifuentes

Contents

1. INTRODUCTION

Multiagent systems (MAS) are systems composed by two or more intelligent agents which interact between them [1]. MAS have proved to be extremely useful for solving distributed problems (i.e., problems in which data, actions, planning or control cannot be processed by a single entity). In fact, the idea of an agent applies perfectly in these situations in which no part of the system has access to all information.

Compared to a complex individual agent, MAS can offer flexibility (the capacity to dynamically increase or decrease the number of agents depending on the needs), reconfigurability (the ability to change the interactions between agents to adjust to a given need), robustness (the capacity to achieve goals despite the failure of some parts of the system) and cost-effectiveness (the ability to consume fewer resources to achieve a goal) [2,3]. MAS are generally designed for physical tasks such as rescue operations in hazardous environments [4], autonomous shepherding [5], space exploration [6,7] or object transportation [8]. MAS, however, can also be used for classification tasks with different types of data (image, text, tabular) or with missing data [9].

A specific use of MAS is decision support systems (DSS), which are programs used to support determinations, judgements and courses of action in an organization or business. In the last years, a large increase in financial frauds followed by subsequent business failures have led to concerns in the veracity of corporation finance statements. Despite new audit standards, it is not uncommon for audits to fail in the challenging task of determining whether a given company is fraudulent or not [10]. Furthermore, auditing is a slow and tedious process, which hinders systematic auditing [11]. Machine learning models are gaining relevance in fraud-detection tasks, as they may outperform humans both in terms of accuracy and time.

Thus, the goal of this project is to create an agent-based decision support system (A-DSS) for detecting fraudulent firms. It aims to properly coordinate a set of agents to form an application where an organization could enter a set of data about a given firm and the system returned an accurate classification stating whether the firm has risk of fraudulence or not.

2. METHODOLOGY

The implementation of the multi-agent system has been done with the JAVA Agent Development Framework (JADE) [12], which is a software for implementing MAS. This software allows for a simplification of MAS implementation through a middleware that complies with the Foundation for Intelligent Physical Agents (FIPA) specifications [13] together with a set of graphical tools that support the debugging and deployment phases.

Moreover, since the main goal of the project includes a machine learning classification task, the need of an appropriate classifier arises. In this case, J48 decision trees based on entropy gain [14] have been used, which have been implemented using the Waikato Environment for Knowledge Analysis (Weka) software [15].

As any machine learning classifier, J48 decision trees must be trained before predicting the output of any new instance. The dataset used in this task has been The Audit Dataset (full name Audit Risk Dataset for classifying Fraudulent Firms), which contains data of 767 firms from different sectors from India [16]. The goal of the dataset is to help auditors to predict a fraudulent firm on the basis of present and historical risk factors. It contains 24 attributes which include company sector, location and several risk factors. The dataset also specifies the class of each instance, whose value is 1 for fraudulent companies and 0 for non-fraudulent companies. Finally, it is worth mentioning that the dataset is slightly imbalanced, with 296 fraudulent companies and 470 non-fraudulent ones [17].

From the original 767 instances of the dataset, a set of 50 has been used for testing and the rest (717 instances) have been used for training. Each of the generated classifiers has been given a random subset of 300 training instances, from which 225 will be used for training and 75 for validation. Each classifier only considers a subset of 6 attributes with which it is trained, to simulate the data that a concrete auditing company is able to gather. Finally, once with the classifiers trained, the testing of the A-DSS is made every time the enterprise using the system enters a new instance. This has been simulated by creating 3 subsets of 15 instances from the 50 separated at the beginning for testing. Each of these 15 testing instances only contains 20 of the 25 attributes, and only the classifiers that receive values for its 6 attributes will be allowed to participate in the decision system.

2.1. ARCHITECTURE OF THE SYSTEM

In this section, the architecture of the designed agent-based decision system is defined. As seen in figure 1, the selected architecture for the proposed problems is based on three different types of agents organized hierarchically. Hence, the architecture is designed and implemented as a multi-agent system. Notice that it consists of a hybrid architecture, since not all the agents are from the same type (deliberative or reactive).

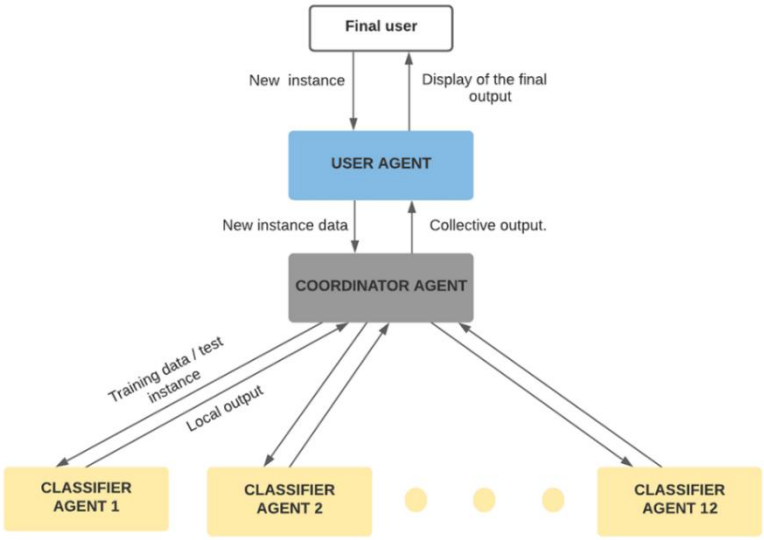

Figure 1: Architecture of the Multi-Agent system.

- **Classifier agents**: They are responsible for classifying as fraudulent or legal a new observation or enterprise after being trained with a reduced set of instances. There are 12 of them, and each shall use 300 instances and 6 attributes in their training. In order to make predictions for a test instance, which must not contain any missing value for the 6 attributes the classifier has been trained with. Their architecture is deliberative as it contains simple goals and makes use of previously known concrete attributes.

  Agent type is considered to be a wrapper, since it contains code belonging to weka in order to perform training and classification. It has been added in a way it can it can communicate with the agents of the system. The properties that the classifier agents exhibit in our MAS are the following:

  - Flexibility: A flexible agent is one that can adapt to the environment. Thus, since the classifier agents will adapt the decision tree's parameters while training and the validation set, they can be said to be flexible. The learning property makes an agent more flexible.
  - Social ability: It cooperates and communicates directly to the coordinator agent in order to receive the correspondent instance and send the correspondent classification result.
  - Rationality: The agents do not have irrational behaviours and neither show irrational results.
  - Learning: An agent is considered to have the learning property if it is able to improve its performance in an automatic way. Since the classification agents will perform a machine learning classification task, the agents have the learning property.
  - Autonomy: The agents know when and what tasks to perform in order to achieve the goal without need of constant indications by the user. Thus, they are said to be autonomous.
  - Temporal continuity: The agents will be running all the time. When they finish their tasks, they will be waiting for the next request of the coordinator agent.

- **Coordinator agent**: It has several roles, among which we find serving as a gateway between the user agent and the classification agents. The coordinator agent's roles include:

  - Splitting the training data to each of the classifiers. It must send 300 instances with 6 attributes to each of the classifier agents for training and save 25% of this train data for validation.
  - Splitting the testing instances to each of the corresponding classifiers. Based on the knowledge the coordinator agent has on the attributes that each classifier has been trained with, the coordinator will send each test instance only to the classifiers that can classify it.
  - Serve as a decision system, gathering the different classification outputs from the twelve classifiers and emitting a single answer to the data introduced.

  It is an agent with a deliberative architecture as it focuses on a set of basic goals (data splitting and decision system. Agent type is collaborative since it cooperates with the rest of the agents by performing

different tasks like the voting system. The communication with the rest of the agents is essential. The properties that the coordinator agent exhibits in our MAS are the following:

  - Social ability: It cooperates with the user agent and classifier agents in order to present the results of the classifications.
  - Rationality: It acts in order to achieve the general goals of the system and not in order to prevent it.
  - Autonomy: Has the ability to perform the required tasks by himself, for example, splitting the instances in order to send them to the classifiers.
  - Temporal continuity: The agent will be running all the time. When it is done with its tasks, it will be waiting for the next message of the user.

- **User agent**: Its role is to provide an interface for the user to interact with the system. For that reason, it provides a way of selecting or introducing as input the attributes related to a firm being fraudulent. It has a reactive architecture as it must react quickly to interactions between the user and the interface provided. In this case, agent type is interface since it asks to the user and reacts to its input. It also provides to the user the final results of the classification. The properties that the user agent exhibits in our MAS are the following:

  - Reactivity: An agent is reactive when it maintains an ongoing interaction with its environment. Thus, the user agent is reactive, since it interacts with the user by receiving its orders. Consequently, none of the other two types of agents are reactive, since they do not interact with the environment.
  - Social ability: It cooperates with the coordinator agents in order to achieve their goals. They interact between them via an agent-communication language in order to work together.
  - Rationality: The agent will act in order to achieve its goals.
  - Autonomy: Has the ability to pursue goals in an autonomous way, it does not interact continuously with the user, only when the user enters the input.
  - Temporal Continuity: The agent will be running all the time. It will be in standby since it must wait for the user input.

- **Other agents**: Besides the user, coordinator and classifier agents, there will be two more agents in the Main Container, which are automatically created by the JADE library [18]:

  - AMS (Agent Management System). The AMS is responsible for the creation/termination of the agents, and for providing a naming service (i.e., ensuring that each agent has a unique name).
  - DF (Directory Facilitator). The DF has information of all the agents in the system. Agents that require services can then use the DF to search for third agents that can provide said services.

Each of the afore-mentioned properties of each agent are summarized in the following table (Table 1).

	AGENTS		
PROPERTIES	*USER*	*COORDINATOR*	*CLASSIFIER*
Flexibility	*No*	*No*	***Yes***
Reactivity	***Yes***	*No*	*No*
Proactiveness	*No*	*No*	*No*
Social ability	***Yes***	***Yes***	***Yes***
Rationality	***Yes***	***Yes***	***Yes***
Reasoning	*No*	*No*	*No*
Learning	*No*	*No*	***Yes***
Autonomy	***Yes***	***Yes***	***Yes***
Temporal continuity	***Yes***	***Yes***	***Yes***
Mobility	*No*	*No*	*No*

Table 1: summarization of the properties of the agents in the ADSS

2.2. GLOBAL MAS FUNCTIONALITY

In this section, the overall functionality and implementation of the ADSS is detailed. The functionality of the MAS is divided in two steps: the initialization of the system and its pipeline when being used.

Initialization

During the initialization step all 14 agents (user, coordinator and 12 classifiers) are created. The user and the coordinator are first created by being manually introduced in the POM file. In each of their class files we have created a register method that adds each agent to the Directory Facilitator (DF) and their corresponding behaviours.

Regarding the coordinator agent, besides from its registration to the DF, it also reads directly the file with the training instances and randomly splits 300 of them to each of the 12 classifiers, each of them receiving instances containing 6 different attributes. However, as mentioned, these classifiers are not created in the POM file as the other two types of agents but are created by the coordinator in its same container after being registered. The way these instances arrive to the classifiers is described in the communication section.

When these classifier agents are initialized, they execute the methods described in their class file, which consist in a registration to the DF and their training with the received instances from the coordinator. The training step

consists in selecting a random subset of 225 instances for training and 75 for the evaluation of its classification. The training instances are used by each agent to train a J48 tree classifier and the evaluation ones to assess its performance, which is stored in an agent attribute for future usage.

Usage

When the last classifier agent is trained, the MAS enters the usage step. During this step, the MAS waits for instances to classify, and when they are received, it classifies them. The pipeline does a loop, since the process starts in the user agent, passes through the coordinator, arrives to the classifiers, then goes back to the coordinator and finally ends up again in the user.

In this stage, the user agent asks/allows the human user of the system to enter a path corresponding to a file containing the firms to classify. Whenever it receives a path, the user reads the dataset and it sends the input instances to the coordinator, who will receive those instances and iterate over each of them checking which attributes they contain, and which are missing. Then, it checks which classifiers have been trained with six attributes that are present in the instance available information and are sent the corresponding instance by the coordinator. Notice that just the six attributes that the classifiers have been trained with are maintained in that given instance, the others are filtered out. No other instance is sent until the classifiers have finished classifying the previous one.

When an instance arrives to the corresponding classifiers, they used their trained J48 model to classify the new firm and sent their estimation back to the coordinator.

Followingly, the coordinator agent waits for the active classifiers' responses and when it has gathered them all, it performs a weighted mean with each classifier vote/result (which will be 0 if no risk or 1 if risk exists), where the weights of the sum are the validation accuracies of each classifier stored during the initialization step (equation 1). If the result is equal or greater than 0.5, the firm is classified as fraudulent, and if the result is lower than 0.5, the firm is classified as non-fraudulent. This voting system is highly related to the plurality system, since each agent has just one vote to give to one of the two possible alternatives: risk or not risk. However, in this case, not all the agent votes are equally considered, which breaks the anonymity property of the social choice rule. Nonetheless, it is done for the best, since the MAS gives more relevance to the votes of those agents with the better performances. Lastly, once with the voting performed for a given instance, the result is sent to the user agent and the next instance is processed and sent again to the corresponding classifiers.

$$Result = \frac{\sum_{i=1}^{I} v_i * w_i}{\sum_{i=1}^{I} w_i}$$

Equation 1. Voting system of the MAS. Let v_i be the vote emitted by classifier i with values in {0,1}, w_i the weight of the vote of said classifier with values in [0,1] and I the number of classifier agents (in this case I = 12). The final result is the weighted average of all the votes.

Finally, when the user keeps receiving the results, it iteratively stores the final result of each instance in an array, as well as each instance true label so that the accuracy of the system can be tested for each set of 15 testing instances.

2.3. COMMUNICATION PROTOCOLS

The agents in the MAS communicate via the Agent Communication Language (ACL) [19]. The ACL is a standard language for agent communications that can contain several parameters. The only parameter that is mandatory in all messages is the performative, even though it is expected that all messages contain also at least a sender, a receiver and a content. The performative parameter defines the goal of the message (what the sender expects the receiver to do/know/reply). The sender parameter contains the agent identifier (AID) of the sending agent and the receiver parameter contains the AID of the receiving agent. Finally, the content parameter contains the information

of the message itself, which can be any object (such as an Instances object sent from the user agent to the coordinator agent or an array of doubles containing the results of the coming from the coordinator to the user agent).

All the communications within our MAS are set with the same parameters. All of the messages are point to point (from a single sender to a single receiver, without the need of a directory facilitator) and with direct routing (the message has no attenuation in strength). All messages have 'inform' as their performatives, since all messages are only used to transmit information. In most MAS, other performatives would be required to properly function, like for instance request, query-if, query-ref, call for proposals, etc. In this case, some of these other performatives could have been added, such as a call for proposals for finding which classifier agents have been trained with X attributes that allow them to classify a given instance. But, in order to reduce the complexity of the system, it was decided to use only the 'inform' performative as it is the only indispensable. For instance, the coordinator agent already knows the attributes the classifiers are trained with, and thus a call for proposals is not necessary.

2.4. COOPERATION AND COORDINATION

The MAS is coordinated by the coordinator agent, since it is in charge of the two basic coordination functionalities of the MAS. On the one hand, the coordinator agent delivers each of the instances to each of the classifiers that can classify it. On the other hand, it gathers all the votes from the classifiers and computes the final answer from the whole MAS.

The cooperation mechanism in the MAS is also based in the coordinator agent. Classifiers do not communicate between them, so they cannot originate explicit cooperative mechanisms by themselves, or interact with the environment, so they cannot originate implicit cooperative mechanisms by themselves. Given the nature of the classifiers, it is not even possible that they show emergent cooperative behaviour based on self-interested agents. Thus, a coordinator agent is required to be the responsible (via communication with all the classifier agents) to aggregate the functionality of all classifiers and orchestrate the cooperation (Figure 2).

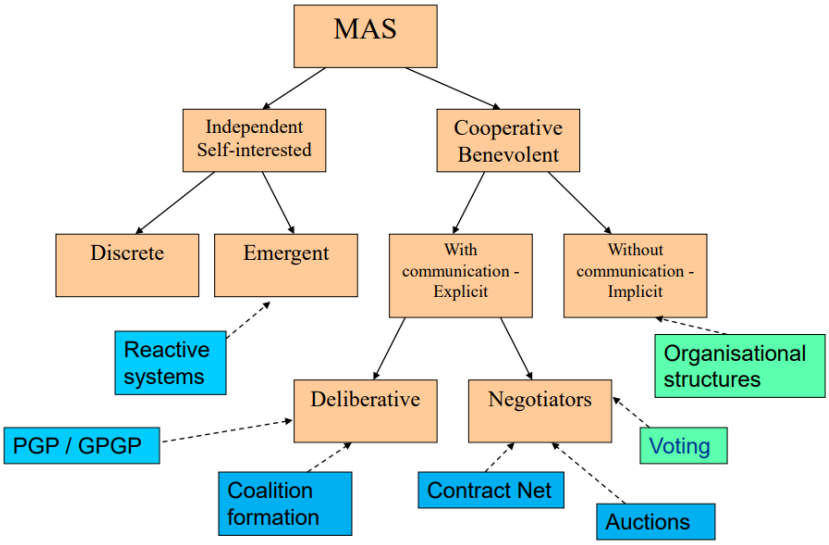

Figure 2. Hierarchy of MAS cooperation mechanisms [20]

3. CODE STRUCTURE

The code implemented through JAVA Agent Development Framework is mainly structured in three files corresponding to the different agent classes (*userAgent*, *coordAgent* and *classifierAgent*). For each of the agent classes, a behaviour file is defined to implement its operation.

All of the class files, have the same main structure. First of all, our agent class is defined as an extension of the Agent class provided by the Jade library. In order to initialize the agent, we implement two main methods: *setup()* (called once when initializing the agent) and *takeDown()*. The *setup()* method is made up of a total of two tasks, which one consist of the DF registration and other consists of starting the behaviour corresponding to the agent. In this case, the registration task has been implemented by calling a private method created in the agent class. On the other hand, the *takeDown()* method has the only function of unregister the agent from the DF.

The basic structure of the definition of each of the classes is the same for each of the agents present in the MAS, the main difference between the classes lies in the call of the methods inside the *setup()* function. The difference consists in the *trainClassifier()* method executed by the classifiers and the *sendTrainingInstances()* from the coordinator agent. This training method of the classifiers is a private method implemented in the agent class itself, and the same happens with the one from the coordinator. They are called in the setup method, right after calling the registration method. In this way, these behaviours are forced to be the first to be executed (only once) when the agent is initialized. Right after calling these methods, both the classifier agent and the coordinator agent call the only behaviour they have, extended from the Behaviours class in Jade. The only agent without an initializing method is the user agent, which implies that in its *setup()* method we limited ourselves to add its behaviour.

As mentioned before, each one of the agents has a corresponding file for its behaviour, which is added in the registration method. The names of the files are the following ones: *userBehaviour*, *coordinatorBehaviour* and *classifierBehaviour*. In the following, the operation of the behaviours (including the training behaviours defined as methods) is detailed from the point of view of the code implementation:

Coordinator initializing method: *sendTrainingInstances()*

As previously mentioned, this behaviour is implemented as a private method in the coordinator agent class when. It is executed after the registration method when the system is initialized. The main code steps are the following:

1. Reading the 'audit.arff' dataset containing the training instances.
2. Definition of the combinatorial list of attributes for each of the classifier agents.
3. Start of a 'for' loop that iterates over each combination of attributes (12 in total). Using the 'numclassifiers' property of the *properites.xml* file, the number of attribute combinations to iterate over (**N**) is selected. We've decided that **N** can only take the values 8, 9, 10, 11 or 12.
    a. Creation of **N** sub datasets with the correspondent combination of attributes and 300 random instances to send to each classifier.
    b. Creation of the **N** classifier agents in the main container.
    c. Send an inform ACL message to each created classifier with the corresponding sub dataset.
4. Sets the global state of the coordinator agent IDLE.

Note that an attribute representing the state of the coordinator agent is defined. This agent can assume two types of states, IDLE or VOTING, throughout the process, which will help to determine the appropriate moments to execute certain tasks.

Classifier's initializing method: *trainClassifier()*

The classifier training behaviour is implemented as a private method in its correspondent agent class. It is executed when each of the **N** new created classifier agents are initialized, just after their registration. The main steps of this method are the following:

1. Wait for the message of the coordinator agent using a *BlockingReceive* instruction, which means that it does not execute until it receives the training instances sent.

2. Extract the content of the message and divide the dataset into to other datasets, one for training (75%, 225 instances) and the other for validating (25%, 75 instances).
3. Train the classifier with the training dataset.
4. Evaluate the trained classifier with the validation dataset.
5. The trained model and the performance are set as attributes of the agent class, so that they can be accessed at any time.

Once the training of the **N** classifiers of the system is completed, the behaviours related to the classification of new instances (*userBehaviour*, *coordinatorBehaviour*, *classifierBehaviour*) are started. The start of the whole procedure takes place in the *userBehaviour*.

System testing / usage by the human user of the application

The **user behaviour** is defined with a class extension of the Jade's *CyclicBehaviour*, which means that once the behaviour is initialized, it will be continuously executed, cyclically. It is formed by two different methods, one of which is the trigger of the whole procedure, and the other is the one that determines the end of the whole process by collecting and showing the results, called *action()* and *collect_results*(), respectively. Both methods will be executed cyclically, but the one that will start will be *action()*, since the remaining one is blocked by a condition that will be discussed later. The main code steps for *action()* are as follows:

1. If the state of the coordinator agent is IDLE (set when finished training) then we enter the code, the agent requests the human to enter a path in the terminal with the input dataset containing the instances to classify.
2. The human user (us) writes the input dataset in the terminal.
3. The dataset is read and an inform ACL message is sent to the coordinator agent with the dataset.
4. Call the *collect_result*() method with an argument that indicates the number of instances classified.
5. The argument is initialized to 0.

Note that it will be another agent responsible for updating the argument. Until this argument has the total value of the instances to classify, the final *collect_result()* method will not be executed.

When the user agent sends a message to the coordinator, its corresponding behaviour (***coordinatorBehaviour***) is executed, since it has a *blockingReceive* function that pauses it until a message with the instances to test is received from the user. This behaviour, as well as that of the user agent, is also defined as cyclic. In this case, it is formed by two methods called *action*() and *voting*(). The first one executed in the code flow is the *action*() method which, as mentioned, is blocked until it receives a message from the user agent. The *voting*() method is also blocked by a condition to prevent the cyclic behaviour from executing it at the wrong time. The main code steps for *action()* are the following:

1. Waits for a message with the instruction Blocking Receive.
2. Set the state of the coordinator to VOTING, meaning that it is busy with a set of instances to classify.
3. Once it receives a message, the code will be executed only if the sender is the userAgent, since it will also be receiving messages from the classifiers, as we will see after.
4. Gets the content of the message (input dataset).
5. Definition of the combinatorial list of attributes for each of the classifier agents (same combinations as in the training).
6. Starts a 'for' loop for each one of the iterations of the input dataset (15 in total):
    a. Extract the corresponding instance of the dataset.
    b. Obtains all the names of the attributes that the instance contains.
    c. Starts a 'for' loop for each possible combination of attributes (each classifier):
        i. Filters the instance in order to have only the parameters of the corresponding combination (classifier).
        ii. If all the attributes of the combination exist in the original attributes of the instance, then sends an inform ACL message to the correspondent classifier with that instance and the corresponding 6 attributes.
    d. Call the method *voting()* with an argument that indicates the true labels of the instances. This method is explained later.

7. Sets the state of the coordinator to IDLE.

In this code, it is important to consider the condition in which the message will be sent to a classifier. The instance extracted from the input dataset contains 20 of the 25 original attributes, so it is necessary to check by means of a condition that all the attributes of the combination corresponding to the classifier exist in the instance that is going to be classified. Also, note that when the *voting()* method ends, meaning that it has received all the classifications made by the active classifiers, the coordinator state is set to IDLE, which will allow the process to be restarted again in due time.

At this point in the system, the next step of the procedure is in the ***classifierBehaviour***. This behaviour is also defined as a cyclic behaviour. In this case, it only contains a single method called *action()*. The main steps are as follow:

1. Waits for a message with the instruction *BlockingReceive*.
2. Gets the trained model.
3. Gets the content of the message (instance to classify).
4. Computes the prediction for the instance with the instruction *classifyInstance* of Weka.
5. Gets the performance of the classification.
6. Sends the prediction and the performance to the coordinator agent via an ACL inform message.

Note that we return to the coordinator agent, but this time it will be the method *voting()* the one that is going to be executed. The main steps are as follows:

1. Stays in a loop while the responses of the classifiers are less than the number of classifiers.
    a. Waits for a message with the instruction Blocking Receive.
    b. Gets the prediction and performance of the classifier and saves them into two different arrays.
    c. Sum 1 to the counter of responses.
    d. When all responses have been received, exits the loop.
2. Computes the vector of weights by dividing each performance with the sum of performances.
3. Computes the result by adding the product of each classification and its corresponding weight.
4. If the result is bigger than 0.5, the instance will be classified as 1. Otherwise, it will be classified as 0.
5. The final result and the true label are sent with an ACL inform message to the user agent.

Note that the voted is applied each time the classifications are obtained for each instance, which implies having collected the predictions of all those classifiers that could classify it. Consequently, a vote will be taken for each instance, i.e. a total of 15 votes (first for loop of the coordinator's *action*() method).

Finally, when the message is sent to the user agent, it is returned to its behavior, which, as mentioned above, has a method called *collect_results*(). The main steps are as follows:

1. If the argument (instances to test) is different from 0 then you enter the code.
2. Stays in a loop while the received instances are lower than the argument 'instances to test'.
    a. Waits for a message with the instruction Blocking Receive.
    b. Saves the predictions and the real labels from the instances into two new arrays.
3. If the number of received instances is equal to the instances to test then print all results.

With this last method, the flow of the entire multi-agent system ends. Once finished, the user agent asks again for the input dataset in order to repeat the process again for the next input. Remember that the coordinator agent has been returned to IDLE status, so it is ready to receive the next dataset.

4. RESULTS

This section details the global tests performed on the system in order to get a general idea of how the system responds to new instances not contained in the classifier training set. Therefore, the creation of 3 test files with 15 test instances each is envisaged. In order to be able to test the system in conditions as close to reality as possible, 5 attributes have been removed for each of the instances to represent missing values. These attributes have been randomly selected and are different for each test instance. Since this is a classification problem, accuracy (percentage of correctly classified instances), sensitivity (percentage of correctly classified positive instances) and specificity (percentage of correctly classified negative instances) will be used as metrics. The results obtained under the detailed methodology are as follows:

Test File	**Accuracy**	**Sensitivity**	**Specificity**
0input_user.arff	93.33%	85.71%	100%
1input_user.arff	100%	100%	100%
2input_user.arff	93.33%	85.71%	100%

Table 2: System performance metrics based on test file

As seen in table 2, the system shows a great performance, especially when classifying negative instances. However, taking into account system considerations it would be better to increase the sensitivity rate (for instance by reducing the voting threshold) since false negative results should be more costly than false positive results. However, the system performs well under most conditions, so we provide the combined data for the overall system in table 3.

Accuracy	**Sensitivity**	**Specificity**
95.56%	90.47%	100%

Table 3: Global system performance metrics

As seen in table 3, the accuracy of the system is 95%, an optimal result obtained thanks to the robustness provided by the multi-agent system when taking into account the different nuances of each of the attributes found in our dataset. Thus, we can conclude that inter-agent communication works well and is able, through a ranking system based on giving higher decision weights to the best classifiers, to provide useful and robust results in real situations.

5. CONCLUSIONS

MAS are used for solving a myriad of different problems [3]. While generally used for solving physical tasks involving different agents, MAS can also be implemented for solving classification problems [9]. In this project, an A-DSS for detecting fraudulent firms has been implemented with the JADE library and the Audit dataset. The system consists of 14 agents (1 user agent, 1 coordinator agent and 12 classifier agents). The agents are able to execute their tasks and to interact with each other. After training, the A-DSS is able to accurately predict 95.56% of the test instances from the Audit dataset, with a sensitivity of 90.47% and a specificity of 100%.

During the development of the code, a number of issues and challenges have arisen and have been addressed as we have progressed. We consider that the main tasks are the following two: coding the functionality of each agent and coordinating all agents. Although the functionality of the agents is the basis of the system, their coordination has been a major challenge in this project. This has been a clear indicator of the importance of coordination and cooperation between agents in a multi-agent system.

Finally, some further functionalities could be implemented as future work. For instance, instead of creating the classifier agents based on a predetermined partition of attributes, the partition could be randomly created. Furthermore, the functionality of creating and terminating classifier agents during the usage phase could also be implemented. This would give the whole project a more agent-oriented style, even though these functionalities are not required for the system to properly classify the instances.

6. BILBIOGRAPHY

1. Niazi, M., & Hussain, A. (2011). Agent-based computing from multi-agent systems to agent-based models: a visual survey. *Scientometrics*, *89*(2), 479-499.
2. Huang, X., Chen, Q., Meng, J., & Su, K. (2016, January). Reconfigurability in Reactive Multiagent Systems. In *IJCAI* (pp. 315-321).
3. Hu, J., Bhowmick, P., Jang, I., Arvin, F., & Lanzon, A. (2021). A Decentralized Cluster Formation Containment Framework for Multirobot Systems. *IEEE Transactions on Robotics*.
4. Spurný, V., Báča, T., Saska, M., Pěnička, R., Krajník, T., Thomas, J., ... & Kumar, V. (2019). Cooperative autonomous search, grasping, and delivering in a treasure hunt scenario by a team of unmanned aerial vehicles. *Journal of Field Robotics*, *36*(1), 125-148.
5. Hu, J., Turgut, A. E., Krajník, T., Lennox, B., & Arvin, F. (2020). Occlusion-based coordination protocol design for autonomous robotic shepherding tasks. IEEE Transactions on Cognitive and Developmental Systems.
6. Steels, L. (1990). COOPERATION BETWEEN DISTRIBUTED AGENTS THROUGH SELF-ORCAMSATION. In proceedings of the First European Workshop on Modelling Autonomous Agents in a Multi-Agent World, Cambridge, England.
7. Hu, J., Niu, H., Carrasco, J., Lennox, B., & Arvin, F. (2020). Voronoi-based multi-robot autonomous exploration in unknown environments via deep reinforcement learning. IEEE Transactions on Vehicular Technology, 69(12), 14413-14423.
8. Chen, J., Gauci, M., Li, W., Kolling, A., & Groß, R. (2015). Occlusion-based cooperative transport with a swarm of miniature mobile robots. IEEE Transactions on Robotics, 31(2), 307-321.
9. Qasem, M. H., Hudaib, A., & Obeid, N. (2019). Multiagent system for mutual collaboration classification for cancer detection. Mathematical Problems in Engineering, 2019.
10. Lou, Y. I., & Wang, M. L. (2009). Fraud risk factor of the fraud triangle assessing the likelihood of fraudulent financial reporting. Journal of Business & Economics Research (JBER), 7(2).
11. Hooda, N., Bawa, S., & Rana, P. S. (2018). Fraudulent firm classification: a case study of an external audit. Applied Artificial Intelligence, 32(1), 48-64.
12. https://jade.tilab.com/
13. http://www.fipa.org/
14. Quinlan, J. R. (2014). C4. 5: programs for machine learning. Elsevier.
15. https://weka.sourceforge.io/doc.dev/weka/classifiers/trees/J48.html
16. https://www.kaggle.com/sid321axn/audit-data
17. Hooda, N., Bawa, S., & Rana, P. S. (2018). Fraudulent firm classification: a case study of an external audit. Applied Artificial Intelligence, 32(1), 48-64.
18. Caire, G. (2003). JADE tutorial: JADE programming for beginners. http://jade. tilab. com/doc/JADEProgrammingTutorial-for-beginners. Pdf.
19. http://www.fipa.org/specs/fipa00061/SC00061G.html
20. Valls, A. (2020) Lecture 05 Cooperation in MAS [05_Lect5-Coord1-Fall2020]. Introduction to MultiAgent Systems

7. ANNEXES

First meeting

11/10/2021 16:00

Participants: Iago, Laia, Mario, Sergi

Agenda:

- Lucía has left
- MAS project decisions
- Work division
- Next meeting

1. Lucía has left:

Lucía has left the IMAS course and thus we are 4 people for the project. For the moment we are not searching for anyone else as we assume that everyone is in a group already.

2. MAS project decisions

We are not going to apply dimensionality reduction to the dataset, since the project definition does not allow for this processing.

The split of the datasets between training and test will be at random.

We will revise the ideas for the architecture of the systems and the agent properties after the class on Wednesday, 13th of October.

We decide that we will try to all take part both in programming and in writing the report.

3. Work division:

- Decide which preprocessing will be applied to the dataset. - Mario
- Ask Aïda or Jordi
    - How can a classifier that works with tabular data make reasonings in a knowledge base or make plans. - Sergi
    - Do all of the classification agents perform the same task? (so all of them are equal?). - Mario
    - Do we need to take in account the DF and AMS agents for the definition of the MAS architecture? - Mario
- Write:
    - The properties of our agents - Laia and Iago
    - The environment characteristics - Laia and Iago
- Update the Team minutes in the moodle - Sergi

4. Next meeting:

Next meeting will be on Friday 15th October at 17h

Second meeting

15/10/2021 16:00

Participants: Iago, Laia, Mario, Sergi

Agenda:

- MAS project decisions
- Work division
- Next meeting

1. MAS project decisions

We are not doing any preprocessing, as the dataset is already preprocessed.

Report order:

- Goals of the system
- Architecture
- Agent's role
- Agent's properties (small paragraph and table)

2. Work division:

We decide to work during the meeting so that we can finish it now.

The report is done. We will finish the power point on Sunday 17th and then upload the delivery.

3. Next meeting:

We will choose next meeting in the future.

Third meeting

04/11/2021 15:00

Participants: Iago, Laia, Mario, Sergi

Agenda:

- First look into the implementation
- Work division
- Next meeting

1. First look into the implementation

For the members of the group, the implementation of a multi-agent system in JADE is a new area. The idea of this meeting is to establish an organization on how to start implementing code using IntelliJ.

Report order:

- Understanding the basic structure of the PingAgent example.
- Propose a first idea about the structure of our code.
- Make a list of behaviours for each agent.
- Make a list of the type of communication between agents.
- Start with the code implementation.

2. Work division

We decided to start working during the meeting and to continue with the code implementation until 10/11/2021 when we will be able to consult the existing doubts. We will work on the code using GitHub and progress will be discussed among the members of the group.

Fourth meeting

14/11/2021 16:00-18:00

Participants: Iago, Laia, Mario, Sergi

Agenda:

- Common update of the divided tasks performed by each member.
- New work division.
- Set next meeting

1.Update

All of the group members reported an error with weka dependencies. This error has been fixed by setting the dependency to Weka's cloud.

Sergi implemented the training behaviour with Laia's help, and then he filtered data by columns, leaving just 6 specified attributes.

Mario and Laia implemented the user agent setUp() and takeDown() functions, as well as a behaviour in which it waits for the user to enter an input path of the test instances file, reads the data, and passes the data to the coordinator agent.

Iago implemented the coordinator agent setUp() and takeDown() functions. In which the coordinator agent responds to the User agent if the input data has been received.

2.New work division

Iago and Sergi will work on the training data division in 10 packs of 300 instances and then send each pack to a classifier with just 6 attributes from the 25. This will be the splitting behaviour of the coordinator agent.

Laia and Mario will work on another coordinator agent behaviour which will be in charge of checking which attributes the new test instances have and which classifier agents are able to classify them.

3.Set next meeting

We will meet again at the end of next week to see how we have progressed.

Fifth meeting

21/11/2021 15:00-19:00

Participants: Iago, Laia, Mario, Sergi

Agenda:

- Common update of the divided tasks performed by each member.
- New work division.
- Set next meeting

1.Update

Sergi and Iago have worked in the behaviour of the coordinator agent that is in charge of the training data split. This behaviour is able to split the data from the data source in 10 packs of 300 instances and 6 attributes. It also creates a classifier agent with the function 'createNewAgent' for each data pack. Once the agent is created, its

corresponding data pack will be sent to the classifier. The behaviour is OnShot type and is meant to be executed only for the training of the agents.

Mario and Laia have worked in the behaviour of the coordinator agent meant for checking which attributes the new test instances have and which classifier agents are able to classify them. Once it is checked, the test data will be sent to the correspondent classifier.

2.New Work

One of the tasks planned to do is create a behaviour for the reception of the split data. We have to work on how to decide when the agent classifier will receive the training data and when it will receive the test data.

Validate the current work in order to decide further tasks.

3.Set next meeting

We will meet again at the end of next week to see how we have progressed.

Sixth meeting

17/12/2021 15:00-19:45

Participants: Iago, Laia, Mario, Sergi

Agenda:

- Common update of the divided tasks performed by each member.
- New work division.
- Set next meeting

1.Update

Sergi and Iago have worked on the implementation of the states in the multi-agent system. The initial idea is to implement two different states (TRAINING and TEST) to differentiate the activities that have to be performed while training or while testing.

Mario and Laia have worked in the behaviour of the coordinator agent meant for the voting system. The aim of the behaviour is to collect the answers of all the classifiers and decide the final result.

2.New work

Execute the multi agent system in order to detect errors in the code.

Finish the implementation of the states and add new states if needed.

3.Set next meeting

We will meet again at the end of next week to see how we have progressed.

Seventh meeting

17/12/2021 16:30-19:30

Participants: Iago, Laia, Mario, Sergi

Agenda:

- Common update of the divided tasks performed by each member.
- Set next meeting

1.Update

In this reunion we have considered that the code is ready to be executed and tested in order to solve the errors. So, we have started to fix the errors that appear when running the multi agent system. For this task we've worked all together.

2.Set next meeting

This task requires more time, so we will meet again at the end of next week to continue until we solve all the problems.

Eighth meeting

28/12/2021 17:00-21:00

Participants: Iago, Laia, Mario, Sergi

Agenda:

- Continue by solving the errors that appears when executing the system
- Set next meeting

1.Update

During this meeting we have continued with the correction of errors in the code. All the members of the group made the necessary modifications to solve the errors that did not allow the system to run completely.

One of the errors solved was the creation of the classifier agents in another container. This caused them to not communicate with the coordinator agent since they did not exist in the same container. Changes have also been made in the voting() function and a clean code has been started for certain files.

2.New work

An error has been found in Weka's classifyInstances() statement, which always returns a prediction of 0.

3.Set next meeting

It is proposed that a future meeting be held to investigate this error.

Ninth meeting

03/01/2022 16:00-20:00

Participants: Iago, Laia, Mario, Sergi

Agenda:

- Solving the problem with the classifyInstance() function from Weka
- Set next meeting

1.Update

During the days between the last meeting and the current one, we searched for information about the classifyInstance() function and performed different tests. In addition, we consulted with Professor Jordi Pascual in order to make progress on this topic, since the tests were not successful.

Finally, a couple of modifications were made to the code that solved the error. The first was to send the instance from the coordinator to the classifier as an Instances type instead of sending it as an array. With this modification, the instance format did not change at any time. The second modification was to avoid generating the instance with attributes created by ourselves.

At this point, the code is working correctly and is able to perform different executions.

2.New work

It is proposed to perform a total code cleanup: add comments, remove non-functional lines of code, etc.

It is proposed to start working on the writing of the final report.

3.Set next meeting

A new meeting is scheduled for 08/01/2022 to review the pending tasks.

Tenth meeting

08/01/2022 21:00-21:30

Participants: Iago, Laia, Mario, Sergi

Agenda:

- Review status of the project.
- Splitting up the final tasks.

1.Update

During the days between the last meeting and this one, progress has been made in the drafting of the final project report. The report is practically finished with only the last details to be completed. In addition, we have begun to structure the slides of the final presentation.

The code has been cleaned up and is now in its final version.

2.Last work

The last remaining tasks have been distributed among all participants. The list is detailed below:

- Finish the content of the slides of the final presentation.

- Write the README pertaining to the code.

- Finish the details of the final report.