Agent-based Decision Support System (A-DSS)

IMAS 21/22

TEAM 1

Authors: *Sergi Albiach, Anna Garriga, Benet Manzanares and Ramon Mateo*

Index

1. Introduction

The goal of this project is to develop an agent-based decision support system. In particular, we aim to develop a system that is able to solve a classification problem in a collaborative way. For this particular case, we will use the dataset AUDIT [1], from the UCI Machine Learning Repository [2], that contains 767 instances with 25 variables defining each sample. At the end, the final goal of this collaborative classifier is to predict whether a firm is fraudulent, basing the decision on some risk factors.

In order to solve the problem we have created 10 classifiers that will make their decision using the C4.5 [3] decision tree algorithm implementation from the WEKA 3.8.5 library [4][5], called J48 [6]. Each of these classifiers is representing a different firm and consequently is able to process only 6 attributes, trying to emulate that each of them was able to collect a different subset of the variables. Moreover, they will use only 300 samples to train the classifier, where 25% of them will be used for validation. Finally, 50 instances from the original dataset will be reserved to test the complete collaborative classifier. Every test call will be made using a subset of 15 samples with 20 of the 25 attributes.

Then, in order to obtain a final classification for each test instance, we have taken into account the validation accuracies of each of the agents and performed a weighted sum. In that way, the classifiers that obtained better accuracy have been given more relevance because we trust them more.

2. Portfolio

To complete this project we have divided the tasks among the group members, and established some deadlines to complete them. In Table 1, we show how we have distributed the tasks during the course. A summary of the meetings can be found at the team minutes.

TASK	DEADLINE	RESPONSIBLE/S
Design general architecture	6/10/2021	All
Design functions and task of each agent	6/10/2021	Sergi & Ramon
Description of the agents	15/10/2021	Sergi & Ramon
Diagrams of training and testing steps	15/10/2021	Benet & Anna
Cooperation Mechanism	15/10/2021	Benet & Anna
FULL DESIGN ACTIVITY	17/10/2021	All
Agents deployment (general structure)	8/11/2021	All
Agents initial communication	8/11/2021	Sergi & Benet
UserAgent: dataset loading and splitting	27/11/2021	Sergi & Benet
CoordinatorAgent: Dataset receiving	4/1/2022	Sergi & Benet
CoordinatorAgent: Dataset splitting for classifiers	7/12/2022	Sergi & Benet
ClassifierAgent: Dataset receiving	7/12/2022	Ramon & Anna
ClassifierAgent: training	7/12/2022	Ramon & Anna
FULL TRAINING PROCESS	7/12/2022	All
UserAgent: Attribute selection and performance metrics	08/12/2021	Sergi & Benet
CoordinatorAgent: Classifiers selection and aggregation method	08/12/2021	Sergi & Benet
ClassifierAgent: Prediction	08/12/2021	Ramon & Anna
FULL EVALUATION PROCESS	07/01/2022	All
Documentation v1	08/01/2022	Ramon & Anna
Documentation v2	09/01/2022	All
Presentation	09/01/2022	All
FULL PROJECT	09/01/2022	All

Even though we faced some delays in some of the checkpoints, we have been able to complete the task with the required specifications.

3. Agents description

Our system will contain three kinds of agents: a user agent, a coordinator agent and several classifier agents; they will all extend from the same agent class (OurAgent, explained at the Code Documentation section) due to code clearance purposes. Below you can see a brief explanation of each of them.

3.1. User agent

Based on the configuration given in the ***settings.xml*** XML file (placed at the ***resources*** folder, along with the dataset .arff), it is in charge of loading the dataset, splitting the data into training and test sets, starting the training and starting the testing and computing the test performance metrics. Its communication with other agents is limited to the *CoordinatorAgent*.

Agent properties and type description

This agent is an interface agent, as it is proactive and it is responsible that the whole process is done. According to the description. The properties that this agent has are:

- Social ability: It communicates with the coordinator agent in order to train or use/evaluate the classification system.
- Rationality: All its actions will be done with a clear goal. For instance, the training process.
- Proactive: It is the agent that takes the initiative of initializing the whole process.
- Temporal continuity: It is attentive if it receives messages from the coordination.

3.2. Coordinator agent

It waits for the *UserAgent* request for performing the training or test steps. During training, it is responsible for creating the classifiers, selecting the six attributes that each of them will use, sending the requests to the classifiers and gathering the results. It is important to note that the attribute selection is done in a way that ensures that every attribute of the dataset appears at least in one of the agent’s assignments; in that way we avoid leaving some important attribute unused and reduce the probability that an instance cannot be classified by any *ClassifierAgent*. For testing, it sends the instances to those classifiers capable of processing them, gathers and aggregates the predictions and forwards them to the *UserAgent*. The aggregation is detailed at the Cooperation Mechanisms section 4.4.

Agent properties and type description

The type of this agent is Facilitator, as it is in charge of the communication. This agent is in charge of the task that allows communication between user agent and classifier agents, splitting datasets, sending data from user agent to classifier agent and distributing tasks between all classifier agents. The properties of this agent are according to the task that this agent needs to perform are:

- Social ability: It will receive queries from the user agent and answer them. Additionally, it will communicate with all the classifiers.

- Rationality: All its communications and processes will be done in order to achieve the desired goals.

- Reasoning capabilities: It will be able to aggregate the results of all the classifiers based on their results during the training.

- Autonomy: It has the control on how to split the data and it has access to all information.

- Temporal continuity: All the time is waiting if it receives new registrations, and messages from the user or classifiers.

3.3. Classifier agents

Receives its part of the training data and creates a Decision Tree based on it. After this, it receives the test cases and sends the output classification results to the coordinator agent. Therefore, this agent can receive two different messages for performing the two different tasks: The tasks that this agent needs to perform are Training, and Test for evaluating models.

Agent properties and type description

The type of this agent is Wrapper. The reason why Classifiers are wrappers is that we implement a communication protocol for executing a classifier that is implemented using another system, so we add a new functionality that allows communication to this system.

- Social ability: It receives orders from the coordinator agent and returns the corresponding outputs.

- Rationality: Only works to achieve its goals. For example, it will not perform a classification if it is not necessary.

- Learning: It will use the J48 [6] decision tree algorithm, learning from the training data.

- Reasoning capabilities: It will extrapolate the knowledge from the training data to its inferences during evaluation.
- Temporal continuity: It is always waiting for new data to train or to classify.

4. Structure design

Our proposed System Architecture consists of a hierarchy (Fig. 1) with the User Agent at the top, the Coordinator Agent just below receiving and answering User Agent requests, and Classifier Agents at the bottom receiving and answering Coordinator Agent's requests .

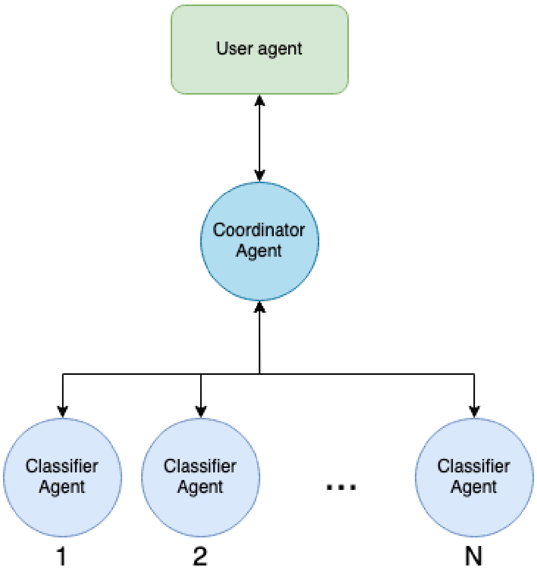


Fig. 1: General diagram of the system architecture

The structure of the system consists of three phases: initialization, training and testing. These procedures are detailed in the following subsections.

4.1. Initialization phase

For the initialization, every agent will register itself in the Directory Facilitator, defining the service type of “user”, “coordinator” or “classifier”, respectively. Note that this will happen at the start of the MAS only for the *UserAgent* and the *CoordinatorAgent*, since the *ClassifierAgent*s are created on demand depending on the configuration (registering itself when instantiated).

In addition, during this phase the *UserAgent* will read the ***settings.xml*** file from the ***resources*** folder, which defines the following:

- **dataset_filename**: Filename of the dataset to use, which is placed also in the ***resources*** folder
- **num_classifiers**: Number of classifiers to create

- **num_training_instances**: Number of instances from the dataset to be used for training
- **num_test_instances**: Number of instances from the dataset for the testing dataset.
- **num_test_subset_instances:** Number of instances from the testing dataset for creating the subset dataset to be used at testing.
- **num_test_attributes**: Number of attributes to select for the testing instances
- **num_instances_per_classifier**: Number of instances to use for training/validation at each classifier
- **num_validation_instances_per_classifier**: Number of instances of for validation at each classifier (the rest will be for training)
- **num_training_attributes_per_classifier**: Number of attributes to be used by each classifier for training

After that, the *UserAgent* will read the dataset and split it into training and test. Finally, it will use the Directory Facilitator for finding the *CoordinatorAgent* (with a blocking periodic query until it is founded) and will start the training phase. Note that we have not implemented any kind of reaction in case less agents than expected do not respond, there is no time out, we just keep waiting.

The *CoordinatorAgent* and the *ClassifierAgents*, after having registered to the DF, remain waiting for any possible request.

4.2. Training phase

The training process is divided into two steps: requests and datasets sending (Fig. 2) and ending notification (Fig. 3).

At the first step (Fig. 2), the *UserAgent* sends a training request to the *CoordinatorAgent* with the training split and settings as content. Then, the *CoordinatorAgent* creates the required *ClassifierAgents* and waits for its complete creation by searching them at the Directory Facilitator. After, a subset training/validation instances (300) with a subset of attributes (6) are assigned to each of the classifiers. Attributes are selected in a way that makes sure to allocate at least once each one of the attributes, first performing a random shuffle of the attributes list, after assigning them sequentially from this list and, for the remaining classifiers, selecting them randomly. Subsequently, a training request is sended for each of *ClassifierAgents* with the filtered training/validation instances (containing only the corresponding attributes) and the size of the validation split. When each *ClassifierAgent* receives this request, splits the data into training and validation, builds the J48 decision tree using the training split and computes the accuracy for the validation split. The *CoordinatorAgent* waits for the *ClassifierAgents* to finish.

For the second step (Fig. 3), the *ClassifierAgents* notifies the finishing of the training by returning the accuracy to the *CoordinatorAgent*. When all of them have answered, the *CoordinatorAgent* computes the average accuracy and answer them to the *UserAgent*, finishing the training phase. These accuracies are stored for weighting at the testing phase, which is started by the *UserAgent* just after training is completed.

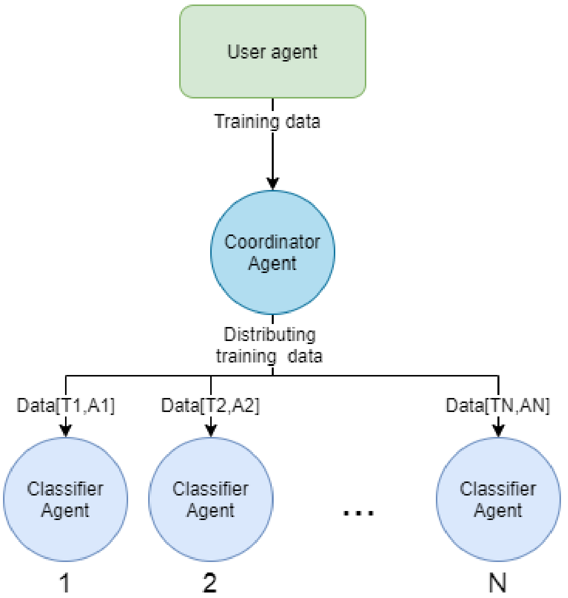


Fig. 2: Diagram of the step 1 of the training phase (requests and data sending)

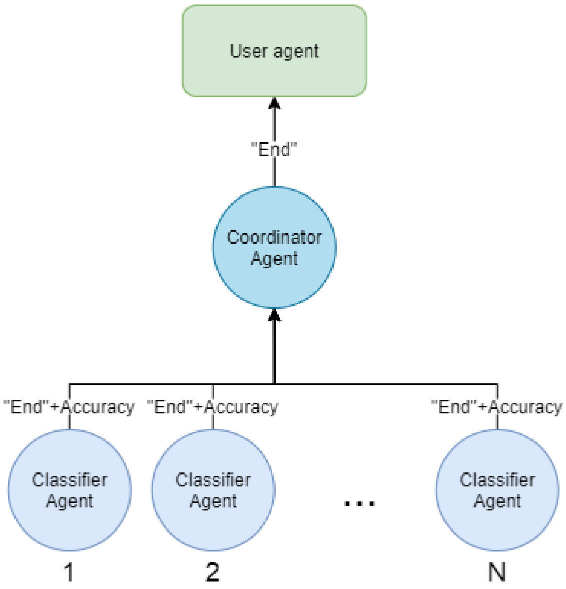


Fig. 3: Diagram of the step 2 of the training phase (ending notification)

4.3. Testing phase

The testing phase consists of two steps: data distribution and predictions computation (Fig. 4), and predictions gathering and performance metrics computing (Fig. 5).

The first step (Fig. 4) is started by the *UserAgent*, which creates a subset of the testing split with the specified size (15) and modifies each of the instances of this split to contain a different subset of random attributes (with size 20 by default). Afterwards, these testing instances are sent to the *CoordinatorAgent* as a testing request. When received, the *CoordinatorAgent* checks which instances are compatible with each classifier and send them as a testing request, waiting for all implied *ClassifierAgents* to answer. The *ClassifierAgents* use its trained decision trees for predicting the corresponding class for each instance.

At the second step (Fig. 5), the *ClassifierAgents* answer an array with all their predictions to the *CoordinatorAgent*. Subsequently, the *CoordinatorAgent* aggregates the predictions of different *ClassifierAgents* for the same instances using the cooperation mechanism specified in section 4.4. Finally, it answers the final predictions to the *UserAgent*, which computes the performance metrics (accuracy, precision, recall and F1 score) and displays them.

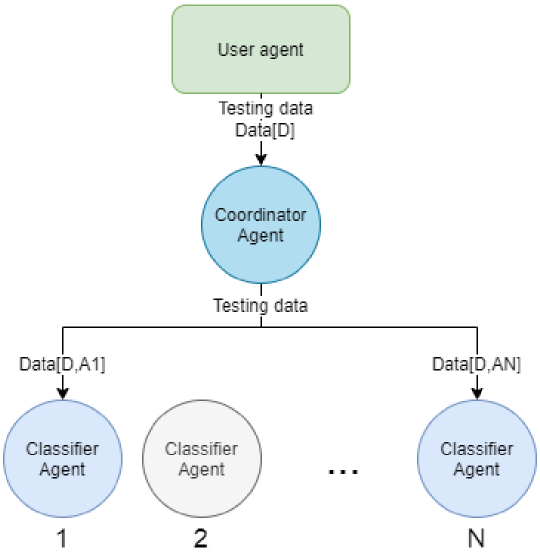


Fig. 4: Diagram of the step 1 of the testing phase (data distribution and predictions computation)

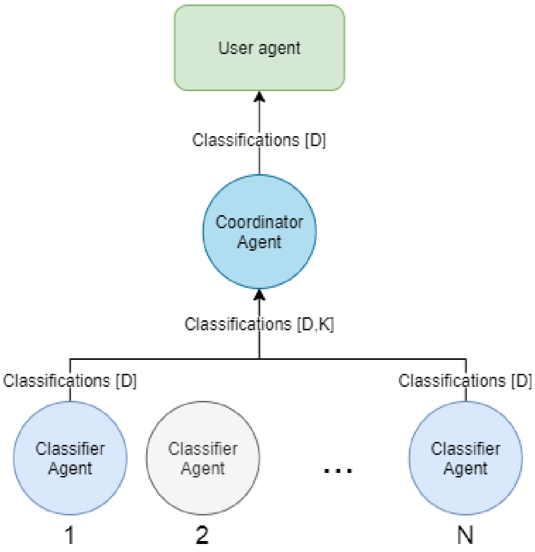


Fig. 5: Diagram of the step 2 of the testing phase (data distribution and predictions computation)

4.4. Cooperation Mechanisms

Once the *CoordinatorAgent* has received the predictions for the testing dataset, its task is to make a final prediction by taking into account the classification of each of the agents. Therefore, we need to discuss some of the cooperation mechanisms that could have been implemented in our architecture to compute the final classification output.

A first approach could be simple voting, and then the coordinator agent chooses the most frequent output among the set of results. Simple voting, however, may not be the best option, as some of the agents may have less accuracy than others and with this method we give all the agents the same weight, as well as generating a problem when there are draws.

Alternatively, we could select the output from the k agents with highest training accuracy. However, it would be a waste of computation as the other classifiers would be ignored.

A more refined approach is to use weighted voting, where each agent has an associated weight in the final output. In our case, that weight is determined by the accuracy obtained at the training step, the validation accuracy. Then, the coordinator agent sums each classifier output (0 or 1) multiplied by the corresponding normalized weight (between 0 and 1), obtaining the weighted prediction as shown in the following formula:

$$F(a_1, ..., a_n) = \sum_{j \in J} w_j b_j$$

where J is the set of agents that had done the classification, $b_j = 1$ if agent j has classified as fraud and $b_j = -1$ if not, and $w_j = \frac{accuracy_j}{\sum_{k \in J} accuracy_k}$. This gives us a number

F between -1 and 1, that the closer it is to one the bigger is the probability of fraud and -1 if not. We have to choose a parameter k (0 in our case) that will be the threshold and then, if $F > k$, we will consider that it is fraud. We could increment this parameter k so that it is more difficult to mark a firm as fraud, predicting so only when the system is very confident of that belief.

Based on the previous explanations, we finally decided to implement the weighted voting approach, as it takes into account every agent and gives to each one a reasonable weight to its output.

4.5. Communication protocols

To do all the communications we employ the following basic communication protocols:

- AchieveREInitiator: Used for performing a request to another agent. It is used when the *UserAgent* asks the coordinator to start the training or test process, or when the *CoordinatorAgent* requests a *ClassifierAgent* to perform a training or testing procedure.
- AchieveREResponder: Used for answering requests from other agents. It is used by the *CoordinatorAgent* to manage the training and testing requests from the *UserAgent* and by the *ClassifierAgent* for the training and testing requests from the *CoordinatorAgent*. In both cases, it is programmed in order to wait for the task to be finished before sending the corresponding reply. In addition, since the training and testing procedures are not necessarily brief, the *AGREE* message of the FIPA-request protocol is used to notify the initiator that the request is accepted and the result is being computed.

We have developed a variation of these protocols that accept more parameters in order to facilitate the visualization of the process and also simplify the code.

For example, the class OurRequestInitiator extends AchieveREInitiator and accepts a string as a task name or descriptor (printed *onStart* and *onEnd*), as well as callback, in case we need to answer the returned message in any specific way.

The class OurRequestResponder extends AchieveREResponder and accepts a function that will be invoked when an *INFORM* response to the request is received (independently if the “agree” message is sent or not before).

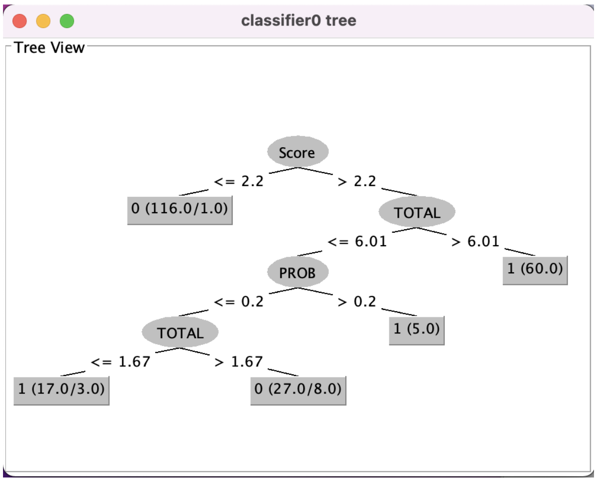
These two protocols are sometimes used as sub-behaviours of *CompositeBehaviour*s. For example, the *UserAgent* performs a *SequentialBehaviour* of two *OurRequestInitiator* sub-behaviours of type Request, one for the training and other for testing. The coordinator has one behaviour of type *OurRequestResponder* to pay attention to the Requests that the *UserAgent* makes and also it performs two *ParallelBehaviours* formed by 10 *OurRequestInitiator* each that request the training and testing to the classifiers. Finally, each *ClassifierAgent* has a behaviour of type *OurRequestResponder* to listen to the requests made by the *CoordinatorAgent*.

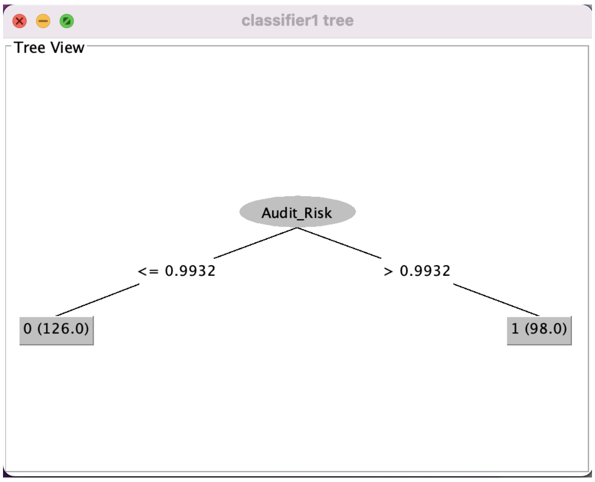
5. Results

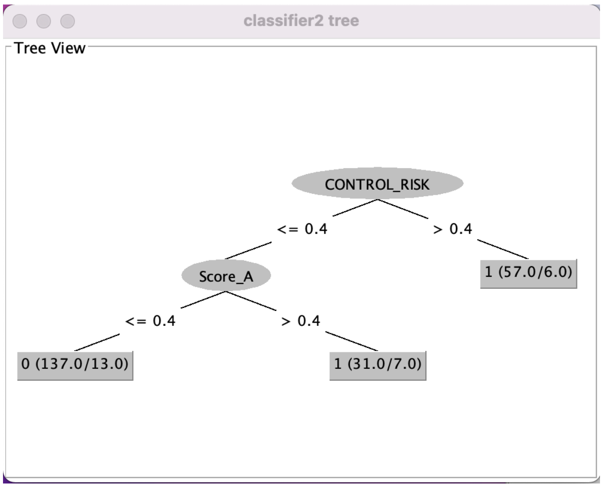
In this section, the results obtained using our MAS are discussed. In particular, the decision trees generated at each *ClassifierAgent* and the performance metrics are depicted and commented on.

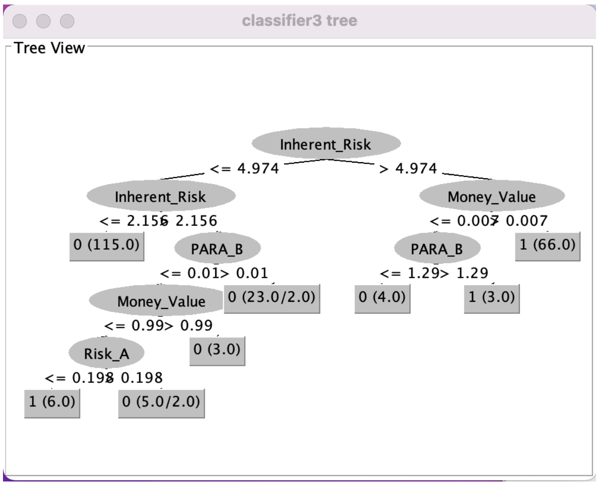
Decision trees generated

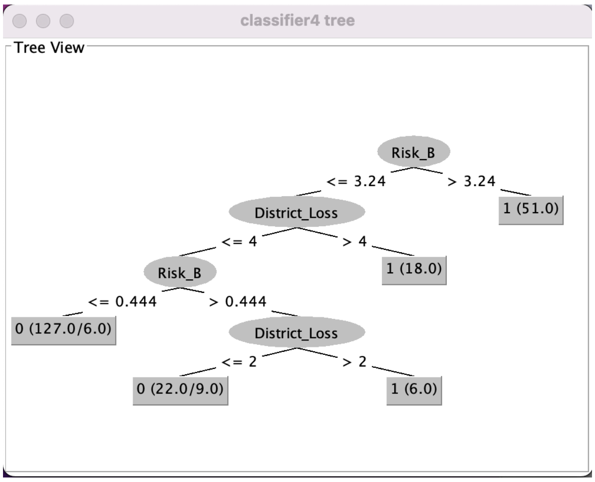
Each of the *ClassifierAgents* has trained a decision tree with the subset of instances and attributes that it received. Accordingly, here we have the trees that were created for each of them:

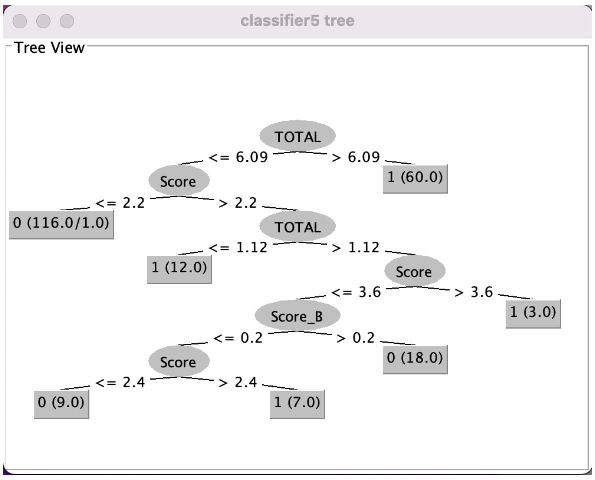


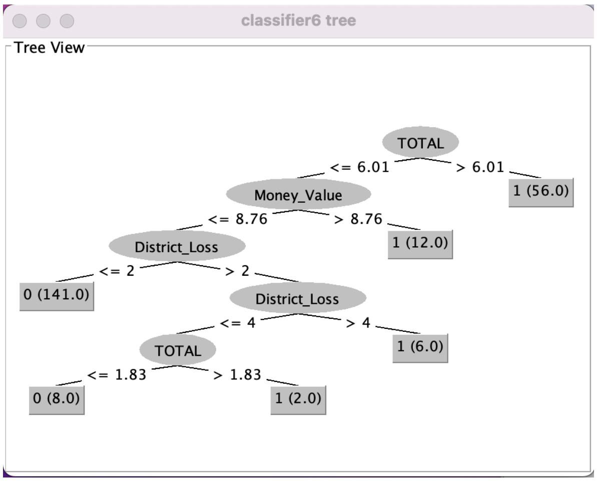


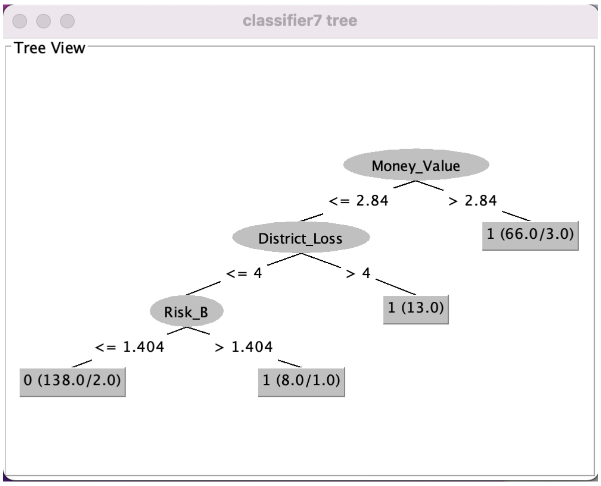


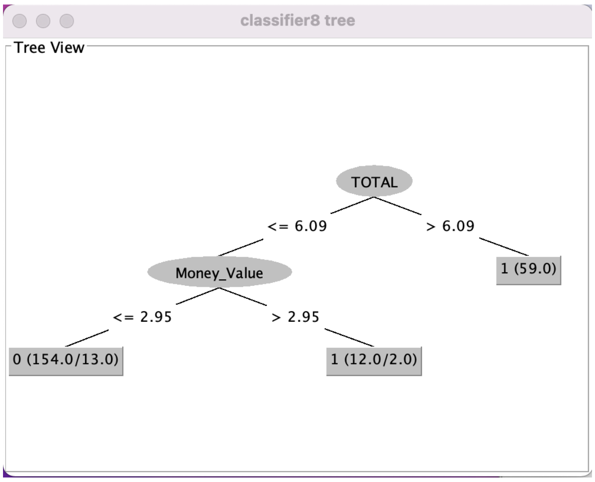


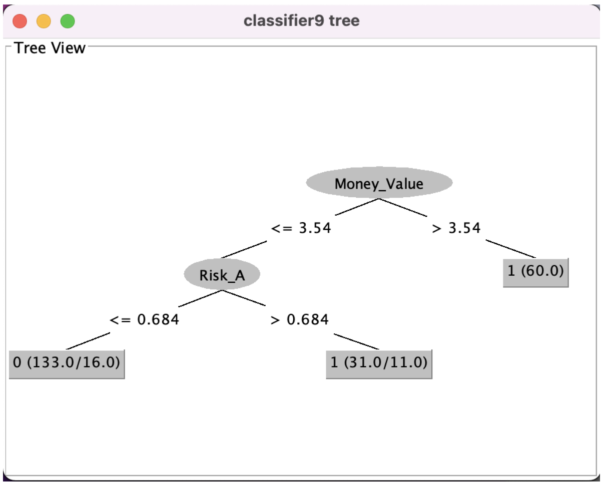












It can be observed that not all the six attributes are used in the final trees. That is because with less of them it is able to obtain quite good results and adding more either does nothing or it is not worth it. For example, *classifier1* is using only one of the attributes, the AuditRisk; while others like *classifier3* is using four of them. This is showing the importance of adding at least once every of the attributes in the dataset in at least one classifier, because otherwise we could be losing some important feature.

Performance metrics

We have performed several clean runs on the architecture to test different performance metrics. First of all, we need to define the concepts true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN).

A true positive is a sample that has been classified as the positive class, when the true label is the positive class. A true negative is just the contrary, a sample classified as the negative class when the true label is the negative class. A false positive is a sample that is classified as positive but the true label is negative. And a false negative is a sample that is classified in the negative class but the true label is positive.

On this basis, we compute the following metrics:

Accuracy: $\frac{TP + TN}{TP + TN + FP + FN}$

Precision: $\frac{TP}{TP + FP}$

Recall: $\frac{TP}{TP + FN}$

F1 Score: $\frac{2 * precision * recall}{precision + recall}$

	ACCURACY	PRECISION	RECALL	F1 SCORE
RUN 1	0.90	0.89	0.85	0.87
RUN 2	0.92	1	0.83	0.91
RUN 3	0.98	1	0.94	0.97
RUN 4	0.96	0.94	0.94	0.94
RUN 5	0.96	0.91	1	0.95
MEAN	**0.944**	**0.948**	**0.912**	**0.928**

We see that the model behaves quite well, having a mean accuracy of 94%. The other metrics are also very high and support the previous reasoning that the model is very robust for this problem in particular.

6. Conclusions

Having seen the result that the model offers, we can conclude that it is an effective and efficient model for the studied problem. Its levels of accuracy and other metrics are quite high and consistent over time, implying that the architecture is robust.

Another of the main conclusions, extracted from the generated decision trees, is that not every attribute is relevant or as relevant as any other to classify a given instance. Sometimes with just one attribute we are able to correctly classify the samples, whereas in other cases we need almost all of the six attributes. That is why it is important to ensure that every attribute is selected by at least one classifier, to avoid cases where important variables are unused. However, this is a theoretical approach and in reality, it could have happened that some relevant attributes are not collected and therefore not taken into account in the classification.

Regarding the Multi-Agent System, we can conclude that it can be a very useful architecture. Nevertheless, it is required to program every communication step very carefully or otherwise it could end up in deadlocks or chaotic behaviours.

Finally we wanted to remark that these types of architectures are not very used nowadays and consequently it is sometimes very difficult to face some of the problems that arise during the programming, being unusual to find related questions/answers at forums and with many of the existing ones linking to closed webs.

7. Annex: Code Documentation

As an addition to our documentation, in this section we provide a more detailed description of our code, dedicating a subsection for each of the main classes.

OurAgent

This is an agent that is created for facilitating the implementation of the rest of agents, extending all agents from this class. This class implements functions that facilitate the message sending, communication between the agents, and the register and search of agents at the DirectoryFacilitator.

FUNCTION_NAME	DESCRIPTION
showMessage(msg)	Prints the message by prepending the agent’s AID
showErrorMessage(msg)	Prints the message by prepending the agent’s AID and the “ERROR” term.

message = createOurMessageRequest (agent, type , content)	Creates a message object with the agent that will receive it, the type of the message, and the data included.
RegisterInDF(type)	To register in the directory facilitator.
[agent] = getFromDF(type)	Returns list of agents in directory facilitator of the specified type.
AgentAID = blockingGetFromDF (type)	It stops until it finds an agent with the specified type.

OurMessage

An auxiliary serializable class which acts as the equivalent to a *C* structs and can be set as content object of an *ACLMessage*. It contains a String that works as the title or type (e.g., used for differentiating between training and test requests) and a serializable object usually employed for the dataset.

OurRequestInitiator

Extends class *AchieveREInitiator* and it is created mainly for avoiding repeating the class definition code each time that it is used. It implements the following functions and handlers.

FUNCTION_NAME	DESCRIPTION
OurRequestInitiator(Agent, request, taskName)	Constructor of the class.
OurRequestInitiator(Agent, request, taskName, consumer)	Constructor of the class with consumer. Consumer will be used for callbacks.
onStart()	Start the task assigned.
onEnd()	Show a message that sinforms that the task is performed.
agent_name = getSenderName(message)	Agent who has sent the message.

On the other hand, it implements the following handlers:

HANDLER_NAME	DESCRIPTION
handleAgree(message)	Agent shows a message that can perform the request.
handleNotUnderstood(message)	Agent shows a message that cannot understand the request.
handleRefuse(message)	Agent shows a message that refuses the request.
handleOutOfSequence(message)	Agent shows a message that is out of sequence.
handleInform(message)	Agent shows a message that informs about message content.
handleFailure(message)	Agent shows a message that something fails.

OurRequestResponder

Child of *AchieveREResponder* which, similarly to *OurRequestInitiator* is created for avoiding repeating code. It has the following methods:

FUNCTION_NAME	DESCRIPTION
OurRequestResponder(Agent, message_template, task, compute_result)	Constructor of the class.
onStart()	Start the task assigned.
onEnd()	Show a message that sinforms that the task is performed.
agent_name = getSenderName(message)	Agent who has sent the message.
message = prepareResultsNotification(request, response)	Prepare results for notification requests.

On the other hand, it implements the following handlers:

HANDLER_NAME	DESCRIPTION
handleRequest(message)	Agent shows a message that is a request. It calls to the computeResult if not agree is sent. Otherwise, computeResult is called at the prepareResponse method.

UserAgent

This agent has two OurRequestInitiator behaviours. One of the behaviors is for the training phase and the other is for the testing phase. To do this, the user has a function setup that registers the two behaviors when it is created. In the setup function the user agent reads all settings and the dataset and tries to find the coordinator agent to save its AID to enable communication with it.

FUNCTION_NAME	DESCRIPTION
setup()	Initialize agent.
readSettings()	Read all settings from the settings file.
readDataset()	Read the dataset from the file.
message = startTrainingOrTestingMS(isTraining, DATASET)	Creates a message with the number of instances according to type of message (training / test) for sending to coordinator Agent.
testData = generateTestInstances(DATASET)	Generate and select the Instances for testing classifiers.
instances_with_attributes = filterAttributes(INSTANCES, DESIRED_ATTRIBUTES)	Selects the instances with the desired attributes.

CoordinatorAgent

This agent has three behaviors. The first of them has requestResponder type, and it is set in the coordinator setup function. The other two are for training and for classification, and they are created when the training and testing phases start.

FUNCTION_NAME	DESCRIPTION
setup()	Starts the agent adding behaviors and registering into Directory Facilitator.
ACLmessage = workingCallback(message)	Creates a Callback for the working phase.
Initialitzation(settings)	Initializes Agent with the settings given by the parameter.
createClassifiers()	Create all the classifiers needed.
getClassifiersAIDs()	Gets all AIDs of the classifiers.
id = classifierNameToIdx(name)	Transforms a String name of classifier to an integer ID.
train(DATASET)	Creates a parallel behavior and sends data training to all classifiers to train a decision tree.
filterAttributes(DATASET, desiredAtrributs)	Filters the desired attributes from the dataset.
trainingCallback(message)	Callback for training phase.
test(dataTest)	Creates a parallel behavior and sends test data to all classifiers.
testingCallback(msg)	Callback for testing phase.

ClassifierAgent

This agent has only one behavior which waits for messages from the coordinator. When it receives one of the two different messages that it can understand, it starts working.

The two messages that it can receive are:

- **train:** When a classifier agent receives this message, it creates a new classifier J48 using the dataset also given in the message. Once a classifier is created it trains the classifier and computes the accuracy. The response is the total accuracy obtained in the training.
- **test:** When a classifier agent receives this message, it calls a function test that uses the classifier created before. This function returns the results of each instance. Finally, its response is all the results obtained for each instance.

FUNCTION_NAME	**DESCRIPTION**
accuracy = train(DATASET, numInstances)	Creates a classifier and trains the model. Returns the accuracy obtained.
classifier = createClassifier(trainingdData)	Creates the decision tree classifier and returns the classifier created.
(ACCURACY) = computeAccuracy(predictions, DATA)	Computes accuracy comparing the predictions with the labels of data.
result = classifyInstance(DATA)	Classifies an instance using the classifier and returns the result obtained.

8. Bibliography

[1] AUDIT dataset. Retrieved on 6/10/2021 from https://archive.ics.uci.edu/ml/datasets/Audit+Data

[2] UCI Machine Learning Repository. Retrieved on 6/10/2021 from https://archive.ics.uci.edu/ml/index.php

[3] C4.5 decision tree algorithm. Retrieved on 6/10/2021 from https://en.wikipedia.org/wiki/C4.5_algorithm

[4] Waikato Environment for Knowledge Analysis (WEKA) library. Retrieved on 6/10/2021 from https://www.cs.waikato.ac.nz/~ml/weka/index.html

[5] WEKA 3.8.5. Retrieved on 6/10/2021 from https://mvnrepository.com/artifact/nz.ac.waikato.cms.weka/weka-stable/3.8.5

[6] J48 decision tree implementation of C4.5. Retrieved on 15/10/2021 from https://weka.sourceforge.io/doc.dev/weka/classifiers/trees/J48.html

[7] Ordered weighted averaging aggregation operator (OWA). Retrieved on 9/10/2021 from https://en.wikipedia.org/wiki/Ordered_weighted_averaging_aggregation_operator