Chapter 6

Framework Definition

An important focus in our research is building a framework that enables us to create, modify, test and export multi agent systems. Such framework allows us to work with different datasets and different preprocess, training and post processing algorithms. The framework was designed and implemented with the goals of flexibility, code reusability and extensibility. The design was focused on three different layers (Figure 6.1):

- The framework, composed primarily of abstract classes and interfaces which allow the different components of an agent system to work together.
- The implementation layer, which provides the real implementation of the different components and techniques.
- The GUI layer, which eases the processes of assembly, training and export of ensemble systems through a graphical interface.

There is a direct dependency of one layer to the one below it. The framework layer is independent, and does not require any of the other 2 layers. This is convenient as anyone may use the framework and provide a different implementation, or even extension to the very framework. The implementation layer depends on the framework, as it consists of a set of techniques, datasets and algorithms which have been implemented. This layer can

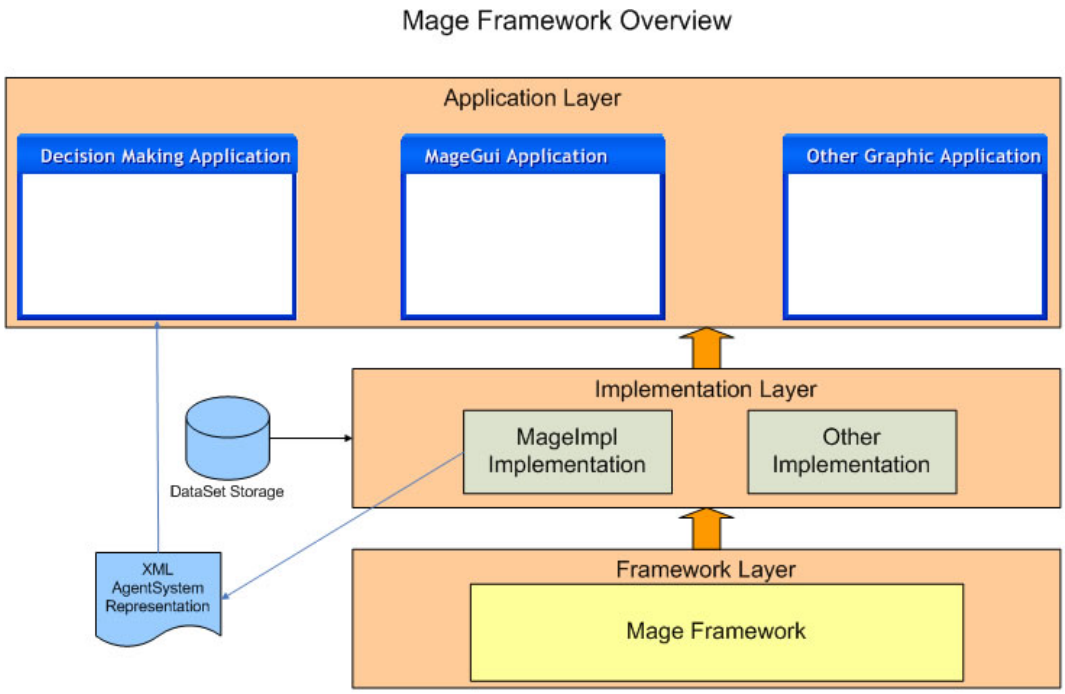

Figure 6.1: General Structure Diagram

easily be expanded with new components. An application may use the implementation layer as a library. The graphic interface layer can be considered one of such applications which make use of the implementation layer.

The implemented System was given the name MAGE which stands for **Multi-Agent Generation System**. The framework layer is then denoted by **mage**, while the implementation layer by **mageImpl** and the GUI layer by **mageGui**.

6.1. The Framework: mage

The framework layer constitutes the bare bones of the system. Through the definition of interfaces and abstract classes, it is the one responsible for calling the corresponding implementation methods and bringing the complete ensemble system together. Figure 6.2 shows the main components of this layer. The features that this layer

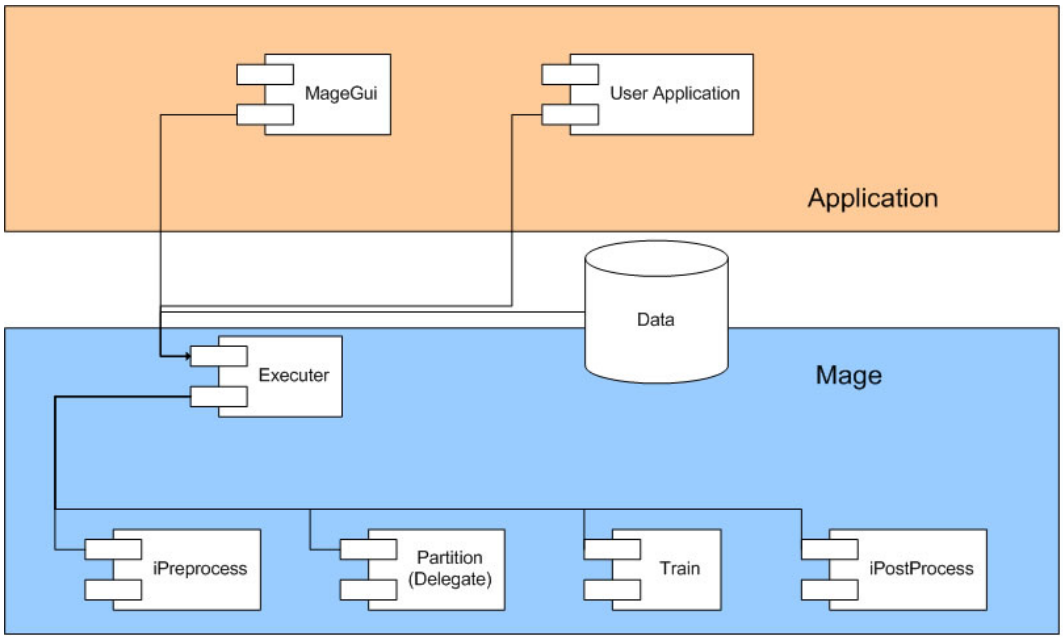

Figure 6.2: Framework Main Components Diagram

presents are:

- Definition of the main structures in the system: iAgent, Agent and AgentSystem, as well as the definition of the different components to be implemented.
- Generation of the agent system from loading the data up to the calculation of statistics in the postprocess stage.
- Execution of learning stage in both single and multi-threaded execution
- Serialization and Deserialization of AgentSystem in XML.

6.1.1. MageTrainer

The main Component in the Framework is the **MageTrainer** class. This class is the executer class of the Framework. Its method **Train** performs the preprocess, train

and postprocess stages and returns an AgentSystem. The general algorithm followed by the Train Method in MageTrainer is the following:

$train(dataSet, preprocess, train, postprocess, agentSystemInfo)$

1 $initialize(dataSet)$
2 $dataSet \leftarrow preprocess(dataSet)$
3 $agentSystemFactory = agentSystemInfo.getFactory()$
4 $agentSystem \leftarrow agentSystemFactory.buildAgentSystem(dataSet, agentSystemInfo)$
5 **for** *each Agent a in agentSystem*
6 **do** $train(a)$
7 **return** $postprocess(agentSystem)$

The train method performs the same algorithm described in chapter 1 and has as a result an Agent System which may be used to make decisions on new data.

An important feature in the train method is the ability to run the train process using different threads concurrently. A parameter may be provided to the run method in the MageTrainer class to indicate concurrent execution. This will generate a different thread for each agent, run them and then wait for all of them to finish. This way, lines 5-7 of the previous algorithm would be replaced by:

$train(dataSet, preprocess, train, postprocess, agentSystemInfo)$

1 ...
2 **for** *each Agent a in agentSystem*
3 **do** $trainThread_a \leftarrow createThread(a)$
4 $start(trainThread_a)$
5 **for** *each Agent a in agentSystem*

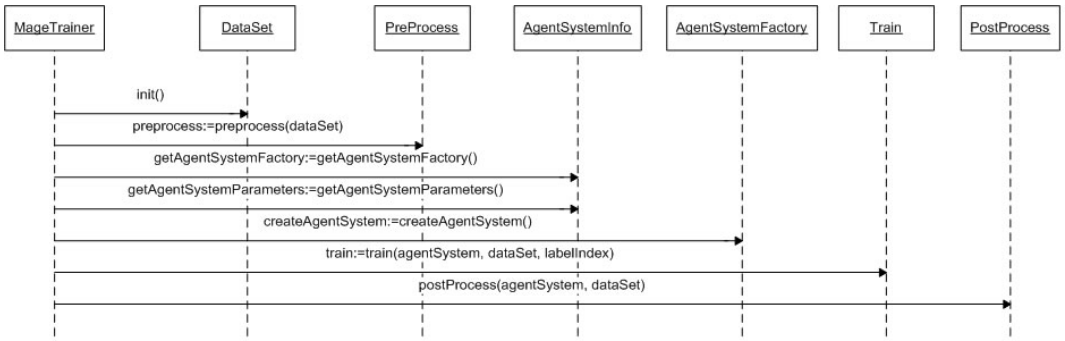

Figure 6.3: Sequence Diagram for MageTrainer's train method

6 **do** $waitFor(trainThread_a)$

7 ...

The MageTrainer class as well provides methods for thread handling. Through the stop, pause and resume methods the execution may be controlled. Figure 6.3 shows the Sequence diagram for the train method.

6.1.2. Component Interfaces and Abstract Classes

The main section of the framework is the definition of interfaces and abstract classes to be implemented. Such classes offer methods which will be called from within the framework but whose real code was provided in the implementation layer. These classes fall under the following categories:

- The entity classes, which represent objects that will be processed. These include **iDataSet, iAgent, Agent and AgentSystem**
- The component classes, which are used for performing the main actions in the creation of the multi agent system. These are **iPreprocess, iTrain and iPostProcess**

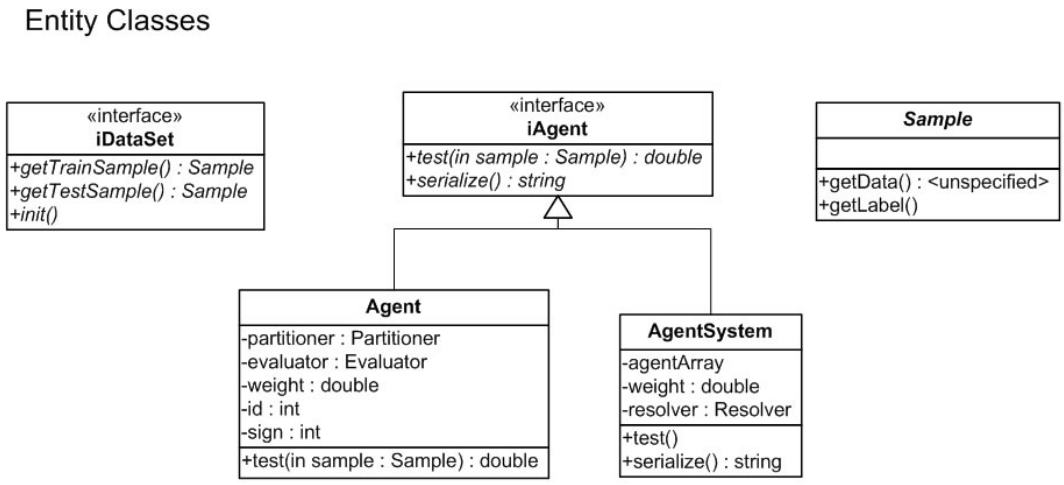

Figure 6.4: Entity Classes in the Mage Framework

- The secondary component classes, which also perform actions on the Entities assisting the component classes
- The delegate classes which receive the responsibility of performing actions for the different agents. These classes include **Partitioner,Evaluator,Resolver**
- The utility classes which allow performing certain operations over the agents. Among these are **ResultStatistics and AgentSystemInfo**

Entity Classes

As noted before, entity classes represent objects that will be modified through the agent system creation process (Figure 6.4). The first component is iDataSet . This interface has the responsibility of representing a data set and providing methods for accessing and modifying test and train samples. Through the usage of the iDataSet interface, there is an storage independency, as it will only be relevant for the implementation layer to know where the data is being stored.

The iDataSet interface makes use of the Sample structure in most of its methods.

This structure represents a standardized way of representing a sample regardless of the implementation. The sample has 2 main standardized components: The data, represented as a double array, and the labels, represented as well as a double array. The Label class is an abstract class, so that any extending classes will have the freedom of any representation they choose as long as for the methods **getValue** and **getLabel** they return the data in the specified format.

Besides of providing Samples, the iDataSet interface also contains methods to gather general information on the dataset, mainly through the **getDataSetInfo** method. This method returns an iDataSetInfo abstract structure, which contains all relevant information, such as number of samples for both test and train sets, size of data in samples and number of labels.

The AgentSystemFactory is responsible of creating the AgentSystem from both the AgentSystemParameters and the DataSet. the AgentSystem class implements the iAgent interface, as does the Agent class. The iAgent interface contains the elemental methods for an agent, which are **test** and **setWeight**. The AgentSystem represents a set of agents or AgentSystems, and the implementation of its test method is based on the agreement of its agents.

The AgentSystem class contains as well a delegate class, which is the **Resolver**. The Resolver is in charge of taking the decision of multiple agents and returning a single response from the complete Agent System. The Agent class contains 2 delegate classes, which are the **Partitioner** and the **Evaluator**. The partitioner is in charge of extracting from the sample the data that the Agent in question will use for its decision. The evaluator is the class which makes the actual decision based on the data returned by the partitioner.

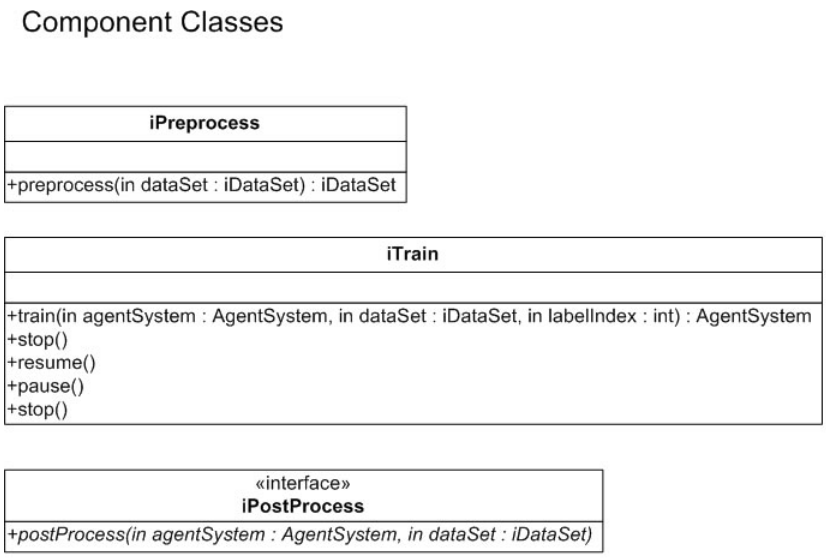

Figure 6.5: Component Classes in the Mage Framework

Component Classes

Component classes are in charge of representing and executing an specific stage in the multi-agent system creation process (Figure 6.5. The preprocess stage is represented by the **iPreprocess interface**, the training stage is represented by the iTrain abstract class and the postprocess stage is represented by the iPostprocess abstract class.

The iPreprocess interface contains a single method called **preprocess**. This method receives an iDataSet implementer as parameter and returns another iDataSet object. The focus on the preprocess method is to modify the received dataset as described in chapter 4.

The iTrain abstract class presents one abstract method to be implemented which is **train**. Train receives an AgentSystem and a DataSet. The received AgentSystem represents the different agents to be considered for building a new System, or the very agents to be trained. This method returns another AgentSystem when training is over. This abstract class contains also methods for execution control. The methods **stop, play** and **resume** allow modifying a monitor variable that lets the training execution

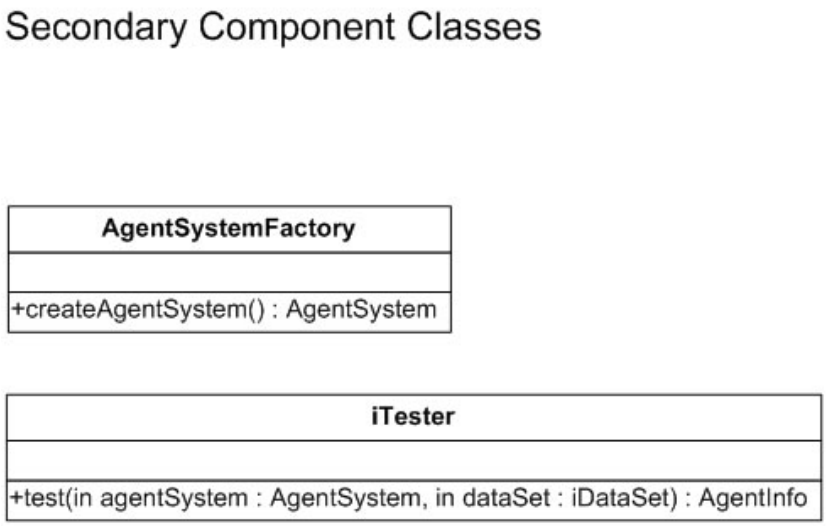

Figure 6.6: Secondary Component Classes in the Mage Framework

be controlled from an external class.

Finally, the iPostprocess abstract class contains the abstract method **postProcess**, which receives an AgentSystem and a DataSet. This method performs the statistics calculation and stores the results on an AgentInfo structure.

6.1.3. Secondary Component Classes

The secondary component classes are also relevant in the generation of the final agent system (Figure 6.6). They are considered secondary, since they do not represent a stage as the component classes do, but provide assistance to those classes in charge of entity transformation.

The first class is the AgentSystemFactory abstract class. This class is in charge of building the AgentSystem to be used in training. The abstract method **createAgentSystem** is the one responsible for this action. Being abstract, the building code will be provided on the implementation layer.

An additional secondary component is the iTester interface. This interface contains

a single method test, which receives an AgentSystem and DataSet and returns a Result-Statistics object. The framework includes 2 implementations of the iTester interface, which are ClassificationTester, which calculates all the classification related statistics, and PredictionTester, which calculates the prediction related statistics.

6.1.4. XML Serialization and Deserialization

The AgentSystem class contains 2 methods for the serialization and deserialization of AgentSystems. Such methods allow exporting all the necessary information regarding the agent system, such that it can be reconstructed either using mage or any other application. A sample XML file generated by mage can be seen in figure 6.7

6.2. The Implementation: mageImpl

As its name implies, this layer provides an implementation of different components of the framework layer. The implementation classes provided fall into two different categories:

- Algorithmic implementations, which include implementations of the iTrain component and delegate classes.
- Experiment related implementations, which include implementations to iDataSet and other component classes.

6.2.1. Algorithmic implementations

The development in the implementation layer was focused on two different extensions of the iTrain abstract class (Figure 6.8). On one hand, Adaboost* was implemented for classification problems. On the other hand, bagging using Neural Networks for each

```
<MageSystem>
    <AgentSystem>
        <Resolver className="mageimpl.MultiClassResolver">
        </Resolver>
        <Agents>
            <AgentSystem>
                <Resolver className="mageimpl.WeightAgreementResolver">
                </Resolver>
                <Agents>
                    <Agent id="322" sign="1.0" weight="0.41280452718296456">
                        <Partitioner className="mageimpl.HardPartitioner" totalAgents="527">
                        </Partitioner>
                        <Evaluator className="mageimpl.SimpleEvaluator" margin="1.0">
                        </Evaluator>
                    </Agent>
                </Agents>
            </AgentSystem>
            <AgentSystem>
                <Resolver className="mageimpl.WeightAgreementResolver">
                </Resolver>
                <Agents>
                    <Agent id="233" sign="1.0" weight="0.534297461333364">
                        <Partitioner className="mageimpl.HardPartitioner" totalAgents="527">
                        </Partitioner>
                        <Evaluator className="mageimpl.SimpleEvaluator" margin="1.0">
                        </Evaluator>
                    </Agent>
                </Agents>
            </AgentSystem>
        </Agents>
    </AgentSystem>
</MageSystem>
```

Figure 6.7: Sample XML Representation of Agent System

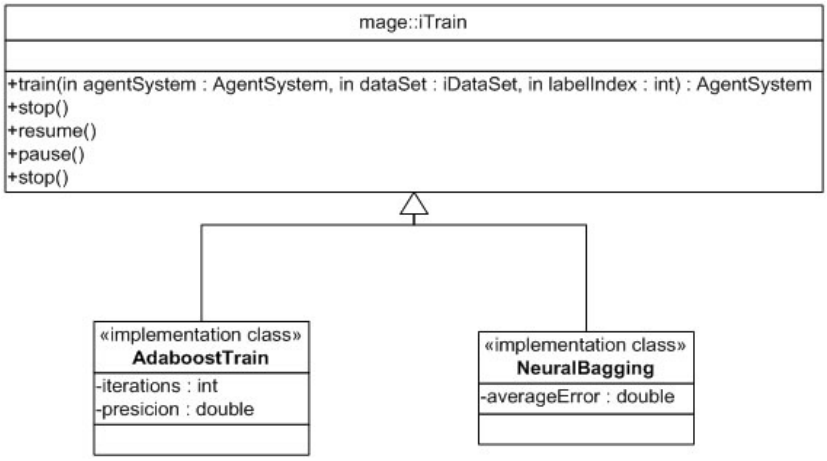

Figure 6.8: Implementations of iTrain interface

agent's decision was implemented for both classification and prediction. Although these implementations are centered on the train component, corresponding delegate classes were implemented.

The Adaboost* implementation

Adaboost* was implemented according to the algorithm presented in chapter 3 in the **AdaboostTrain** class. The train method receives an iDataSet and an AgentSystem structure, and creates a new AgentSystem structure selecting agents from the received parameter.

The Adaboost* implementation includes as well code for execution management, frequently checking a monitor variable through the **doPause()** method of iTrain. It also checks against the **isStopped** variable to identify the case where execution has stopped.

The implementation has 2 parameters, which are the **precision** and **number of iterations**. The precision value v, explained on chapter 3, affects the convergence speed, which the number of iterations indicates how many agents will be chosen for the fi-

nal AgentSystem. The train method finished when this number of iterations has been reached or convergence on the training data has been obtained (which means that all training samples are correctly classified). The parameters have a default value of $v = 0{,}5$ and $totaliterations = 100$, but can be modified through the main constructor, or defined in a properties file to be loaded by the $ClassificationParameters$ utility class.

The Neural Bagging implementation

The neural bagging implementation uses a neural network library created by Aydin Gurel [11]. The train method receives an AgentSystem and a DataSet, and unlike the AdaboostTrain class, it does not create a new AgentSystem but trains the one it receives. As described in chapter three, bootstrap replicates are created from the dataset and then a neural network is trained using this new dataset. The Neural Network parameters are defined based on the dataset information and parameters provided through the $PredictionParameters$ utility class. After training, the Neural Network is used to initialize an Evaluator component through the **init** method. The NeuralEvaluator is the Evaluator designed to store this information, however any other new or existing evaluator could be used as the init method simply receives an object. In the end, a Neural Network will be trained for each Agent in the received AgentSystem.

The specific parameters that can be set for each trained neural network are:

- Number of Layers and Neurons: The number of neurons in the input and output layer depend on the information of the dataset. The number of features in each sample will indicate the number of neurons in the input layer, while the number of labels will indicate the number of neurons on the output layer. However, it is possible to modify the number of internal layers and the number of layers in each. Each layer has an

hyperbolic tangent as the activation function excepting the output layer, which is a linear function.

- Error ratio limit: The error ratio is a measure to track the learning process of a neural network, which indicates the ratio of the new error regarding the old error. The stop condition for training will be when a certain error ratio is reached, which can also be specified as a parameter.

The software used allows other parameters to be modified, however the current implementation does not make use of those parameters and assigns them hard-coded default values.

After training is over, the returned AgentSystem of the train method contains a predictor for each label, where each contains an evaluator with a neural network trained with a bootstrap replicate.

Although it is not coupled with the NeuralBagging implementation, the NeuralEvaluator class is generally used as the evaluator class for agents, as it can be initialized with the trained neural network generated by this method.

6.2.2. Experiment Implementations

For the execution of the multiple experiments in our research, we made an implementation of multiple classes to have all the necessary components to generate the agent system.

DataSets

The implemented datasets were not linked to certain data but to the original data format and the medium of storage. Figure 6.9 shows the different implemented datasets.

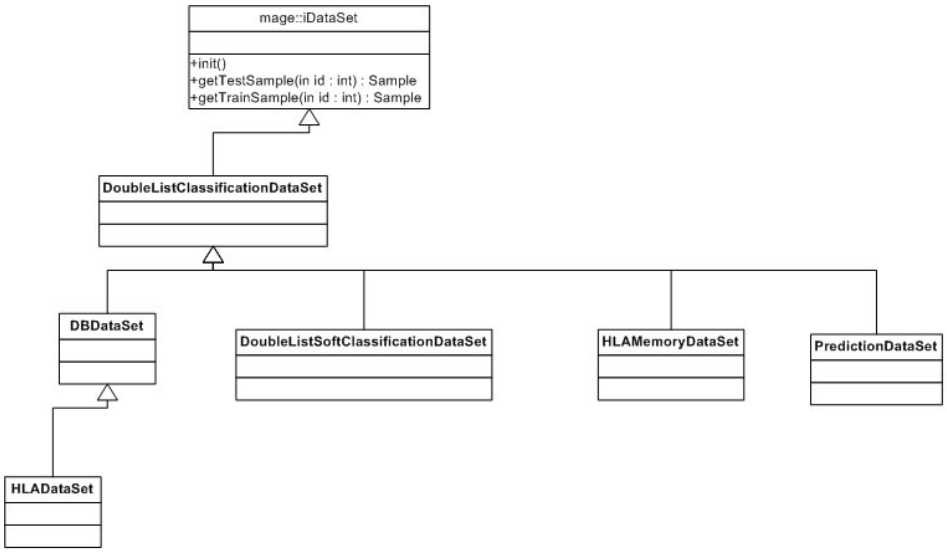

Figure 6.9: Implementations of iDataSet interface

- DoubleListClassificationDataSet: This is the basic implementation of the iDataSet interface. It stores the samples in memory
- DoubleListSoftClasificationDataSet: This DataSet is focused on the soft partition of data, where each agent has access to 2 subsequent features, so that there is one feature shared by two agents.
- DBDataSet: The DBDataSet extends the DoubleListClassificationDataSet changing the storage medium to a database.
- HLADataSet: Extends the DBDataSet adapting to the format of data in the HLA DataSet source document.
- HLAMemoryDataSet: Similar to HLADataSet, but the storage is not on a database but in memory.
- PredictionDataSet: Extends the DoubleListClassificationDataSet adapting the data format for the Boston House Cost prediction dataset.

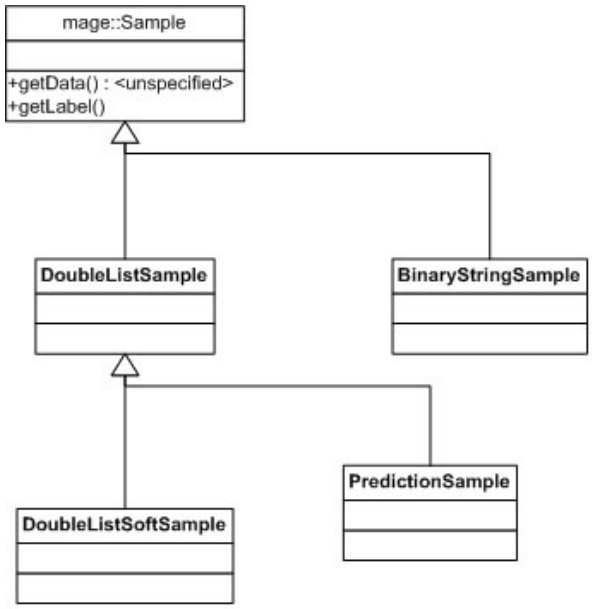

Figure 6.10: Implementations of Sample abstract class

The implementation of DataSets is directly related to the implementation of Samples. The important method in the Sample implementations is generally the constructor which allows reading a String representation of a Sample. The implemented Sample structures are shown in figure 6.10

Components

Besides of the iTrain implementations, implementations are provided for iPreprocess and iPostProcess components. In the case of iPreprocess, a simple implementation is provided which does not modify the dataset. In the case of iPostProcess, two implementations are provided: PredictionPreprocess, which calculates statistics related to prediction, and ClassificationPostProcess, which does the same for the Classification Case.

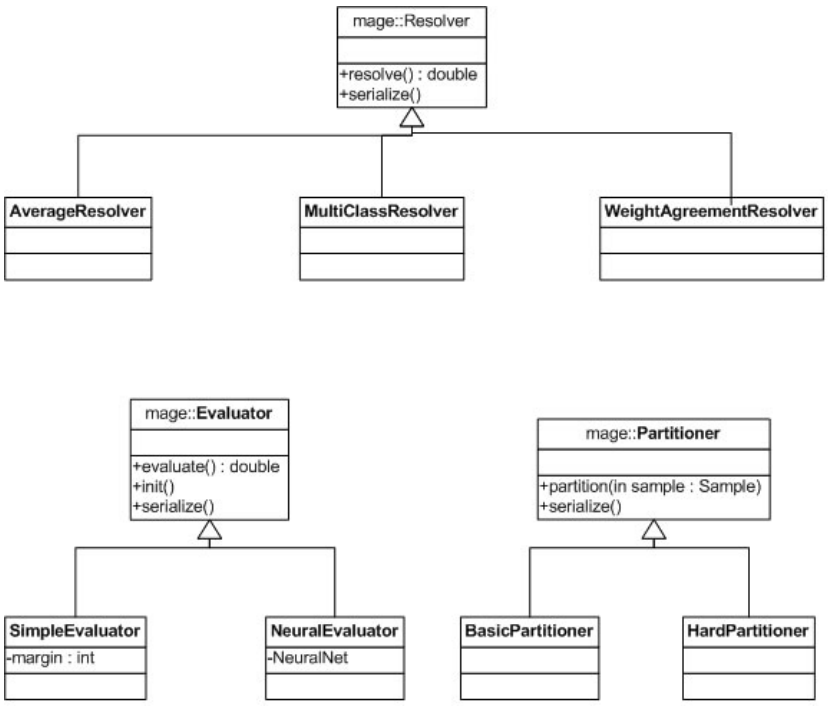

Figure 6.11: Implementations of Partitioner, Resolver and Evaluator (Delegate Classes)

Delegate Classes

Implementations are as well provided for delegate classes (Figure 6.11). In the case of partition, two partitioners are provided. The BasicPartitioner simply returns all the data in the Sample, while the HardPartitioner, based on the agent's id, extracts a subset of data different for each agent.

In the case of the Evaluator, 2 Evaluators are provided. The SimpleEvaluator compares the input value against a margin value and returns 1 if it is bigger or equal, -1 if smaller (Which is an implementation of Decision Stumps). The NeuralEvaluator uses a NeuralNetwork, so that the data are introduced into the input layer and the prediction of the output layer is returned.

Finally, in the case of the Resolver, three different implementations are provided. AverageResolver returns the average of the input data, while WeightAgreementResolver

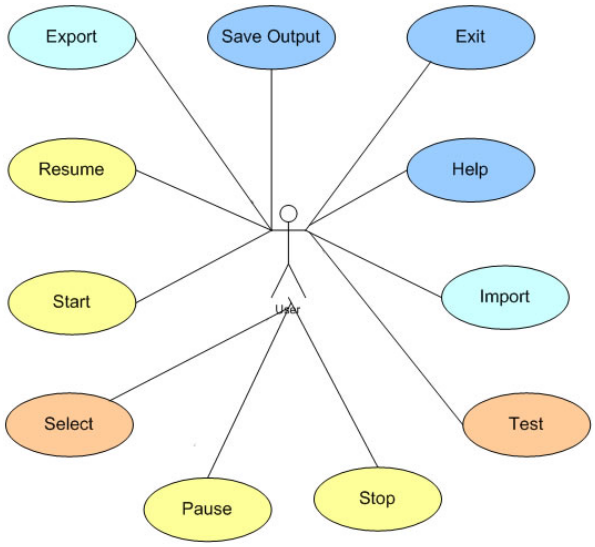

Figure 6.12: Use Case Diagram for MageGui

returns the weighted sum. Multi-class resolver returns the label corresponding to the maximum value.

Secondary Components

For the secondary components, 2 implementations for the AgentsystemFactory are provided, again focused on the classification and prediction case.

6.3. The Application: magegui

An application was created to ease the execution and control of Agent System creation. This application, code named **MageGui** makes use of both the implementation and the framework layers to load components, train, test and export an Agent System. It's main features, shown in the use case diagram in figure 6.12, are divided in 4 different categories:

- Framework Interaction Features: These features are involved with working with the

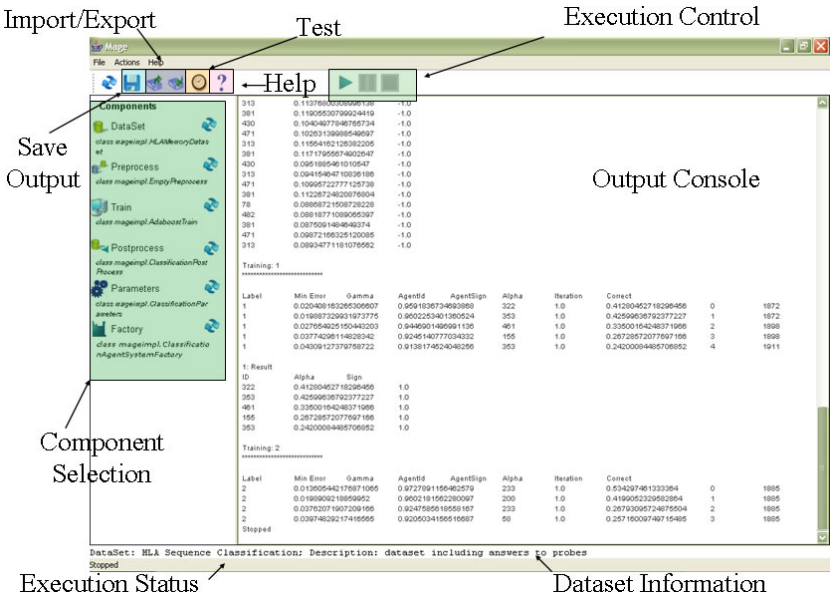

Figure 6.13: Screenshot of GUI Application. Different features controls are indicated in the figure

Framework and the generated Agent System. Selecting the different components and testing the resulting AgentSystem fall into this category.

- Export/Import: As the name says it, these features are involved with the serialization/deserialization of Agent Systems.
- Execution Control: These features are related to controlling the training process.
- Interface Specific: These features are more focused on characteristics necessary to make it easier for the user to use the application.

A screen-shot of the application can be seen in Figure 6.13.

6.3.1. Framework Interaction Features

There are two main features that allow the application interaction with the framework. First is the component selection feature. Using class reflection, all the imple-

menters of the different interfaces and abstract classes in the framework which are included in the mageimpl package are detected and added to a list. This way, the user is able to select the components he wants to use to build the Agent System. The second feature is the ability to test either a recently created agent system or an imported agent system using the iPostProcess component and a DataSet only. The test feature logs all the results to a file, based on the data generated by the postprocess method.

6.3.2. Export/Import

The export and import feature makes use of the XML serialization capabilities of the framework to either save the agent system into an XML file or load an existing agent system from an XML file. This feature makes it easy to test already existing agent systems and have a tangible representation of the system after training.

6.3.3. Execution Control

Execution Control is focused on allowing the user to control the execution state of the process. The user may start execution, pause, resume and stop it. This grants him the flexibility of focusing the machine resources to a different task or evaluate intermediate data, as usually training processes occupy a considerable amount of processing capability and memory.

6.3.4. Interface Specific

This final set of features is focused on making it easier for the user to use the application. The user may save the output into an external file and check the help documentation. As well, the current state of the system is always clearly visible to him through the status bar.