Classifier Workbench

Gianluigi Ferraris

The usage of classifier system requires the implementation of a quite complicated computer code and the test of several parameters' configurations since parameters' values must be tuned and adapted to each particular situation. Practically, to work with classifier systems one should acquire specific knowledge and devote a lot of time to "tuning" operations. CW simplifies the approach to classifier systems by allowing users to handle them easily.

The CW code contains a generalised classifier system and an application (a workbench): while the former copes with a variety of problems, the latter simulates them. The workbench helps the user in choosing the values of parameters and in verifying the classifier system's performance. Then he can include the kernel of the CW (i.e. the generalised classifier system) in any Swarm model and use classifiers without writing a specific code. In addition, he can directly specify the best parameter-configuration he has tested using the workbench.

The CW code is written in Objective-C, to be used in Swarm. Its architecture is based on the Environment - Rules - Agents framework (Terna, 1999) (hereafter ERA). ERA splits a model in four levels: environment, agents, one or more rule masters and rule makers. The "rule master" selects rules whereas the "rule maker" generates new rules. Each agent owns a datawarehouse that stores its rules, and uses an interface to translate data from the metrics of the classifier system to the desired /actual one and vice versa.

In CW, the rule master, after calling the detecting routine (a detecting routine performs the coding of environmental information in the body of a message and adds it to the message-list), carries out the auction between the rules. The latter is activated by a matching the messages in the list and selects those that will be allowed to post their message. From time to time, the rule master asks the rule maker to let the rules contained in the datawharehouses evolve. Whereas the rule master performs credit apportionment, the rule maker performs the genetic algorithm. (As Goldberg (1989) points out, in a classifier system there are three main components: rule and message system, apportionment of credit system, genetic algorithm).

The kernel of CW is a learning classifier system. CW can manage cooperation among rules: a rule may write messages to activate others rules or to suggest actions through the effectors (effectors are Swarm objects able to suggest an action, to be executed in the environment. Matching one of the messages in the list can activate each effector). Each rule is a Swarm object that contains two arrays of characters: the condition part and the action part. A rule can be read as "if <condition part> then <action part>" The condition part verifies the match with the messages in the list, the action part is the text of the message that each rule will put in the list as it is asked to fire. All matching rules are asked to bid in an auction whose winners are allowed to post their messages in the message list. Each message is a Swarm object containing an array of characters in which the action part of the rule is copied or, if the message is made by the detecting routine, the environmental information is coded in the same format. The action part has the same length as the condition part so that messages posted by a rule can be used either to activate effectors or to trigger other rules.

Each position of the action and condition part of a rule contain the values '0', '1' or '#'. Environmental information is encoded, with the same format, by the detecting routine. Each position represents an environmental characteristic or an event: for instance, the value '0' mean that the environment has not a given characteristic or that the event has not happened, the opposite holds for value '1'. Usually value '#' is not used for environmental messages. The effectors are designed as rules: their action part can be read as a map of actions that will be completed following the value of the single position.

CW deals with quite complex behaviour. Of course, CW can also manage simple decisions about single quantities, such as prices and so on. To simulate those complex problems we use, in the workbench, an object named "lottery". It gives each agent a random integer number, in the range chosen for the inputs, then arithmetic operations are performed to obtain a second number, in the range of the output values. Agents, i. e. the classifier system, have to guess that second number. This kind of simulation of problems, where several data has to be used in handling a set of actions, can be easily performed. For example, we can imagine that each position of the binary translation of the input number, can represent a status of an environmental element or variable, where '0' may mean false and '1' true: randomising numbers in a finite interval of values, the lottery can simulate a lot of complex environmental stati. The classifier has to match each set of stati with an appropriate strategy that may be represented as set of flags, whose value can be either '0' or '1'. We can interpret binary translation of the correct output number, as the best strategy the agent adopts.

CW is fully independent from the semantic meaning of the rules, and therefore the use of arrays with mixed meaning is allowed. For instance, the user can code into the environmental message a price and a set of flags

representing conditions. The same holds for actions too: the action can be translated to obtain, say, the quantity that the agent is going to buy or sell, while other parts are read as flags used to decide actions the agent has to perform.

The early stages of the population of rules can be:

1) loaded from a file, whose name is stored into a variable in the modelSwarm;
2) randomly generated.

Since the rules of each agent are stored in its own datawarehouse, in the same model the user can introduce agents with either loaded rules or randomised rules.

The values of the parameters driving the behaviour of the classifier system and of the genetic algorithm are stored in the object called classifierParm, whose *id* is kept in the datawarehouse. In this way, the user can model agents that deal with different classifiers and use the same ruleMaster and ruleMaker instances. The data for classifierParm are set to default values in createEnd method. They can, also, be loaded from a file, whose name is stored into the model. For each parameter the class ClassifierParm allows getting and putting the value through methods called "get" + the name of the parameter or "set" + the name of the parameter.

1 - How to use CW

To use CW we have to include the kernel (i. e. classes: RuleMaster, RuleMaker, Rule, Effector, Message, Treasury, ClassifierParm, Shuffler, Dump) in the project, making a few customisations of the parameters stored in classifierParm and of the interface object, which deals with the semantic interpretation of messages and actions.

Let's show the action of the model:

a) The agent is asked by the environment to do something, usually "a" is the selector "step"

b) The agent asks the environment for information

c) The agent asks its interface to translate the environmental information in an array for the classifier system, i. e. the ruleMaster.

d) The agent asks the ruleMaster for advice by means of the selector:

   "applyRuleTo: (DataWarehouse *) dataWarehouse with: (id) interface"

   where dataWarehouse and interface are the ids of its instances of classes DataWarehouse and Interface.

e) The ruleMaster asks the interface for the eventsMap, i. e. the translation of environmental information.

f) The ruleMaster asks the dataWarehouse for the lists of rules, messages, effectors, worklists and so on.

g) The ruleMaster asks the classifierParm for parameters that will be used to handle selection and evolution of rules.

h) The ruleMaster sends statistical data to the dataWarehouse.

i) When an advice is given, i. e. an effector has been activated, the interface is told to set the suggestion map, i. e. an array that contains the actions suggested by the activated effector. These actions are coded in the metric of the classifier: the interface will provide the translation when required by the agent. In order to receive suggestions, the class Interface has to be able to deal with the message:

   "setSuggestionMap: (char *) x"

where x is the address of the first element of the array of characters that represents the suggested behaviour or strategy.

j) The agent asks interface to translate advice.

k) The agent performs the suggested action in the environment

l) Evaluating the results of the action it has just performed, the agent computes a reward for the rules and tells it to the ruleMaster using selectors:

"setReward: (double) w to: (DataWarehouse *) dataWarehouse".

Note that w cannot be negative.

m) RuleMaster asks the dataWarehouse for the lists and accomplishes credit apportionment by means of an algorithm that is similar to the "Bucket Brigade" described by Goldberg (Genetic Algorithms in Search Optimization & Machine Learning - 1989 Addison Wesley Longman):

> "… as an information economy where the right to trade information is bought and sold by classifiers. Classifiers form a chain of middlemen from information manufacturer (the environment) to information consumer (the effectors)."

The algorithm is based on an auction among activated rules (i. e. classifiers). Each rule has to pay a tax (bidTax) to the treasury to participate. Each winner has to pay an amount in proportion to its strength to the rules that has made the message it has matched. Each rule may have incomes from other rules, matched by the message it has made, or from the environment. In this way, the better is the rule, the stronger is becomes.

n) Following its parameters, ruleMaster manages changes in rules by asking ruleMaker for evolution. RuleMaker is told ids of dataWarehouse and classifierParm and it is going to ask for lists and parameters.

2 - How to build objects and initialise them

The kernel of CW contains three kinds of objects: main, service and data objects. Main objects are instances of classes: Interface, DataWarehouse, ClassifierParm, RuleMaster and RuleMaker. Data objects are rules, messages, effectors, treasury, several lists and so on: they are created by dataWarehouse instances. Service classes are Dump, Shuffler and Trace; only an instance of Dump and Shuffler is needed; Trace is used, simply, importing the file Trace.h

In the method "buildObject" of the modelSwarm various number of instances have to be created. Private interfaces or dataWarehouses are not technically requested, but the isolation of the data of the single agent is strongly recommended to obtain a more intelligible model.

The user can make several classifierParm to manage different classifier systems in the same model; different ruleMaster and ruleMaker are not mandatory required, but several instances of them can be created and used.

The creation of all the objects is obtained trough the usual sequence of "createBegin" and "createEnd". Between createBegin and createEnd values of specific variables can be sent through appropriate selectors. All selectors that give a value to be set are named beginning with "set"; all selectors that ask for a value are named beginning with "get".

Trace.h contains several macros to allow the objects in the kernel to print information about their actions. No instances of Trace have to be created. Dump is used to take dumps, i. e. to print the contents of parts of the storage in a standard form. These objects will be useful in solving problems and finding bugs. Shuffler is used to shuffle a list but, now, a specific pre-defined "shuffleList" is available in Swarm.

First the user has to create service-objects, with the statements:

```
"dump       =    [Dump createBegin: self];"
"dump       =    [dump createEnd];"

"shuffler   =    [Shuffler createBegin: self];"
```

```
"shuffler          =        [shuffler createEnd];
```

Then ruleMaker and ruleMaster have to be created:

```
"aRuleMaker        =        [RuleMaker createBegin: self];"
"                           [aRuleMaker setDump: dump];"
"                           [aRuleMaker setShuffler: shuffler];"
"aRuleMaker        =        [aRuleMaker createEnd];

"aRuleMaster       =        [RuleMaster createBegin: self];"
"                           [aRuleMaster setDump: dump];"
"                           [aRuleMaster setShuffler: shuffler];"
"                           [aRuleMaster setRuleMaker: aRuleMaker];"
"aRuleMaster       =        [aRuleMaster createEnd];"
```

At least one instance of classifierParm is required:

```
"aClassifierParm =          [aClassifierParm createBegin];"
"aClassifierParm =          [aClassifierParm createEnd];"
```

the parameters can be set through specific "set" messages, but usually they are loaded from a file whose name can be stored in a variable:

```
"char * parmSet =           "ClassifierParm.dat";"
"                           [aClassifierParm loadFromFile: (char *) parmSet];"
```

Several agents can share the same datawarehouse or interface, but usually an instance of the objects is created for each:

```
"aDataWarehouse =           [DataWarehouse createBegin: self];"
"                           [aDataWarehouse setClassifierParm: aClassifierParm];"
"aDataWarehouse =           [aDataWarehouse createEnd];"

"anInterface       =        [Interface createBegin: self];"
"                           [anInterface setClassifierParm: aClassifierParm];"
"anInterface       =        [anInterface createEnd];
```

At starting time rules can be loaded from a file:

```
"                           [aDataWarehouse loadFromFile: (char *) filename];"
```

or can be set at random:

```
"                           [aDataWarehouse setAtRandom];"
```

All data objects are created by dataWarehouse in the methods loadFromFile or setAtRandom.

Sending the appropriate selector followed by the value of the right kind can modify the value of all parameters in classifierParm. Selector’s names are made with "set" followed by the name of the variable that is going to be modified.

The agent has to receive and store ids of its dataWarehouse, interface, and ruleMaster. The agent needs these data to be able to communicate with ruleMaster and interface.

3 - The parameters

The classifierParm contains three kinds of parameters: general parameters, auction parameters and evolution parameters.

i) General parameters are:

NumberOfRules: (integer) the number of individuals going to be created and used in the classifier system

NumberOfEffectors: (integer) the number of actions the ruleMaster is able to suggest.

GeneLength: (integer) number of positions that each array, i. e. condition or action part of a rule or effector or body of a message has to store. The maximum allowed number of positions is 256, but with a few intervention in the classes it can be modified.

MaxNumberOfMessages: (integer) the highest number of messages that can be produced in each run. If this number is set to 1 no cooperation between rules is allowed.

EffectorsFlag: (integer) if set to 1 the action part of each rule is obtained copying a condition part of an effector. Otherwise the action part is set at random.

WildCardRate: (float) the allowed percentage of wildCard in random generation.

Confidence: (float) the minimum level of errors for the classifier. If the ratio between the times the agent has given reward zero to the rules and the number of suggestions given in the latest period, is less than confidence value, then the evolution, i. e. the search for new rules through the ruleMaker, stops. This implies that the ruleMaker is asked to generate new rules only when the zero reward-confidence ratio is higher than "confidence". To avoid this behaviour the user can give the confidence parameter a negative value.

InitialStrength: (float) the initial amount of credits of each rule.

ii) Auction parameters which handle the flow of strength are:

BidTaxRate: (float) the percentage of credit that each matched rule has to pay to make a bid in the auction.

LifeTaxRate: (float) the percentage of credit that all rules have to pay all times.

The treasury cashes all taxes

BidRatio: (float), linearBid1: (float), linearBid2: (float). EffectiveLinearBid1: (float), effectiveLinearBid2: (float), bidSigma: (float), bidMu: (float) are used to compute the bids each matched rule can do in an auction and will have to pay if win. Each rule makes an effectiveBid computed as follows:

EffectiveBid = bidRatio * (effectiveLinearBid1 + effectiveLinearBid2 * specificity) * strength + W * bidSigma + bidMu;

Where specificity is the relative specificity of the rule: the ratio between the number of # in both action and condition part, and the length of the rule (i. e. the number of positions of the rule). W is a random real number between 0 and 1. EffectiveBid is the amount each rule offers in the auction.

Bid = bidRatio * (linearBid1 + linearBid * specificity) * strength;

Bid is the amount the winner has to pay to the authors of the messages it has matched. If the author is the detecting routine the treasury cashes the amount.

iii) Evolution Parameters which handle genetic manipulation are:

EvolutionRate: (float) the probability with the ruleMaster calls the ruleMaker. If it is set to 1 then the ruleMaster calls the ruleMaker after a number of selections equal to the number of rules: in this way all rules have the possibility to be evaluated. If it is set to zero the classifier becomes a non-learning classifier.

TurnoverRate: (float) the portion of population that will be involved in each evolution.

CrossoverRate: (float) the probability of crossover

MutationRate: (float) the probability of mutation

CrowdingRate: (float), crowdingFactor: (float) the percentages of rules that will be compared to pick up the worst rule and the rule which is most similar to the new rule. The rule picked up will be dropped to make room for the new one.

4 - How to use the workbench

In the probe-map of the observer the user can choose the frequence of display and set a print flag. If the flag "printDataWarehouse" is set to 1, in addition to each display each dataWarehouse is sent the message "print". All the data objects in the kernel can handle this message printing their contents: this way each dataWarehouse can print its contents simple sending a "print" message to all rules, messages, effectors and so on.

In the probe-map of the model the user can set the number of agents, the number of events, actions and situations. The number of events is used to compute the number of positions needed for each array: if you set four events each condition, action, message will be four positions long. “Number of actions” is the number of actions the ruleMaster can suggest. Number of situations is the number of different combinations of events that the lottery has to generate.

Since a combination of '0' and '1'can be read as a binary number, to generate several situations the lottery, simply, generates a random natural number between zero and the chosen number of situations. In so doing, it is possible to test the behaviour of the classifier in different cases, for example you can specify a number of events, i. e. positions, higher than what is needed, to verify if the classifier is able to obtain rules with some # characters at the start of the condition.

To link each situation to a correct answer, the lottery divides the random number by the number of actions and takes the rest as the correct answer for the situation. In this way becomes easy to link different number of situations to actions.

Also the verification of the rules in the dataWarehouse is simplified: using numbers each rule has to map in the condition part a different integer between those allowed as situations and in the action the rest of the division. If the number of positions is not high it is possible to verify the rules simply by reading their body.

If the winningsTreshold, the last value in the probe-map of the models, is set to 0 no modifications in the behaviour of the lottery are made. If the value is higher than zero and at least one agent achieves this number of consecutive correct answers the lottery switches manipulation by adding, or not adding, one to that number before dividing it. In this way all correct answers change and the classifier has to learn again.

NumberOfRules, maxNumberOfMessages, effectorsFlag are used to override corresponding parameters of classifierParm.

The graphic produced by the observerSwarm shows the learning process of the agent. For each period the number of correct answers given by each agent is reported.