MSc Knowledge Engineering
Second Assessed Practical

Consensus Learning Among Autonomous Agents

February 25, 2005

In this practical, you will implement a multiagent system in which agents exchange hypotheses about a learning problem to reach agreement about the best hypothesis. Use the `submit` procedure to submit your solution by executing the command

```
submit msc ke 2 <your-filename>
```

on any DICE machine. The deadline for submission is **Friday 18th March at 4pm**.

Your solution should consist of a gzipped tarball[1] that contains a set of Java source code files and a text document that contains a discussion of your results and, if necessary, additional documentation for the code you submit.

1 Introduction

One of the key advantages of multiagent systems is the fact that they enable spatial distribution of knowledge and local evaluation of results. Unless the communication and coordination overhead induced by this decentralisation becomes too costly, it may aid to substantially reduce the complexity of the original task.

In this assignment, we are going to consider a set of agents each of which locally maintains a set of learning examples and a local learning hypothesis that is used to classify these examples. Using a blackboard system for communication, each agent may suggest to the community of agents to adopt her own hypothesis, wherupon other agents will compare this hypothesis to their own previous hypotheses, and will either accept others' proposals or reject them and make a new proposal. Thus, agents will (hopefully) cooperatively form improved hypotheses and the system will terminate after some agreement criterion has been met (e.g. if the majority of the agents has accepted the hypothesis).

2 The System

On the module Web site (`http://www.inf.ed.ac.uk/teaching/courses/ke`) you will find a tarball ke.tgz containing Java source files for the following classes:

- `Start` – A class to run agent simulations, contains the `main` function
- `SystemManager` – The system manager component that initialises the agents and spawns threads for them; it also maintains the blackboard onto which messages are posted by agents and decides when to terminate the simulation.
- `Agent` – The main agent class; each agent runs in its own thread of control and has a built-in reasoning loop in which it decides whether to generate a new message or not. It also maintains a set of learning samples for evaluation and a local current learning hypothesis.

[1] See `http://www.inf.ed.ac.uk/systems/support/FAQ/#D5` for instructions on how to use tarballs.

- `Hypothesis` – Class used to represent learning hypotheses, to generalise from two hypotheses and to evaluate a hypothesis on a given learning sample.
- `Message` – A class for representing messages posted onto the blackboard; each message contains the sender's name, a performative, and some content (usually a hypothesis)

These classes provide the core structure of the system and its basic run-time functionality. More particularly:

- The code contains control loops for the system manager and a fixed set of agents (embedded in implementations of the `run()` method of the `Runnable` interface). In other words, the system will start concurrent threads for the manager and all agents upon start up automatically. The `run()` methods in the `Agent` and `SystemManager` classes are not to be modified!
- The basic reasoning cycle of each agent is embedded in the `Agent.reason()` method that is called by each agent in every reasoning cycle. Currently, all this method does is to generate a message, but obviously this should be the part of the code in which the agent's reasoning is performed. Note that unless `null` is returned, the message will be automatically dispatched for publication on the system manager's blackboard.
- Agent and system manager threads (and with them the whole system) are terminated when a user-defined termination criterion is met, which is checked for in the main processing loop of the `SystemManager` class. This should be used to implement a method of checking whether agents have reached an agreement. Thus, each simulation run can be viewed as a conversation between the agents which terminates with an agreement (if it terminates at all).

In the following, you will be asked to use the existing code and extend it to implement a multiagent consensus learning system.

Q1: Compiling and running the system, inspecting the code (0%)
As a preliminary exercise, compile the classes using the `javac` command and run a simulation using the command

```
java ke.Start
```

The system will start one hundred agents which send some messages to the blackboard and then terminate after a short while.

Familiarise yourself with the source code of the classes provided to gain an understanding of the basic control flow and the distributed nature of the application.

3 Distributed consensus learning

The basic learning problem the agent society is confronted with is specified as follows:

> Each agent is presented with a set of instances (learning samples) with boolean attributes. The hypothesis space contains all conjunctions of lists of (potentially negated) attributes. The task of the society is to determine a hypothesis that will maximise the number of correctly classified samples (in the sample sets held by the participating agents).

To perform this task, interaction among the agents unfolds in the following way:

Agents `propose`, `reject`, `accept` or `withdraw` hypotheses, depending on the current best hypothesis they can determine. Let us call all hypotheses that are currently visible on the blackboard "open" hypotheses, and the ones that have either been withdrawn by their proposer or removed by the system manager because they were rejected by too many agents "closed". We refer to a hypothesis that is accepted by enough agents for the system to terminate as a "solution". Then, more specifically, an agent

- can use her current own hypothesis and other open hypotheses from the blackboard to determine a new hypothesis in each reasoning cycle (either by (i) keeping her old hypothesis, (ii) replacing it by another open hypothesis or (iii) creating a completely new hypothesis by generalising over her current hypothesis and some open hypothesis)
- proposes a hypothesis if she has discovered a hypothesis that performs better on her examples than those currently open
- withdraws an open hypothesis proposed earlier by herself if she has discovered a better one
- accepts a hypothesis if it appears better than any of the hypotheses she can think of
- rejects any open hypotheses that perform worse on her examples than some other open or self-constructed hypothesis.

In that, agents are not allowed to store previous hypotheses, they can only use the ones that are currently visible on the blackboard and their current own best hypothesis.

It is the system manager's task to maintain the set of open hypotheses on the blackboard and to determine when an agreement has been reached. For this purpose, the system manager has to:

- Keep track of the percentage of agents that have accepted/rejected an open hypothesis. In general, the termination criterion that will be used is

  > Agreement on hypothesis h has been reached if at least p percent of the agent population have accepted it.

  Once any open hypothesis has been agreed upon in this sense, the system terminates and outputs this hypothesis as a solution.
- Remove hypotheses (and statements in favour of/against them) that have been withdrawn by the agent who created them.
- Remove hypotheses for which there has been sufficient rejection so that the necessary percentage of approval cannot be reached anymore.
- Remove proposals by different agents concerning the same hypothesis (more precisely, replace any redundant `propose` messages by respective `accept` messages on the blackboard).

4 Application: Learning diagnostic rules for a new disease

The application scenario for the implementation of the consensus learning approach is reaching agreement over pre-diagnostic rules for an unknown disease. We assume that each of the agents represents a medical centre that has patient data regarding symptoms and test results for a new virus. Each learning sample is given as a list of symptom values (true/false) for the following ten symptoms

CongestedNose, *Cough*, *Headache*, *Fever*, *Indigestion*, *Insomnia*, *Itch*, *Rash*, *StiffNeck*, *Toothache*

and a boolean classification for whether the patient tested positively for the new virus. For example, if $Cough = true$ for a patient and $Test = true$ then the patient had a cough and was found to be infected with the virus.

Now let us assume the virus test itself is extremely expensive, so doctors would like to have a good diagnostic rule for when to conduct these tests, and that they want to reach agreement over which combination of symptoms seems to predict a positive test result for the virus using their (disjoint) sets of patient data.

In other words, they want to agree on a common hypothesis by evaluating each other's hypothesis and modifying their current hypothesis with those of others.

For the purposes of this assignment, you are provided with data files for five agents (`agent0.data` to `agent4.data`) each of which contains ten patient data records (rows) with eleven columns that correspond to the ten symptom categories listed above (in the alphabetical order used above) and (in the rightmost column) the virus test result for the respective patient, i.e. his/her classification ("1" denotes "true" and "0" denotes "false" in all columns).

For example, the column "1 1 1 1 0 1 0 0 0 0 1" denotes that the patient had a congested nose, a cough, a headache, fever and insomnia but neither of the indigestion, itch, rash, stiff neck or toothache symptoms, and that this patient was tested positively for the virus.[2]

The hypothesis space agents may use is defined as the set of all possible conjunctive statements about (possibly negated) symptom values that would yield a positive classification. Example for such hypotheses are:

$$\begin{gathered} \mathit{Cough} \wedge \neg \mathit{Fever} \wedge \neg \mathit{Insomnia} \wedge \mathit{Rash} \wedge \mathit{StiffNeck} \\ \mathit{Headache} \wedge \neg \mathit{Itch} \wedge \mathit{Toothache} \\ \vdots \end{gathered}$$

An easy way to represent such hypotheses (as suggested in the source code) is to use fixed-length integer lists for the symptoms while maintaining the above ordering and to encode "true" with a "1" value, "false" with "-1", and "missing"/"not checked" with "0", such that the above hypotheses would be encoded as

```
[0, 1, 0,-1, 0,-1, 0, 1, 1, 0]
[0, 0, 1, 0, 0, 0,-1, 0, 0, 1]
```

Q2: Input/output functionality (15%)

Implement a method in the `Agent` class to read the data files and convert them into internal samples encoded as integer arrays. Consult the Java API documentation[3] to determine appropriate auxiliary classes for this purpose (such as `FileInputStream`, `StringBuffer`, `StringTokenizer` and others in the `java.io` package). Also, you will need methods to convert hypotheses to and from text in order to generate and decode messages sent to/read from the blackboard.

Extend the `Start.main` method to enable generating a number of agents by passing a (variable-length) list of data file names (one for each agent) as arguments from the command line and the agreement percentage p (a real number between 0 and 1) to the system. Your code should allow for the following command line syntax:

```
java ke.Start <p> <file1> .... <fileN>
```

where `<p>` is the agreement percentage, and `<file1>` to `<fileN>` are n data files that will result in a system with n agents (note that although you are provided with five data files, the code you submit will be tested on agent populations of different sizes).

Implement a basic output behaviour for the system manager. It should at least provide the following facilities:

- Displaying (in the current text terminal) messages as they are posted to the blackboard
- Displaying open hypotheses and their status in terms of acceptance/rejection by others (note that you might encounter problems of size in the set of messages contained in the blackboard, try to summarise the information if necessary)

[2] Note that each of the five agents has a biased set of examples: For the first agent, it would seem that positive samples have symptoms that resemble a cold, the second might think that the patients suffer from some kind of allergy, the third might diagnose symptoms of stress. For the fourth agent, the virus produces symptoms similar to those of a food poisoning, and the last agent might infer that the virus resembles a dental problem or injury. Thus each of them has a totally different view of the new disease.

[3] Available online from `http://java.sun.com/j2se/1.5.0/docs/api/index.html` or on the DICE file system from `/usr/share/doc/java-1.5.0-sun-manual-1.5.0/api/index.html`

- Reporting termination of the system and the final result, together with a justification for choice of the solution (something along the lines of "hypothesis ... was agreed upon by agents 1, 2 and 3, i.e. 60% of the population")

Q3: Implementing the agents (35%)
The agent functionality is the core element of the consensus learning system. The task of designing and coding the agents can be broken down into a number of subtasks:

1. Basic communication behaviour: The basic flow of interaction between the agent and the blackboard needs to be specified and designed with great care. You have to think about
   (a) when and in which way to check for new proposals on the blackboard
   (b) how to monitor whether previously supported hypotheses have been withdrawn and a new hypothesis to support has to be sought for
   (c) when to propose, accept, reject hypotheses (this involves issues such as avoiding duplicate proposals, responding to open proposals in a timely fashion, etc.)
   (d) practical housekeeping issues (what is the model of the blackboard the agent has, does it keep local copies of portions of it, how are these updated, etc.)
2. Dealing with hypotheses: Using the procedures defined in Q1, you will first enable agents to load samples from a file. Then, you should provide methods to
   (a) Construct an initial hypothesis
   (b) Generalise over two hypotheses
   (c) Evaluate a hypothesis on locally stored samples
   (d) Pick the best known hypothesis in any situation

   This is the "local learning" part of the system and is essential for its overall performance. The idea is that the agent will pick a (potentially bounded) number of open hypotheses from the blackboard and combine them one by one with her current best hypothesis. Then, an evaluation of the "classification score" of each of these candidates will allow the agent to pick the current best hypothesis (and this may involve breaking ties between hypotheses that perform equally well.

   Note that none of these methods needs to be highly sophisticated or complex. Rather, your method and implementation of it should be tractable, easy to understand and intuitively reasonable.

Implement this functionality by extending the `Agent` and `Hypothesis` classes, and, if necessary, by creating additional ones.

Q4: Implementing the system manager (25%)
As described above, the system manager maintains the blackboard and tests for satisfaction of the termination criterion.

Modify the `SystemManager` class so that it realises the following functionality:

1. Constant monitoring of blackboard consistency and blackboard modification where necessary
   (a) deletion of duplicate proposals
   (b) deletion of withdrawn proposals and of accept/reject messages relating to a withdrawn proposal
   (c) deletion of messages with incorrect content (i.e. content that cannot be interpreted as a hypothesis)

2. Keeping a record a acceptance/rejection votes for each open hypothesis and updating this as new messages and testing frequently for satisfaction of the termination criterion.
3. Shutting down the system when a solution has been found (or determining that no solution can be found in certain situations, if you think this is possible (?)).

As with agents, carefully designing the control flow of the system manager is a key issue, as you typically want to avoid, for example, redundant processing cycles that check for consensus while no new messages are posted.

Q5: Evaluating the system (25%)
For this question, you will have to write an evaluation report (about 2 pages) that contains sections on the following topics:

1. Design & implementation of your system (summarises the work done for Q2-Q4)
   Describe key design issues, problems encountered, and how these were (not) solved, trade-offs made. Justify your decisions.
2. Performance of the system
   Conduct several simulations with your system and report on
   (a) the quality of the solutions obtained (by evaluating them on the union of all sample sets),
   (b) runtime behaviour of the system (how long simulations took, whether runtime problems occurred, etc.)
   (c) an evaluation of the advantages and disadvantages of a multiagent approach in this case (for example, you can test your hypothesis construction and generalisation methods on the union of all sample sets and compare their performance to the multiagent case).
3. "Playground" section (optional)
   Report on any other ideas you have had to improve your system, and which of these worked. Also, you might draw more general conclusions regarding the consensus learning approach and suggest potential improvements.

5 General implementation guidelines

Please follow the following guidelines for this programming assignment:

- **Comment your code**: be verbose about what the sources do, explain methods, variables and the control flow between individual components. It doesn't have to be `javadoc`, but at least comments inside the source files. No credit will be given for undocumented code.
- Do not access components internal to other agents or the system manager by writing your own methods for doing so. All agents and the system manager are allowed to "see" are the contents of the blackboard. Use of the "public" keyword in the `SystemManager` and `Agent` class source code indicates which methods other objects are supposed to use.
- Integrate your code into the `ke` package, so that it is easy to compile and run, and provide additional information for its use where necessary. It should always be possible to run the code using the command line format suggested above in the current working directory, i.e. without peculiar classpath settings etc.

As a final remark, this assignment allows many degrees of freedom and does not make prescriptive statements regarding all details of the implementation. Try to produce a coherently designed, functional, and reasonably elaborate piece of software, neither a masterpiece nor a almost-trivial implementation.