Learning in a Multi-Agent System

Sarah Mercer
H.M.G. Communications Centre
Hanslope Park
Hanslope
Milton Keynes, MK19 7BH, UK
email: sarahm@hmgcc.gov.uk

Sue Greenwood
Intelligent Systems Research Group
Department of Computing
Oxford Brookes University
Oxford, OX33 1HX, UK
email: sgreenwood@brookes.ac.uk

Abstract

This paper describes work undertaken to incorporate agents that can learn into an existing multi-agent system, designed to advise on good programming practice. The power of this approach is that the learning agents within the system can adapt domain knowledge to evolve programming standards over time. A prototype system has been developed which uses multiple agents to embody complementary learning algorithms to achieve a level of adaptability suitable for the domain. Test results are presented with an analysis and discussion.

Introduction

Previous work involved the development of a prototype system that provides the foundations for a system that will advise on good programming practices [Mercer and Greenwood, 2001]. A major objective of the research is to provide a system that is able to learn and adapt domain knowledge to evolve over time.

Knowledge of the source code to be reviewed is fragmented between a number of worker agents, detection agents then collate and reason over this knowledge to detect defects. Using machine-learning techniques these agents learn for themselves the defects that they are supposed to detect.

As machine learning is applied to real-world tasks, difficulties arise that do not occur in simpler textbook experiments. One such difficulty is selecting the best attributes to use for learning from a large set of candidate attributes. Ideally, a learning algorithm's generalization performance would improve when it is given the information supplied by additional attributes. Unfortunately, the opposite often occurs: additional attributes can interfere with other more useful attributes [Caruana and Freitag, 1].

To achieve the level of learning that would be desirable in such a system the following had to be considered. Given the number of attributes that could be extracted from the code, the number of ways each attribute could be interpreted and the number of different defects that the programmer may want the system to identify (concepts learnt) the *search space* would be too unwieldy for use on conventional hardware and operating systems.

Another complication is the requirement for the learning algorithm not to be too reliant on the structure / ordering of the training data as it is intended that the system will always learn and refine its knowledge, it must learn from a real-life perspective – which does not allow tailoring of input. This is directly related to another known difficulty of machine learning, in that the quality and quantity of training data is an important issue for any learning algorithm. Without extensive built in knowledge of a domain, a learning algorithm can be totally misled attempting to find patterns in noisy, insufficient, or bad data [Luger and Stubblefield, 1998]. Therefore the system needs to be aware of the domain.

To facilitate learning the search space needs to be reduced, this is achieved by attribute and feature selection. Attribute (or variable) selection refers to the problem of selecting input variables that are most predictive of a given outcome. Feature selection refers to the selection of an optimum subset of features derived from these input variables [Koller and Sahami, 1996].

There are many potential advantages of automating the attribute selection process. It allows the system designer freedom to identify as many useful attributes as possible. New attributes can be introduced more easily. Also in domains where the world changes, it allows the current best attributes to be those best suited to the current state of the world. Lastly, it allows the set of attributes used to change dynamically as the amount of training data changes [Caruana and Freitag, 1]. Much research has been and is being carried out on the problem of automated attribute / feature selection, [Raman and Ioerger, 2002], [Talavera, 1] and [Raman and Ioerger 2003].

Although manually selecting attributes is normally considered challenging and can lead to inferior selection this was not a problem in this case. The selection was achieved via domain knowledge; in a static domain (programming languages are well defined and static) the benefits gained from automatic attribute selection are no longer needed.

Attribute selection in this system is achieved by the use of learning spaces, where the attributes are selected based on domain knowledge. The search space is reduced to (currently) two basic learning spaces; variable and function. A set of attributes that describes a variable and the actions performed on / with that variable is defined as the variable learning space. A set of attributes that describes a function and the statements contained within is defined as the function learning space. Two further learning spaces are planned: module and project learning spaces, which will allow reasoning at higher scope levels (helpful for rules such as unused functions).

The decision to divide the attributes into these partitions was based on a study of the HMGCC global coding guidelines [HMGCC, 2001]; the majority of the guidelines were aimed at function or variable level detail (22 out of 35).

The *learning space*: a set of attributes retrieved from the worker agents, each learning space focuses on one aspect of the code, for example variables or functions.

Again some domain knowledge is needed to ensure that the attributes presented to the learning algorithm are carried out to achieve their full potential. Some attributes are made more specific and some more general. Within the variable learning space the value attribute denotes the value assigned to the variable when it was declared. However a more general attribute is also included which denotes whether the variable was initialised as a Boolean value.

The *learning plane*: the set of attributes from the learning space formatted and presented to the learning algorithm.

User directed learning

Due to the requirement that the system is always learning and should be adaptive to its user, it is important that the user has significant influence over its learning.

The system allows the user to input two different fragments of code that represent positive and negative examples of a rule. If this is the first time a particular user has submitted an example of the rule, the system will prompt the user for a description of the rule. This functionality allows the programmer to submit code to the system that was changed by a peer review, i.e. the before and after. The system is *only* given the two code fragments and the name of the "rule" it should create or affect.

The first task for the system is to identify the differences within the source code fragments; this is achieved using a similarity engine [Wetzel, 1]. The knowledge in the detection agent is extracted into the variable and function learning spaces; from here the relevant formatting is applied resulting in learning planes that contain sets of data from both the positive and negative examples.

The similarity engine is then able to match variables and functions between the positive and negative sets; to achieve this, weightings are applied to a subset of the learning space attributes. These weightings are based on domain knowledge: e.g. for variables: name, type, what-happens-to (action) and declaration-type are given the following weightings respectively: 4,5,1,4. Once the similarity engine has cross-matched variables and functions it is able to assert the positive and negative pairing(s) into the "long-term" memory of the agent as classified example(s) of the given rule.

A number of algorithms used to induce rules are now described and compared. To compare such algorithms a metric by which their performance is measured is needed.

To measure the performance of the algorithms over time, they are tested against training data sets of different sizes. Each algorithm is run three times; the average of the percentage of positives covered by "correct" clauses is calculated.

The test data used to teach these algorithms is not structured in anyway, structuring the data would, in most cases, reduce the number of examples needed to induce the correct rule, but it was important to maintain the unstructured nature of the data for these trials.

Learning algorithm #1

The learning algorithm used within the detection agent is based upon the Generic Separate and Conquer Algorithm (GSCA) presented by Nils J. Nilsson [Nilsson, 1998]. This particular algorithm was selected because of the power and flexibility that accrues from its simplicity.

The GSCA algorithm is able to induce rules of the form:

$$\alpha_1 \wedge \alpha_2 \wedge \ldots \alpha_n \supset \text{positive}$$

The GSCA algorithm was modified so that it could handle attributes that had continuous / discrete values (not just binary). It was also modified so that instead of producing multiple rules to cover all the positive examples, it produces a single rule that uses disjunctions to cover all the positive examples.

The modified GSCA algorithm (algorithm #1) is able to induce rules of the form:

$$\gamma_1 \vee \gamma_2 \;\; \vee \ldots \gamma_n \supset \text{positive}$$
where γ is of the form:
$$\alpha_1 \wedge \alpha_2 \wedge \ldots \alpha_n \supset \text{positive}$$

Which when applied to the learning data is able to detect defects simply by looking for learning data that are not covered by the rule.

Rule refinement

Due to the greedy nature of the learning algorithm, a refinement stage was included to remove any redundancy in the induced rule. This removed any attribute-value pairings from a clause whose coverage was a subset of another pairing's coverage in that clause. This procedure also removed any clauses whose coverage was a subset of another clause's coverage.

Observations

The algorithm was tested against a number of training data sets, which represented a number of concepts. The concepts fell into three distinct categories; those the algorithm was able to induce correctly within a reasonable amount of time, those it was unable to achieve, and those it was able to induce, but was unreliable or took a great deal of data to achieve.

The algorithm was reasonably adept at learning rules that comprised clauses containing one attribute-value pairing, for example *"init-before-use"* where at the function scope the rule induced was:

```
For all variables where (not…
        (action[0]='inited') or (action[0]='assigned') or (declaration-type='parameter')
) = 'init before use' defect.
```

The data for this concept is considered distinct as each element falls into one and only one clause.

The general trend was increasing over time, however the accuracy was not considered high enough for this system. It was apparent that the introduction of data that fell into "new" categories had a drastic effect on its performance. The 17th example included a parameter variable. The algorithm needed two further examples of a parameter variable to recover. Although it is reasonable that an algorithm needs a set of examples to be able to draw the necessary inference – it is not acceptable that an algorithm be so distracted by the addition of one new datum.

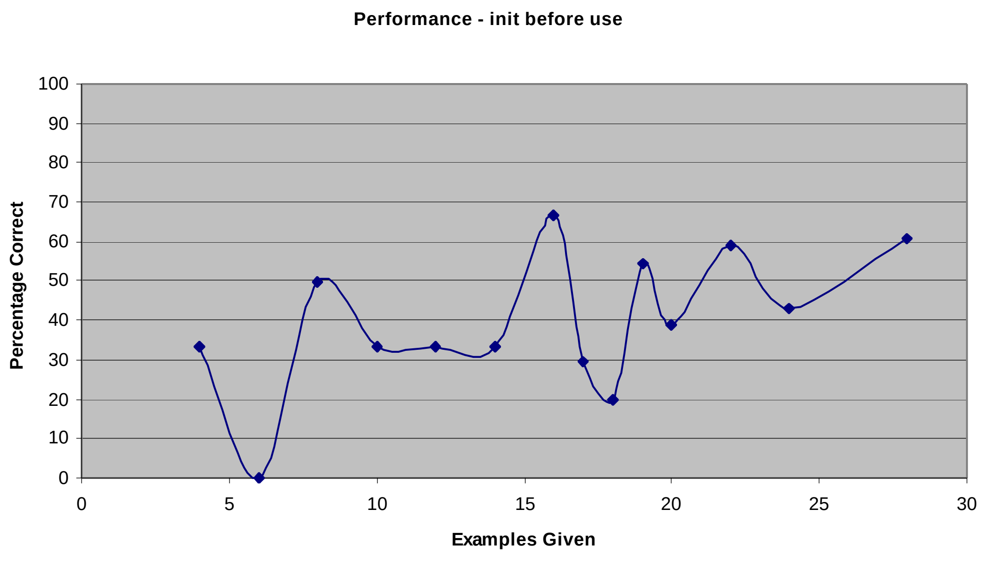

Figure 1: Performance of Algorithm #1 for the "init-before-use" concept.

For rules such as *"unused variables"* where the commonality is in the defects and not the positive examples – the induced rule was incorrect during the early stages, where the number of examples was low. The rule finally induced was:

For all variables where (not

(action[0]='used') or (action[0]='assigned') or (action[0]='tested') or (action[0]='inited')

) = 'unused variable' defect.

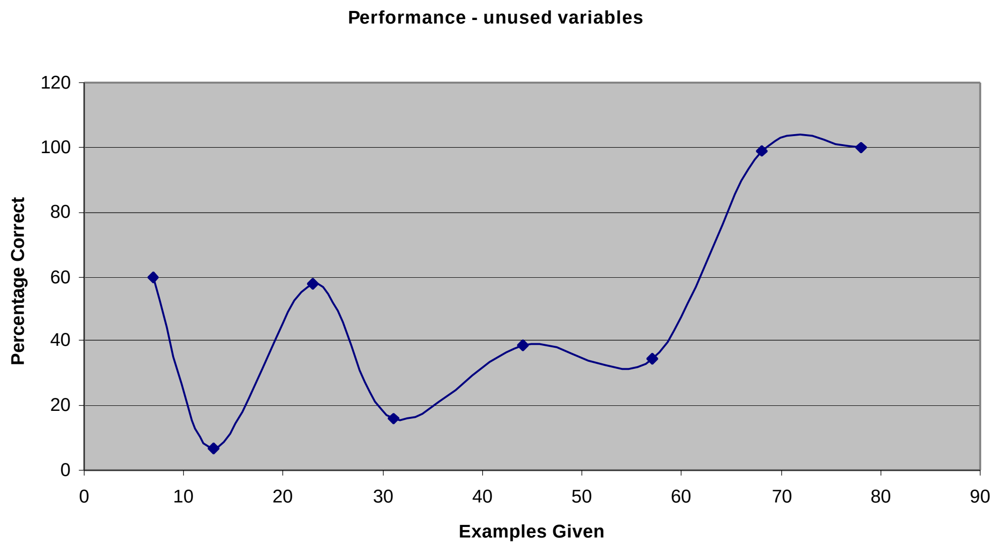

Figure 2: Performance of Algorithm #1 for the "unused-variables" concept.

Over time the induced rule was able to achieve 100% accuracy. In this case this is due to the fact that an unused variable's action is "nothing" or it is not used or initialised or tested or assigned. Therefore the complement of nothing could be determined, however the number of examples needed to achieve this was large, 60+. It is feasible to assume that there will be concepts whose commonality is in the defects that do not have a clear complement in the positive set, in which case the algorithm would be unable to induce an appropriate rule.

For rules such as *"variable naming"* where the rule induced should have been of the form:

For all variables where (not

((name[0]='n') and (type='int')) or (name[0]='l') and (type='long') …)

) = 'variable naming' defect.

It was unable to achieve a satisfactory conclusion.

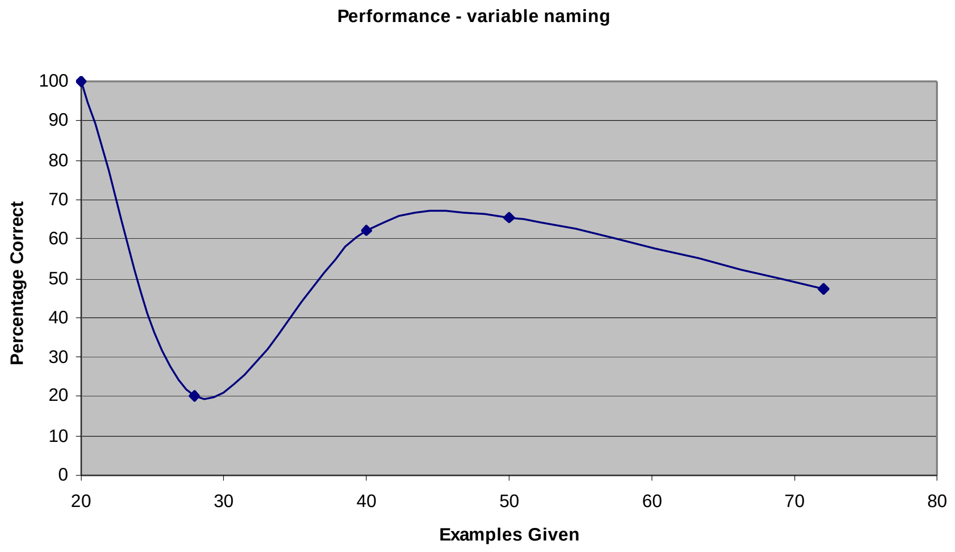

Figure 3: Performance of Algorithm #1 for "variable-naming" concept.

Over time the accuracy of the induced rule fell, as the noise in the data increased with the number of examples given.

The "problem" was identified as being a mixture of two facets of the learning algorithm when trying to learn this type of concept. Firstly the method of calculating the "best attribute": it used the net amount of positive examples that matched the attribute–value pairing (i.e. number of examples in the positive set that satisfy the pairing – number of examples in the negative set that satisfy the pairing) which after time resulted in the algorithm being swayed by noise, in this case the pairing name[4]='e' was selected as a best attribute after 28 examples were given, as this was common across variable names regardless of type. Secondly the way in which matches for each clause are removed from the learning set after each clause is completed. In this case as the first clause was wrong due to the calculation of the best attribute selection, the remaining data was a misrepresentation, which meant that further attribute-value pairing selection was not optimal, and often detrimental. This situation is unrecoverable and unavoidable with this algorithm.

Learning Algorithm #2: Improving accuracy

To address the concepts that algorithm #1 coped with but took a long time to arrive at the core of the rule, algorithm 1 was modified to have zero tolerance. Instead of allowing attributes that "mostly covered" the positive examples, it selected the attributes that "only covered" the positive examples.

This modified algorithm (algorithm #2) is able to induce rules of the form:

$\gamma_1 \vee \gamma_2 \ \vee \ldots \gamma_n \supset$ positive
where γ is of the form:
$\alpha_1 \wedge \alpha_2 \wedge \ldots \alpha_n \supset$ positive
where at most only one rule is produced.

It is possible and acceptable for this algorithm to fail to produce a rule. However, when a rule is produced by this algorithm it is generally more efficient and more generic than that provided by algorithm #1.

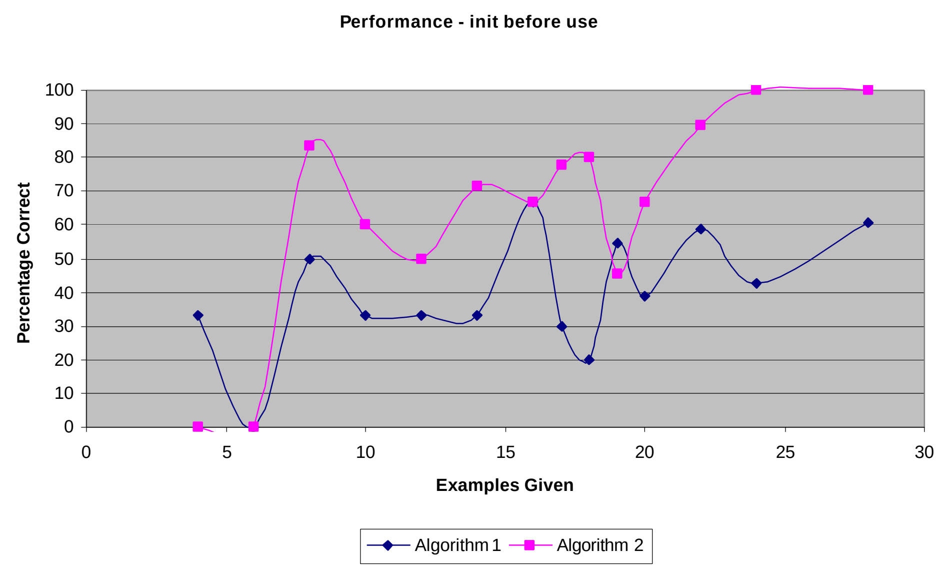

Figure 4: Performance of Algorithms #1 and #2 for "init-before-use".

As can be seen the second learning algorithm out-performed the first quite significantly for concepts such as init-before-use (where the data is distinct), achieving 100% accuracy after 24 examples.

Learning Algorithm #3: Improving coverage

To address the concepts that algorithm #1 struggled with, the algorithm was reversed, so instead of looking at what the good code examples had in common, it looked at what the defects had in common.

Modifications to algorithm #1 meant that learning algorithm #3 is able to induce rules of the form:

$\gamma_1 \vee \gamma_2 \quad \vee \ldots \gamma_n \supset$ negative
where γ is of the form:
$\alpha_1 \wedge \alpha_2 \wedge \ldots \alpha_n \supset$ negative

Which when applied to the test data is able to detect defects simply by looking for data that are covered by the rule.

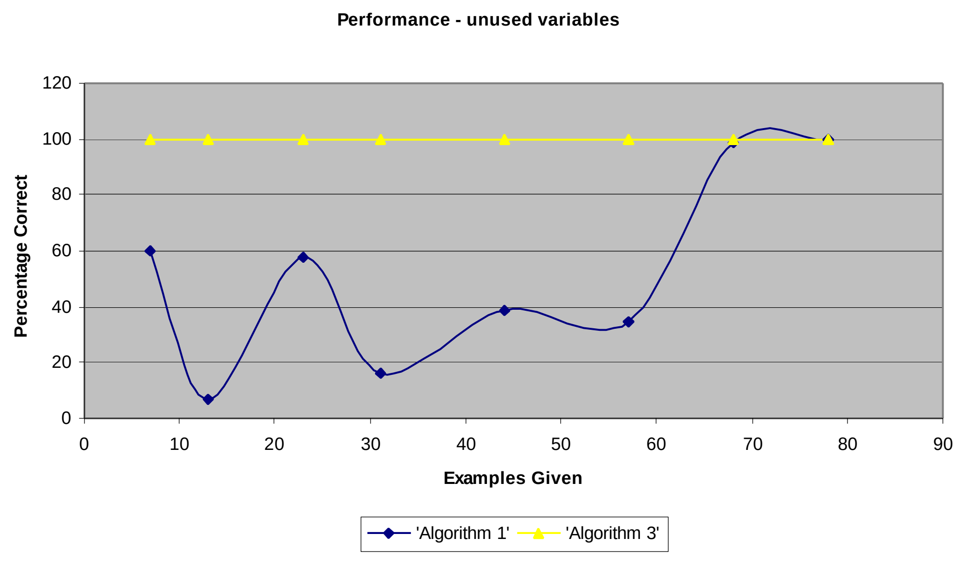

Figure 5: Performance of Algorithms #1 and #3 for "unused-variables".

For rules such as unused-variables learning algorithm #3 performed well, in the specific case of unused-variables it was remarkably accurate (able to achieve 100% accuracy with only a few examples provided) due to the simplistic nature of the concept to be induced: an unused variable is one which is has an empty action list.

Learning Algorithm #5: Improving noise tolerance

To combat the rules where over time too much noise was introduced due to the number of examples that it became detrimental to the rule induction algorithm, an improved method of calculating the "best attribute" was required.

For example:

When given the same training data, algorithm #1 is unable to distinguish between name[0]=n and indirection=0 as the first attribute-value pairing to select, as both have a net performance of 16.

```
name0 n     P=38, p=17, N=32, n=1, diff=16.
indr  0     P=38, p=30, N=32, n=14, diff=16.
```

Where P is number of uncovered positive examples, N is number of uncovered negative examples, p is the number of examples in P that meet attribute-value pairing and n is the number of examples in N that meet attribute-value pairing.

Using the following formula for calculating gain taken from Propositional FOIL [Quinlan, 1990] Gain Metric.

$$Gain = p * (\log2(p/(p+n)) - \log2(P/(P+N)))$$

In the same situation, learning algorithm #5 correctly selects name[0]=n.

```
name0 n     P=38, p=17, N=32, n=1, diff=16, gain=13.581185.
indr  0     P=38, p=30, N=32, n=14, diff=16, gain=9.864432.
```

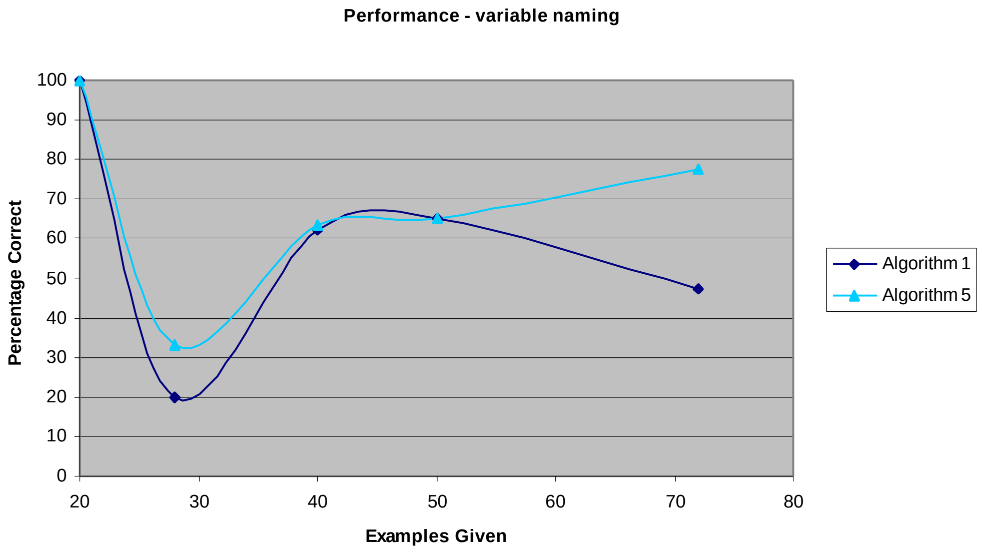

Figure 6: Performance of Algorithms #1 and #5 for "variable-naming".

As can be seen the performance of algorithm #5 is more robust over time, and is able to tolerate noise far better than algorithm #1.

Conclusions and Observations

The algorithms were compared against each other for an example of each of the three rule types:

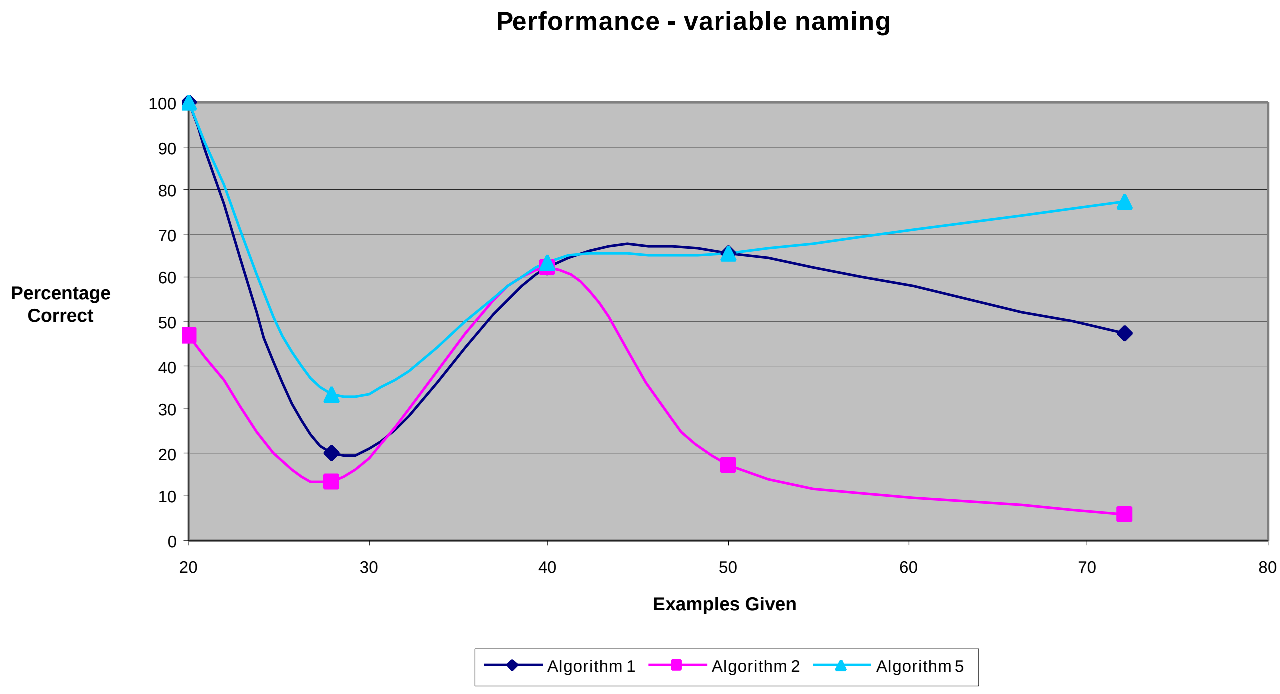

Figure 7: Comparison of Algorithms #1, #2 and #5 for "variable-naming".

For rules that are susceptible to noise like variable naming, Algorithm #5 gave the best performance over time.

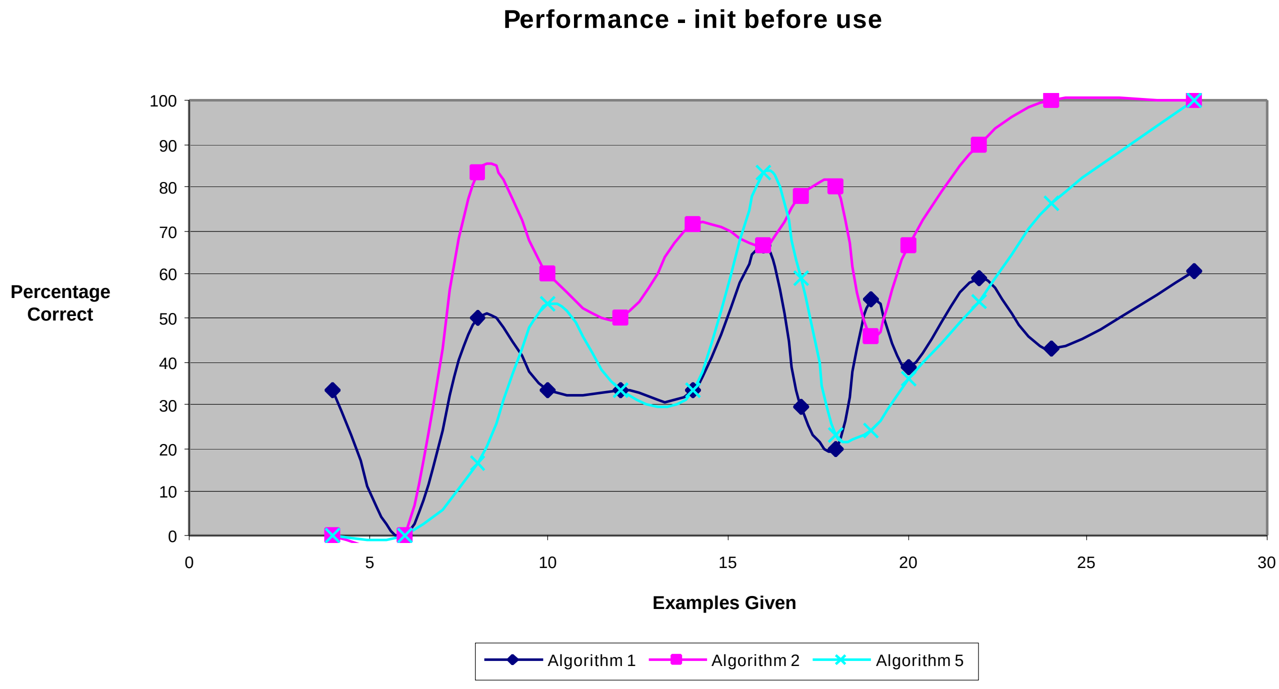

Figure 8: Comparison of Algorithms #1, #2 and #5 for "init-before-use".

For rules that have distinct data, Algorithm #2 gave the best performance from the start, however over time Algorithm #5 appeared to be able to achieve the same accuracy.

For rules whose data has commonality in the negative examples and not the positive ones, Algorithm #3 gave the best performance (Figure 9), although it is important to note that the rules that match this criterion have distinct data also.

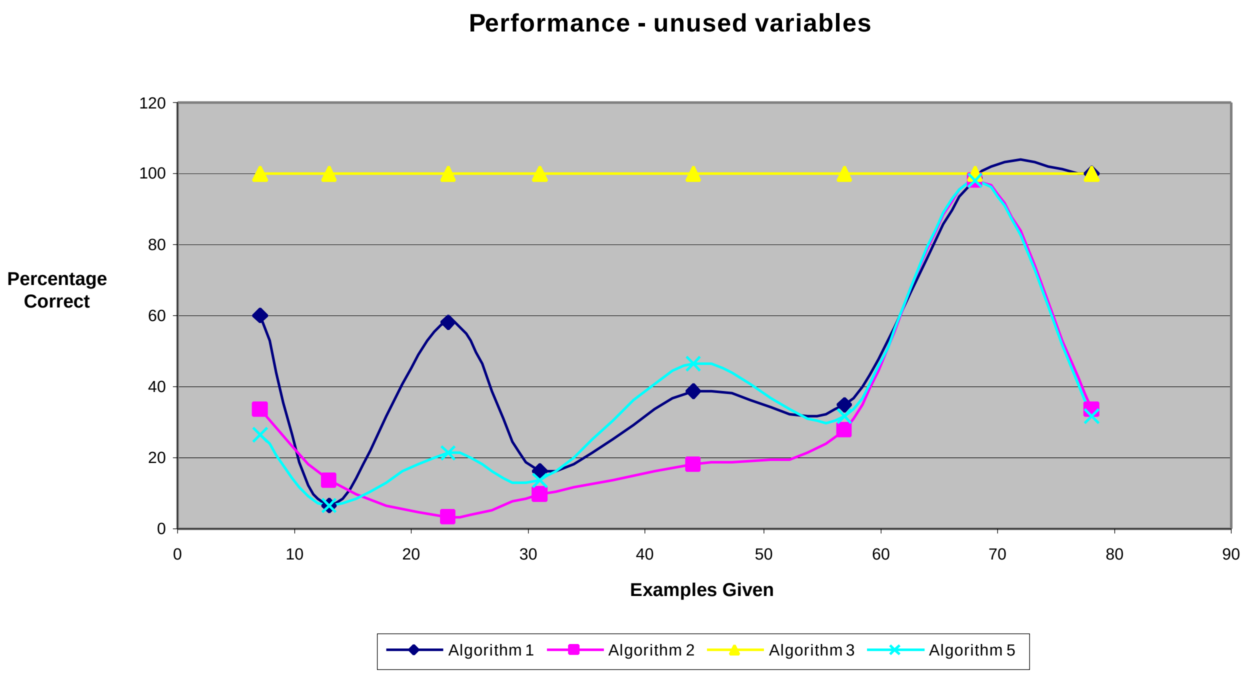

Figure 9: Comparison of Algorithms #1, #2, #3 and #5 for "unused-variables".

Expansion to include multiple learning algorithms as multiple detection agents

It was apparent that maintaining learning algorithms #2, #3 and #5 in the system would improve performance and usability. Instead of attempting to predict which algorithm would be best suited for the training data or combining them into one larger algorithm that exhaustedly tried to find the best rule, it was decided that multiple detection agents could be deployed each containing a different learning algorithm. This would allow the learning algorithms to stay separate (and therefore retain their simple / flexible nature) whilst providing the best coverage.

All the detection agents use the same long-term memory data (positive and negative examples collected over time). They also all attempt to learn the same rules. When given a new code snippet to learn from (positive and negative examples), each agent is notified of the new examples and relearns the concept from the new set.

When directed by the user, the system is able to review a module of code. The agents reason over the code and attempt to detect the defects; this is carried out in parallel by each of the detection agents, they collect the information from the worker agents independently and use their own rules to pin-point errors in the code. These defects are then put forward with a confidence rating (of their own rule) to a controlling agent that is able to select the most appropriate for feedback to the user. There are a number of factors for determining system confidence in a defect, as discussed in the next section. The user is able to contradict or confirm a specific defect or confirm a review, this reinforcement / refinement by the user is necessary to ensure the knowledge and learning of the detection agents is "on track", and has a bearing on the confidence of the system.

Weightings

Before the controlling agent notifies the user of a defect, it calculates the confidence it has in that defect. There are a number of factors that determine this confidence rating:

- The originating agent's confidence:
  - The confidence the agent has in the defect-detecting rule.
  - The confidence the agent has in the clause that triggered the defect.

- The controlling agent's confidence:
    - The highest agent confidence value for this defect.
    - The highest confidence the controlling agent has in an agent that reports this defect.

The originating agent has no concept of time, and therefore is unable to provide a confidence based on past performance. However the controlling agent maintains a history and is able to alter confidence in an agent-defect pairing based on user feedback. This mechanism allows the system to be made aware of which agents are good for which defects, i.e., which learning algorithm is best suited for the to-be-induced concept.

Results of testing the system that comprises detection agents that implement learning algorithms #1, #2, #3 and #5 are now presented. The input to the system is real life – project based code, which has not been seen before by the agents. However, it has been written by the same programmer as the training data. The full feedback is given to the system for each defect found – to see over time how the confidence mechanism adapts.

Observations

When given further examples of the three test concepts, it is initially apparent that the noise tolerant algorithm implemented within DetAgent5 is alone sufficient to provide the learning mechanics of the system. It out performed the other algorithms not just on the type of concept it was developed for (variable-naming) but also for init-before-use.

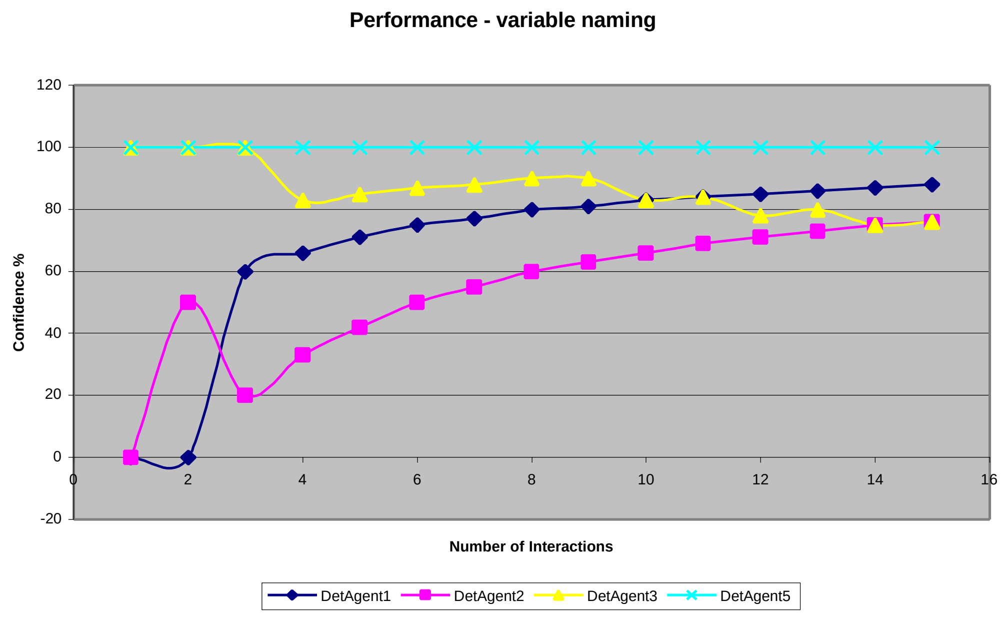

Figure 10: Performance of all algorithms when user feedback is given for "variable-naming".

When user feedback was taken into account for the variable-naming concept, over time the controlling agent's confidence grew in all the agents except for DetAgent3 whose defects were often contradicted. DetAgent5 was never contradicted, and was able to remain 100% accurate for all of the examples given.

When compared to all of the other agents, when learning "init-before-use" (Figure 11), DetAgent5 was again the best performer over time, with the highest confidence rating after 10+ examples. DetAgent2 that implemented the algorithm developed for this type of concept also performed well.

For the concept unused-variables (Figure 12), DetAgent1 and DetAgent3 were able to maintain 100% accuracy where DetaAgent2 and DetAgent5 initially faltered. Given the initial training set it only took a few (3-5) interactions for each agent to induce the final rule.

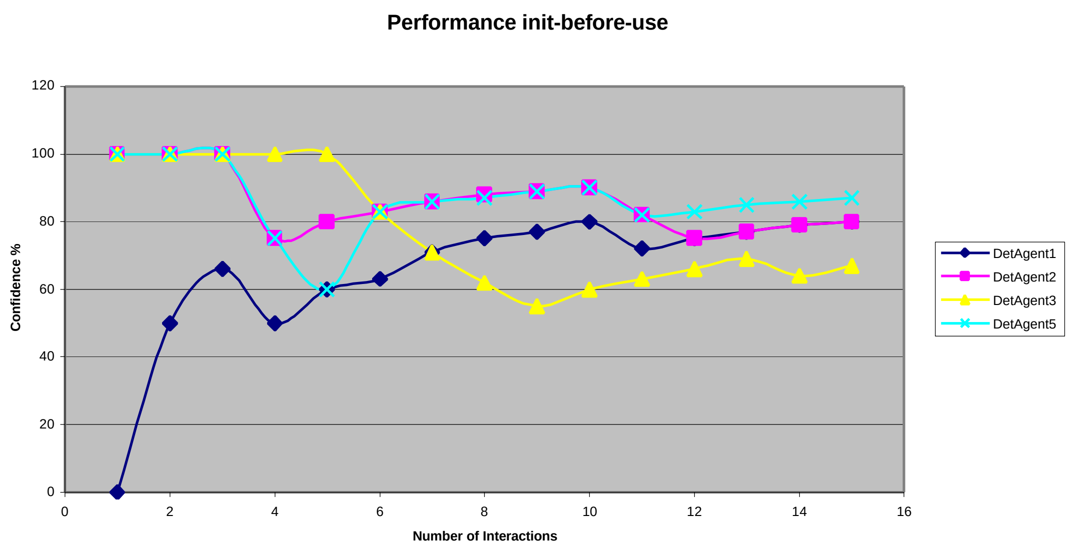

Figure 11: Performance of all algorithms when user feedback is given for "init-before-use".

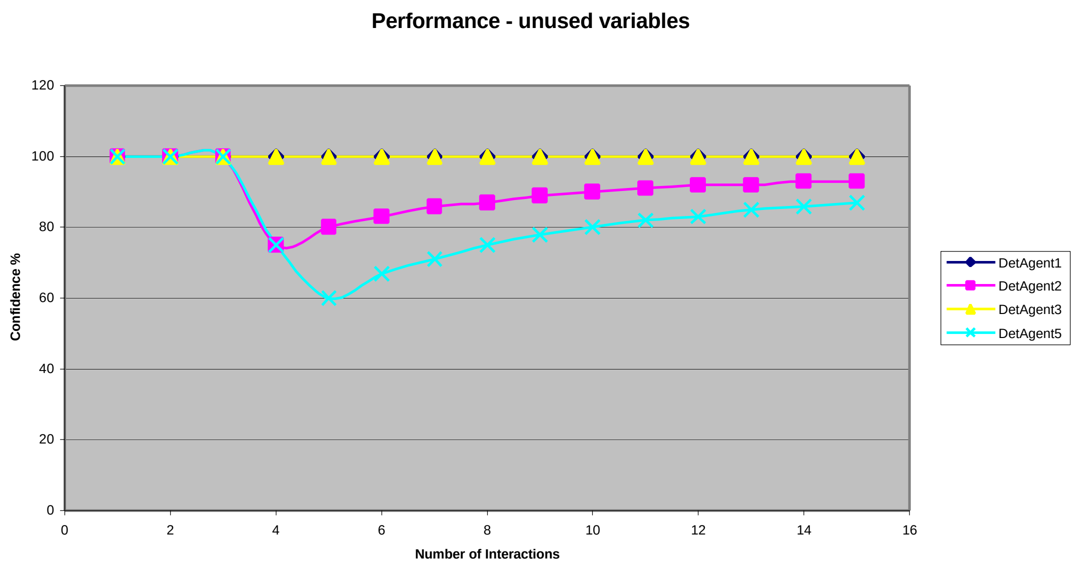

Figure 12: Performance of all algorithms when user feedback is given for "unused-variables".

Problems faced during testing included the adequacy of the test data and the interpretation of the systems results:

- Inadequacy of test data – the source code was in pre review state, this meant that certain rules, like variable naming had not been strictly adhered to, this gave the system a good set of positive and negative examples to work with. However in the case of init-before-use, which is a concept aimed at bug-detection and not stylistic issues, there were very few examples.
- Interpretation of system results – there were a number of unexpected challenges to the assumptions made before testing and what the tests highlighted. For example the concept induced from the training data for 'unused variables' was not actually correct, and classified variable as used, if they were only initialised or assigned. When in the context of real code this was more apparent as a defect. Also the concept initially induced for init-before-use was incorrect as it triggered on unused variables.

Conclusions

The results from testing the four algorithms earlier in the paper indicate that to allow the system a fighting chance of inducing a rule that fits with relatively few examples, the more direct algorithms are needed (the ones that work on net coverage). Then over time algorithms that perform better against noise are needed, to ensure that learning stays "on track".

The results from testing the algorithms with user feedback was encouraging as it showed a marked improvement on the accuracy of the system.

It has been theorised that a solution to a global problem can emerge from the collective activities of independent agents, research in this area has demonstrated that highly goal-directed, robust, nearly optimal behaviours can arise from the interactions of simple individual agents [Luger and Stubblefield, 1998]. Luger goes further to say that full intelligence can and does arise from the interactions of many simple, individual, embodied agents. Although this development is on a much smaller scale, the basic principle still applies, that having a group of simple (but non trivial) agents working together, with the user, a level of intelligence can emerge that betters the task at hand.

The next phase of development will include an improved rule refinement mechanism that is able to remove clauses or attribute-value pairs if they are covered by one **or more** existing elements. This may help to reverse, or at least lessen, the effects of incorrect best attribute-name pair selection.

The existing learning planes will also be improved, in particular for "init-before-use" to be helpful to the programmer it needs to be based on more detailed information than just the generalised actions of used, tested, inited and assigned. In some cases being 'used' as a parameter into a function results in the initialisation of the variable, i.e. memset for a character array.

Further learning spaces and planes will be added to the system to allow reasoning over code at a wider scope – module and project level guidelines can then be implemented.

References:

[Mercer and Greenwood, 2001] Sarah Mercer and Sue Greenwood. *A Multi-agent Architecture for Knowledge Sharing*. Proceedings of the Sixteenth European Meeting on Cybernetics and Systems Research, 2001.

[Caruana and Freitag, 1] Rich Caruana and Dayne Freitag. *Greedy Attribute Selection*. Carnegie Mellon University, USA.

[Luger and Stubblefield, 1998] George F. Luger and William A. Stubblefield. *Artificial Intelligence*. Addison-Wesley, 1998.

[Koller and Sahami, 1996] D Koller and M Sahami. *Toward Optimal Feature Selection*. Proceedings of the Thirteenth International Conference. Morgan Kaufmann, 1996.

[Raman and Ioerger, 2002] Baranidharan Raman and Thomas R. Ioerger. *Instance Based Filter for Feature Selection*. Texas A&M University, USA.

[Talavera, 1] Luis Talavera. *Dynamic Local Feature Selection in Incremental Clustering*. Universitat Politecnica de Catalunya, Spain.

[Raman and Ioerger 2003] Baranidharan Raman and Thomas R. Ioerger. *Enhancing Learning using Feature and Example selection*. Texas A&M University, USA.

[HMGCC, 2001] Sarah Mercer, HMGCC Global Coding Standards, HMGCC, 2001.

[Wetzel, 1] Baylor Wetzel. *Project: Selection Engine*. http:// sourceforge.net/projects/selectionengine/.

[Nilsson, 1998] *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.

[Quinlan, 1990] J. R. Quinlan. *Learning Logical Definitions from Relations*. Machine Learning. Morgan Kaufmann, 1990.