COBBIT—A Control Procedure for COBWEB in the Presence of Concept Drift

Fredrik Kilander and Carl Gustaf Jansson

Department of Computer and Systems Sciences
Royal Institute of Technology and Stockholm University
Sweden

Abstract. This paper is concerned with the robustness of concept formation systems in the presence of concept drift. By concept drift is meant that the intension of a concept is not stable during the period of learning, a restriction which is otherwise often imposed. The work is based upon the architecture of COBWEB, an incremental, probabilistic conceptual clustering system. When incrementally and sequentially exposed to the extensions of a set of concepts, COBWEB retains all examples, disregards the age of a concept and may create different conceptual structures dependent on the order of examples. These three characteristics make COBWEB sensitive to the effects of concept drift. Six mechanisms that can detect concept drift and adjust the conceptual structure are proposed. A variant of one of these mechanisms: dynamic deletion of old examples, is implemented in a modified COBWEB system called COBBIT. The relative performance of COBWEB and COBBIT in the presence of concept drift is evaluated. In the experiment the error index, i.e. the average of the ability to predict each attribute is used as the major instrument. The experiment is performed in a synthetical domain and indicates that COBBIT regain performance faster after a discrete concept shift.

1 Introduction

With a given data set, incremental or batch learning is a matter of taste. Because the data is given, it is finite and therefore available for scrutiny by any available mode of preparation, clustering and post-processing. Such is the state of much scientific data: collected, stored, processed and analyzed at length.

Reality throws a wild and evolving environment at us; dependencies between observable and non-observable features change over time. The ability to accurately categorize and associate phenomenon is highly valued in both people and machines, but there is a danger in the complacency that follows seemingly unshakable competence. Knowledge that is not regularly confirmed runs the risk of becoming obsolete when concepts surreptitiously drift, nurturing a creeping knowledge rot. A carefully captured flow of experience may be invalidated by a sudden, single blow of altered environmental conditions.

The problem of concept drift in the context of this paper is characterized by a change in the environment observed by an incremental machine learning system. The machine collects a number of samples from the environment and builds a

tree over the conditional probabilities that relate smaller groups of observations. Each such group represents a concept.

Concept drift occurs when the environment change. The conditional probabilities reflected by the new observations change too and are no longer accurately represented by the concepts in the machine learning system. The occurence counts on which the probability estimates are based, include observations that the environment no longer can provide.

Programs and systems that continuously updates their view of the world fall into two groups. In the first are those for which the tracking of concept drift follows involuntarily from the basic architecture of the system. In the second are the programs designed to behave like concept trackers.

Common to the former kind of systems is their limited amount of storage for concepts or learned structures. Their ability to stay alert with recent trends is a side-effect of old knowledge being deleted. The deletion occurs simply because a section of storage used by old knowledge, is claimed for more recent data. Examples of such architectures are neural network models and genetic algorithms.

The group of systems designed to track concept drift include STAGGER [20, 19], FLORA [13], FLORA3 [21], FAVORIT [12] and RL [3]. These supervised learning systems focus on the ability to deal with training sequences involving concept drift. The FLORA system, for instance, employs a queue of recent training examples. When the oldest example leave the queue, FLORA updates its set of induced classification rules to be consistent with the contents of the queue. Discarded rules are kept suspended, awaiting future rehabilitation.

A shared behaviour of the systems that track concepts intentionally is that they maintain generalizations under a notion of recency and non-monotonicity. Objects that once supported general structures may be withdrawn, and the corresponding generalization diminished or made impotent. The difference these systems display from those that track concepts inadvertently, is that the decision to delete or inactivate old knowledge is conscious.

This paper is concerned with concept drift in unsupervised learning and is based on Douglas Fisher's COBWEB [6], an incremental, probabilistic, conceptual clustering system. COBWEB is designed to work under a condition of concept constancy, just as most other machine learning systems. The presentation sequences created by the temporal ordering of examples while allowing a change of concepts over time (concept drift), coincide with those orderings that are least suitable for COBWEB. The concept hierarchy becomes skewed, in that a concept node is found to be subordinate to the node that optimally is its peer. For a recent treatment of ordering effects in COBWEB, see Fisher, Xu and Zard [5].

The experiments with COBWEB were performed on COBBIT, Kilander's implementation of COBWEB in C. The COBBIT system is an extension to the control system in COBWEB. COBBIT uses a queue of training instances just as FLORA3 does and dynamically alters the size of the queue depending on its performance. This behaviour is intended to remove old knowledge from the hierarchy, leaving room for new, updated concepts.

2 Noise and Concept Drift

The term concept drift is more easily defined in terms of incremental, supervised learning. The learning situation there is equipped with a strong source of feedback on performance; the class label. The learning program creates a concept for each class, separating instances. If the program receives an object with a class label that is not what the concept assigns, the contradiction can be attributed either to noise (the label is wrong) or misconception of the concept (the induced concept is wrong).

If previous training instances are stored by the system (or implicitly trusted through the concepts they generate) it is possible to find an old training object which is identical to the disturbing one, save for the class label. Noise must then be the problem, and the system can deal with it in several ways, for instance by rejecting the contradicting instances. If the source of the misclassification cannot be established the system's only recourse is to modify its concept description, in the hope that the concept definition is improved.

The assumption that underlies the above incremental learning method is that the domain which is approximated by the concepts in the learning systems is fixed; that all examples are equally important regardless of when they were observed. The concepts in the domain are assumed to be eternal and stable. Removing this assumption allows for the possibility that a recent observation appears to contradict an earlier one, but the early observation can no longer be made and concepts built upon it are indeed wrong. Concept drift is present.

Removing the constancy assumption introduces another restriction; if any distinction in temporal terms is to be meaningful, the examples must be presented in the order they were labelled.

In supervised learning concept drift can be seen as a change in the process that labels examples. In unsupervised learning (clustering) there is of course no explicit class labelling. However, there is a similar process which can be thought of as the assignment of attribute values to objects. Only certain combinations of attribute values can be observered at any one time, and it is this set of dependencies between attributes and their values that a clustering systems attempts to approximate by forming groups of objects.

3 Cobweb

COBWEB is an incremental, probabilistic, conceptual clustering system [6]. COBWEB is discussed both as an incremental learning system as well as a potential model for basic level effects in an indexed memory. This work concentrates on the former aspect of COBWEB's capabilities.

From a stream of object instances, COBWEB creates and maintains a hierarchy, each level of which partitions the objects in an optimal way. Instances form the leafs of the hierarchy, the generalizations above form the concepts. The incremental nature of the system allows it to learn and perform at the same time. As expected, performance is limited until a representative sample of the

domain has been acquired, but the prediction of a missing piece of information may be attempted from the start.

COBWEB is suitable as a normative system because it:

1. is well known
2. is easy to understand
3. is fairly easy to implement
4. is a foundation for other systems
5. is susceptible to concept drift, and
6. does not require parameters.

The closest rival to COBWEB is CLASSIT [7] but CLASSIT only accepts linear attributes. Other conceptual clustering systems to choose from are not lacking; UNIMEM by Michael Lebowitz [14, 15], WITT by Hanson and Bauer [9, 10], INC by Hadzikadic and Yun [8], OPUS by Bernd Nordhausen [18] and the works of Ryszard Michalski and Robert Stepp with the CLUSTER systems [16, 17].

UNIMEM is a forerunner of COBWEB but requires parameters. INC is a system very similar to COBWEB but it also needs parameters. WITT, OPUS and CLUSTER are batch clusterers. WITT can be set to learn incrementally using a series of smaller batches, but it still involves too many parameters.

Recent COBWEB descendants that should be mentioned but has escaped evaluation due to time-constraints are: ITERATE by Biswas et al. [2] and COBWEB$_R$, a system by Allen and Thompson [1].

The emerging qualities of COBWEB are that it is *incremental* and that it *lacks parameters* which constrains an otherwise unwieldy number of examination and experiment dimensions. The fact that COBBIT reintroduces a number of parameters is lamentable but impossible to avoid. Hopefully further research will be able to advise automatic settings for them.

The following problems appear in various degrees of severity when COBWEB is used on material which is subject to concept drift.

1. COBWEB does not distinguish between new and old training examples. New features must therefore appear in ever larger numbers if they are to replace their predecessors.
2. COBWEB retains every training example. Learning must therefore eventually stop at the limits of the computational or practical resources.
3. COBWEB may create different concept hierarchies depending on the input ordering. The input ordering found in material affected by concept drift is among the worst.

4 Cobbit

COBBIT is built around an implementation of COBWEB which for the larger part duplicates the original. No significant extensions have been applied to the COBWEB model of storing nodes and concepts in the concept hierarchy, the classification or prediction algorithms.

4.1 Detecting Concept Drift

In applications where COBWEB is used to make predictions about unseen objects one would certainly want the prediction mechanism to pay greater attention to recent information than to old and early one (unless a static domain is ensured). Mechanisms capable of detecting changes in concept stability are interesting because they may serve as triggers of automatic actions; relearning or adjustment of the internal structure.

Here follows six algorithmic devices to survey changes in the COBWEB hierarchy, two of which are strategies to deactivate undesirable objects:

1. **Trends in past preference counters.** For each attribute, past preference [4] will be established at the node where the most common value of the attribute is the majority value averaged over all examples beneath the node. If extra disjunctive structure is imposed on the category (i.e. novel data coming from changes) the level of past preference is expected to drop to a lower node, or even be split among several nodes. This can be monitored by following the development of the *correct-at-node* and *correct-at-descendant* counts. A major change in majority values among nodes beneath the node under surveillance can be suspected if the latter counter is increasing faster than the former. An immediate application of this is with the prediction mechanism which upon detecting this kind of phenomenon should ignore a higher *correct-at-node* count in favour of a prediction further down.
2. **Operators used to classify a new example.** In a well established hierarchy most of the domain can be expected to have been observed. The presence of a new example can therefore signal either change or a very rare event. The example will be placed by itself at some level as its own category. The level (relative the size of the hierarchy) is an indication of novelty. This only works for true novelty, a cyclic reoccurrence will not be noticed.
3. **Update frequency and its correlation to expected relative frequency.** If the domain is changing one can expect differences in the relative frequencies between two time periods. Although a possible case of the gambler's fallacy, one may expect certain observations to appear with a rate that corresponds to their previous exposure. If this frequency distribution is changing more than is likely to be attributed to random variation, one can suspect that it is an effect of an altered domain.
4. **Continuous monitoring of performance.** As each example is presented to COBBIT, it attempts to predict each and every attribute that has a known value. The percentage of correctly predicted attributes is output as the current *performance index*. The trend of this index allows for corrective action as soon as a drop in performance is observed.
5. **Monotonic deletion.** Its a simple matter to modify COBWEB and obtain the kind of short term memory used in FLORA. A first-in-first-out (FIFO) queue facilitates subtraction of each example after a suitable time.
6. **Dynamic deletion.** The insensitive nature of the previous approach can be cushioned by using one or more of the detection mechanisms outlined

above. The idea is that a concept suspected to be no longer present in the domain is to be removed. The motive for removing the object is in this case based on more than simply the time spent within the system, and there is an ambition to retain objects that are beneficial.

Deletion of an object can be achieved by marking it as *inactive*, or subtracting it from the hierarchy. Removal from the central concept hierarchy does not necessarily imply ejection from the system; several schemes of retaining objects backstage are plausible.

4.2 Cobweb Modified

This section deals with the consequences of equipping COBWEB with a version of the sixth device, dynamic deletion of old examples. This is implemented in COBBIT with a queue of examples and continuous monitoring of performance (device 4). Together they uphold an interval of examples; when an example leaves the queue it is subtracted from the hierarchy. The performance index is used to determine the size of the queue (within certain boundaries).

It can be expected (assuming a fair description language and input ordering) that a basic hierarchy is established and that further input either confirms what is already learned or is one of the following two cases: a new example that adds knowledge, because there is still more to learn or a new example that replaces old knowledge, because the domain is evolving.

Both cases contain new knowledge, and both cases should be added to the knowledge structure. So the problem is really one of identifying *old* knowledge and data structures. The simplest way to do this is using a queue, which assumes that the oldest example no longer can be trusted and ejects it, using a FIFO-strategy for deletion. If the features represented by a deleted node are still active in the domain, then they will manifest themselves again soon, if not already present in the queue.

Another approach is to remove concepts that have not been referenced or reinforced by the domain within a *reasonable* time. This is the least-recently-used (LRU) strategy for selecting a node to be deleted. However, under LRU the property of the queue (and the hierarchy) being a proper statistical sample of the domain is lost. LRU favours rare but regularly recurring events and disfavours short manifestations.

COBBIT's modification to COBWEB resides in the *control system*, the way learning, predictive performance and training are combined to a complete system. The following implements have been produced:

1. A procedure to subtract nodes from the COBWEB concept hierarchy.
2. A function that samples the the predictive performance of the concept hierarchy and COBWEB's prediction algorithm.
3. A function that provides a mapping from performance (as measured by the function in item 2) to a target size of the COBWEB concept hierarchy. The size is expressed as the number of training examples found at leaf level of the concept hierarchy.

4. A FIFO queue for training examples.
5. An extended control algorithm. The COBWEB standard control loop (*read-learn*, see table 2) has been extended to *read-evaluate-learn-trim* by using the queue in item 4.

Table 1. The Extract_Object procedure.

```
Extract_Object(object)
begin
    P = ParentOf(object).
    While P ≠ NULL do
    begin
          P = P − object
          R = ParentOf(P)
          If P is empty then delete P
          P = R
    end
end
```

Table 2. Control loop in COBWEB.

```
While not eof do
begin
   object = the next object from the input.
   If object is a training instance then
      root = Cobweb(object, root)
   else
      predict missing values and report.
end
```

4.3 Cobbit's Control Algorithm

The algorithm used to classify new examples in COBBIT is the same as in COBWEB. The change is introduced in the next upper level of the control hierarchy.

A new procedure that extracts objects from the hierarchy is shown in table 1. An *empty* node occurs when all its children have been subtracted from under

Table 3. Control loop in COBBIT.

```
While not eof do
begin
   object = the next instance from the input.
   IF object is a training instance then
   begin
      Error = PredictionError(object).
      root = Cobweb(object, root).
      Queue = Queue + object.
      MaxQSize = ((1 − Error) * (u − l)) + l.
      IF Length(Queue) > MaxQSize then
      begin
         N = 1 + (Length(Queue) − MaxQSize) / 4.
         While N > 0 do
         begin
            Extract_Object(Head(Queue)).
            Queue = Queue − Head(Queue).
            N = N − 1.
         end
      end
   end
   else
      predict missing values and report.
end
```

a generalization. All counters in the empty node are zero, but it still occupies space in the concept hierarchy and is therefore deleted from the tree.

The idealized control loop of COBBIT is described in table 3. The parameters u and l control the upper and lower bounds on the number of elements in the queue at any time. The queue can only grow one instance at a time, but it can shrink faster. The division by 4 when determining the number of objects to be ejected (N), dampens the effect of noisy examples which otherwise may cause an overly rapid depletion of objects. The quantity 4 was chosen arbitrarily—the effect of other values is not evaluated. Notice also the call to *Cobweb()*, COBWEB's classification procedure.

5 Experiments

Kibler and Langley [11] suggests several ways to measure performance. One is the general measure of *"the ability to predict a missing attribute's value, averaged across all attributes"*. For incremental systems, they suggest that learning is turned off every nth instance for testing against a test set of examples. Alternatively, every instance is treated as both a test and training object. This is

the major instrument of measurement in these experiments. For each training example an error index is calculated thus:

$$1 - \frac{\text{nof correctly predicted attributes}}{\text{nof attributes}} .$$

This averages the ability to predict each attribute, and uses every example for both testing and training.

Kibler and Langley continues to remark that the learning curve (resulting from such a performance measure) can be informative, but that the information can be condensed into the asymptotic performance and the number of examples required to reach it. This fails to be immediately applicable under concept drift as the asymptotic target constantly is redefined.

It should be pointed out the concept trees generated by COBWEB are not evaluated in terms of cluster analysis. The interest is focussed on the system's ability to predict missing attribute values.

5.1 Experiment Design

The synthetic domain which provides the training examples is designed to condition its concept drift on a particular attribute, the *cue* attribute. When this attribute is active the language is complete and concept drift not present (although ordering effects remain). Holding the value of the cue attribute fixed, it can no longer be used to predict the values of the other attributes and concept drift can be simulated.

The first experiment shows the effect on COBWEB when the cue attribute is locked. There is no concept drift in the data—the next state of the domain is chosen at random. Even so, one could view this as a demonstration of a situation where the rate of concept drift is beyond the sampling rate of the input collectors.

The second experiment intends to show COBWEB's behaviour when the rate of concept drift is sufficiently slow to follow. The data provides a single alteration of state at the 11th example.

The third experiment is in two parts. It shows the effect of applying COBBIT's queue mechanism on the same data sets as in the second experiment. Variation of the lower and upper queue size parameters shows the effect of the queue device on performance.

The fourth experiment compares COBWEB and COBBIT on a complex domain with six independent substates, altering their state cyclically and each at a different rate. The interference of state changes is intended to stress the performance capacity of both systems, and give an indication of what performance can be expected under complicated input.

Finally a comparison is provided which reflects the cost of assimilation for the next example. As expected, COBWEB needs more and more time while COBBIT stays bounded by the size of the queue.

The primary sources of evaluation are the graphs over prediction error and queue sizes, as presented later in this section. In most instances the graphs display averages from 10 data-sets. The fraction of error used as a measure of performance quality, is a sum over several attributes.

5.2 Description Language

The domain used is *equal–not equal*. It corresponds to an ideal situation using three binary attributes. Attributes A_1 and A_2 are regular attributes. They take their values from 0, 1. Attribute A_0 is the *cue* attribute, it supplies information about the state of the domain, the state that will be changed during learning. Table 4 gives all combinations of values. Since the complete set of possible examples quickly is exhausted, examples are repeated many times during learning.

Table 4. Domain *equal–not equal*.

A_0	A_1	A_2
0	0	1
	1	0
1	0	0
	1	1

5.3 Cobweb with and without Cue

Figure 1 shows the effect of hiding the cue attribute. The domain consists of three binary attributes from the *equal—not equal* domain. Each line represents the performance index averaged over 10 independent data sets, with 30 examples in each data set.

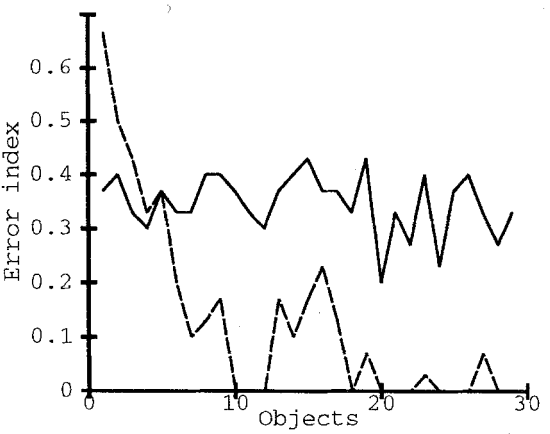

Fig. 1. Errors with visible and locked cue attributes.

The dotted line is the domain where the cue attribute reveals the state of the two other attributes. The state shifts randomly, but COBWEB quickly acquires

the pattern. The solid line shows the performance when the cue attribute is set to zero, regardless of the state of the domain. COBWEB has no way to discern between the two states and prediction does not improve as more examples are seen. Note that the with-cue line begins at a higher level of error than without cue. The reason for this is that with the cue is always set to zero, it cannot be mispredicted. The resulting error average is kept around 50 % of 2/3.

5.4 Single Shift at Example 11—Cobweb

This domain consists of three binary attributes from the *equal—not equal* domain. The training sequence begins in the unequal state and shifts to the equal state at the 11th example. It then holds that state to the end. The graph in figure 2 is an average of 10 independent data sets, with 30 examples in each data set. With the cue attribute locked the maximum error is 2/3.

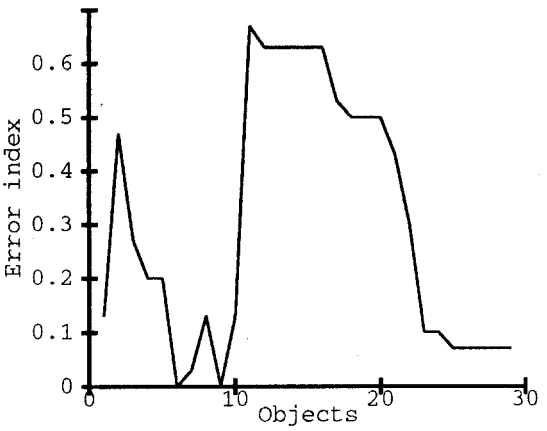

Fig. 2. Error for a single state, shifting at example 11.

Competence is gained as COBWEB begins to process the training set and the error level is low at the 10th example. When the domain changes, prediction suffers and the error soars to the maximum level. It takes another 11 examples before the error drops in earnest, but perfect performance as shown before the shift appears less likely.

The training instances in this run are identical to those in section 5.3, where COBWEB failed to learn (the solid line). The difference between the two experiments is that in the previous one the two states were randomly mingled. Here the states are sequentially separated, and COBWEB has time to accumulate conditional probabilities. After the shift has occurred, it takes almost the same number of examples to turn the accumulated counts the other way and perform well in the new state of the domain.

5.5 Single Shift at ex. 11—Cobbit Varying Queue Size

The graph in figure 3 is composed from four independent runs of COBBIT, using the queue mechanism. The data sets are in each case identical with the one used in the previous experiment, section 5.4. The parameter that is varied between each run is u, the upper limit of queue size. The lower limit l, was fixed at a size of 5 in all four runs.

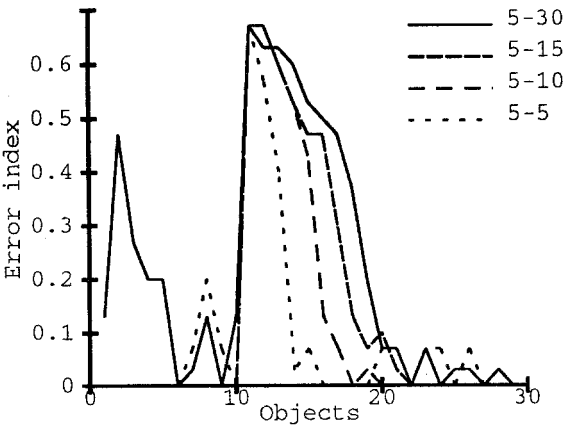

Fig. 3. Errors when varying the queue's upper limit.

The lines show the different settings of the u parameter; 5 (dotted), 10 (wide dash), 15 (long dash) and 30 (solid). Settings 20 and 25 are omitted but their positions are easily interpolated.

The graph shows how performance is regained must faster when the older examples (and their associated probabilities) are removed from the classification tree by the queue. Notice also that the difference between $u = 5$ and $u = 10$ is much larger than between $u = 15$ and $u = 30$. The reason for this is the simple domain; the short queue will flush out the old objects much faster. One should not be deceived into believing that a small value of u is beneficial under all circumstances; that would imply that all knowledge is harmful.

The queues for the four runs (figure 4) grows linearly until the shift occurs (with the exception of $u = 5$ and $u = 10$). At this point they level out and delay further growth of the queue until the error peak has diminished. When performance again is improving (error lessening) the queue again is allowed to grow.

A second collection of curves in figure 5 shows the behaviour when the queue's lower limit (l) is varied between 5 (dotted), 10 (wide dash), 15 (long dash) and 30 (solid). The upper limit was fixed at 30 in each instance.

The first striking difference is that all curves (except for $l = 5$) indicates a *worse* performance compared to figure 3. The curve for $l = 30, u = 30$ coincides

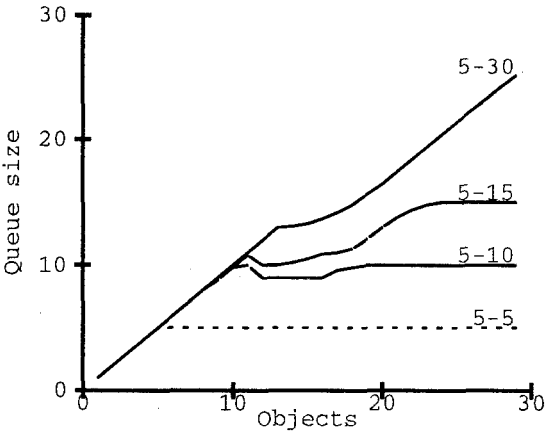

Fig. 4. Queue sizes when varying the queue's upper limit.

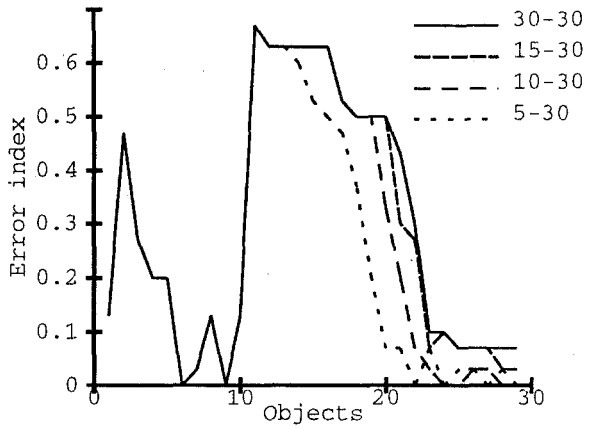

Fig. 5. Errors when varying the queue's lower limit.

with the performance for COBWEB as there are only 30 thirty examples and the queue never overflows. When the difference between the upper and lower limits is increased, COBBIT's control algorithm can, and does, use the queue to delete early examples.

The curve with the greatest range, 5-30 ($l = 5, u = 30$), is common to figures 3 and 5 both. It also appears to be the pivot point between the two graphs; in the former it appears as the worst case and in the latter as the best. Although the figures appear to advocate small settings of l and u this will only facilitate swift adaptation in the most trivial of cases.

Comparing the graphs that shows the evolution of the queue-size, a similar pivotal property is evident: in figure 4 the $l = 5, u = 30$ settings displays the

queue with the most examples, in figure 6 it is the one with the fewest.

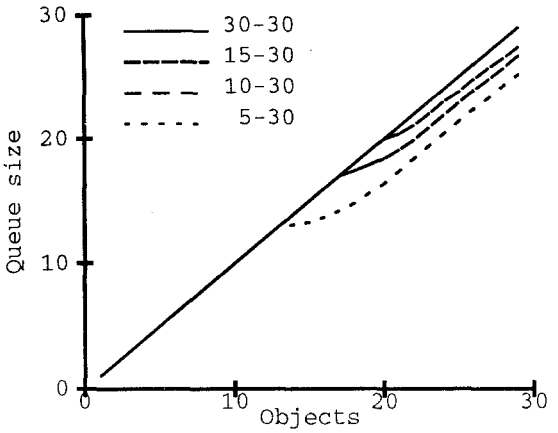

Fig. 6. Queue sizes when varying the queue's lower limit.

The queue must give the best performance for the most complex throughput and there is no simple strategy to guide the settings of the l and u parameters. A compromise appears unavoidable. If both parameters are set to similar values, the scope for dynamic behaviour lessens. Therefore is appears better to set u high, and provide COBBIT with the ability to exercise various queue sizes. The setting of l should be regarded as a minimal number of examples, beneath which generalization and induction becomes meaningless.

5.6 Complex Domain, Multiple Shifts

This test features a complex domain consisting of six concatenated attribute triplets from the equal—not equal domain. This yields a description language consisting of $6*3 = 18$ attributes. Each triplet is set up to alter its state cyclically, but the cycles are tuned to be out of phase so the resulting input data is shifting fast. The cycles used are 7, 15, 19, 29, 33 and 39. The resulting behaviour is that the three attributes $\{A_0, A_1, A_2\}$ alter their state every 7th example; attributes $\{A_3, A_4, A_5\}$ alter their state every fifteenth example, and so on. All cue attributes $\{A_0, A_3, A_6, A_9, A_{12}, A_{15}\}$ are locked to the value zero, giving a maximum error of 2/3. COBBIT's queues are set to $l = 10, u = 40$.

Figure 7 gives the performance graphs for COBWEB (dotted) and COBBIT (solid). The origin of each data point is an average from 10 runs, but the graph shows rolling averages over 10 adjacent data points. The purpose of this is to clearly show the trend in the material.

Both systems behave almost identically up to the 29th example where the projected limit on the queue is reached (see figure 8). COBBIT begins to eject old

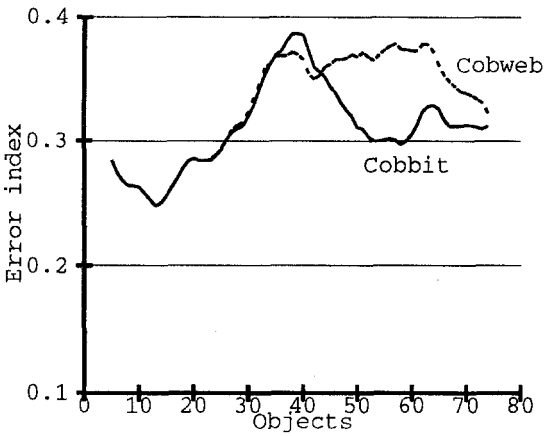

Fig. 7. Errors from Cobweb and Cobbit in the complex domain.

material, some of which apparently was important for prediction at this stage, since the level of error rise above COBWEB's . But COBBIT regains competence and displays a lower prediction error than COBWEB in the region of the 42nd example and onwards.

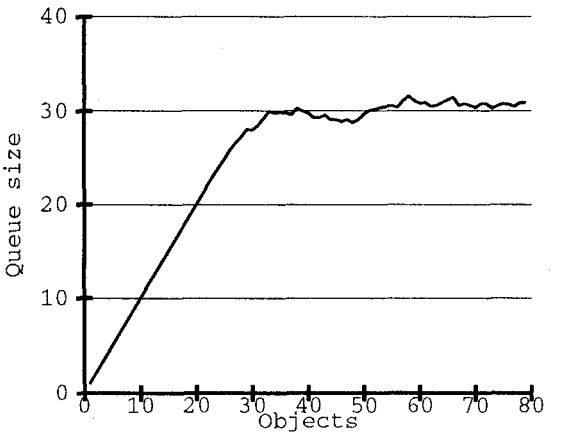

Fig. 8. Cobbit's queue size in the complex domain.

Also obvious in the graph is that COBWEB's error level also is dropping. This is because the number of possible examples in the domain are limited and COBWEB's collection of training examples is swelling. Should the run be extended, COBWEB will match COBBIT in predictive performance, because previous input will return. But the point is that COBBIT reacquires predictive per-

formance *faster* than does COBWEB, and that COBWEB makes no difference between $P(A) = 100/1000$ and $P(A) = 1/10$.

The final graph (figure 9) compares COBWEB and COBBIT on the cost of assimilating an object into the hierarchy. The absolute times per object are not interesting here; the curves show rolling averages of 10 adjacent data points, each of which is an average from 10 independent runs. The input data is the same as in figure 7. Note how COBBIT requires *more* processing time per example from (approximately) 20 to 36 objects. After that break-even is achieved and COBWEB continues to increase, while COBBIT stays behind.

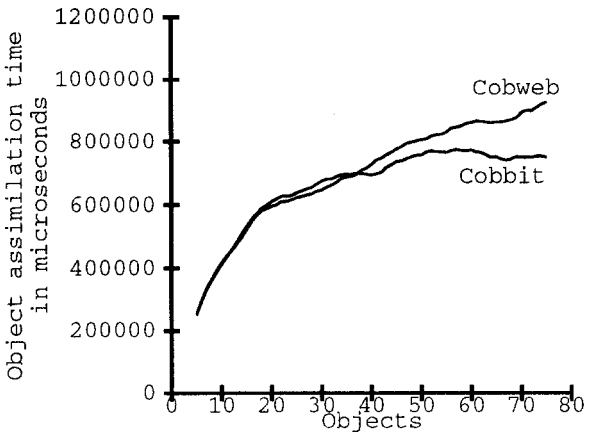

Fig. 9. COBWEB and COBBIT's object assimilation times.

6 Summary

The **desirable** properties of COBBIT include:

1. Predictions are based on recent input.
2. It is not burdened by lingering base rates.
3. Update time is bounded by a parameter.
4. It requires no change to COBWEB's classification procedure.

The experiments confirm that COBBIT recovers faster than COBWEB from drastic concept drift. The experiments also confirm that COBBIT in the long run require less processing time than COBWEB, due to COBWEB's monotonic accumulation of training examples, a behaviour avoided in COBBIT.

Among the **undesirable** properties of COBBIT are found:

1. Training examples and induced knowledge is lost (thrown away).

2. Examples are discarded indiscriminately, without regard for informative content.
3. It introduces two parameters, l and u which must be set by the user.

Section 3 provided a list of problems that appeared when COBWEB was used on data with concept drift. The COBBIT system introduce a list of devices: the two lists relate to each other in the following way.

COBWEB does not distinguish between new and old material, requiring new features to replace previous ones by quantity. In COBBIT, old instances are subtracted from the concept hierarchy when they reach the end of the queue. Their statistics are subtracted from all parents and they are effectively *unlearned*, using the *Extract_Object()* procedure.

COBWEB retains every training example and must stop when the resources for further learning are exhausted. In COBBIT it is the size of the queue that controls the number of examples retained. The size is continuously adjusted to approach a number determined by the recent performance, in the range between the l and u parameters.

The concept hierarchies created by COBWEB are usually dependent on the ordering of the training examples. The orderings found in data affected by concept drift are especially difficult. COBBIT does not alter the input ordering in any way. However, because of the queue mechanism and the subtraction of examples, ordering effects are only applicable to the training instances in the queue.

Acknowledgement

Thanks to our collegues in the Stockholm Machine Learning Group (ILP project members and others). This research was funded by the Swedish National Board for Industrial and Technical development under grant no. 9001734.

References

1. John A. Allen and Kevin Thompson. Probabilistic concept formation in relational domains. In *Machine Learning—Proceedings of the Eighth International Workshop (ML91)*, pages 375–379. Morgan Kaufmann Publishers, Inc, 1991.
2. Gautam Biswas, Jerry Weinberg, Qun Yang, and Glenn R. Koller. Conceptual clustering and exploratory data analysis. In *Machine Learning—Proceedings of the Eighth International Workshop (ML91)*, pages 591–595. Morgan Kaufmann Publishers, Inc, 1991.
3. Scott Clearwater, Tze-Pin Cheng, Haym Hirsch, and Bruce Buchanan. Incremental batch learning. In *Proceedings of the Sixth International Workshop on Machine Learning*, pages 366–370. Morgan Kaufmann Publishers, Inc, 1989.
4. Douglas Fisher. Noise-tolerant conceptual clustering. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, volume 1, pages 825–830. Morgan Kaufmann Publishers, Inc, 1989.
5. Douglas Fisher, Ling Xu, and Nazih Zard. Ordering effects in clustering. In *Machine Learning: Proceedings of the Ninth International Workshop (ML92)*, pages 163–168. Morgan Kaufmann Publishers, Inc, 1992.

6. Douglas Hayes Fisher. *Knowledge Acquisition via Incremental Conceptual Clustering.* PhD thesis, University of California, Irvine, 1987.
7. John Gennari, Pat Langley, and Doug Fisher. Models of incremental concept formation. *Artificial Intelligence*, (40):11–61, 1989.
8. Mirsad Hadzikadic and David Y. Y. Yun. Concept formation by incremental conceptual clustering. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 831–836, 1989.
9. Stephen José Hanson and Malcolm Bauer. Machine learning, clustering and polymorphy. In *Uncertainty in artificial intelligence*, pages 415–428. Elsevier Science Publishers, 1986.
10. Stephen José Hanson and Malcolm Bauer. Conceptual clustering, categorization and polymorphy. *Machine Learning*, (3):343–372, 1989.
11. Dennis Kibler and Pat Langley. Machine learning as an experimental science. In *Proceedings of the Third European Working Session on Learning.* Morgan Kaufmann Publishers, Inc, 1988.
12. Miroslav Kubat and Ivana Krizakova. Forgetting and ageing of knowledge in concept formation. Technical report, Computer Centre, Brno Technical University, Udolni 19, 60200 Brno, 1989.
13. Miroslav Kubat and Jirina Pavlickova. System flora: Learning from time-varying training sets. In Yves Kodratoff, editor, *Machine Learning—EWSL-91*, number 482 in Lecture Notes in Artificial Intelligence. Springer-Verlag, 1991.
14. Michael Lebowitz. Concept learning in a rich input domain: Generalization-based memory. Technical report, Department of Computer Science, Columbia University, New York, 1984.
15. Michael Lebowitz. Experiments with incremental concept formation: Unimem. *Machine Learning*, (2):103–138, 1987.
16. Ryszard S. Michalski and Robert E. Stepp. Learning from observation: Conceptual clustering. In Jaime G. Carbonell, Ryszard S. Michalski, and Tom M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 331–363. Tioga publishing company, 1983.
17. Ryszard S. Michalski and Robert E. Stepp. A theory and methodology of inductive learning. In Jaime G. Carbonell, Ryszard S. Michalski, and Tom M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 83–129. Tioga publishing company, 1983.
18. Bernd Nordhausen. Conceptual clustering using relational information. In *Proceedings aaai-86 Fifth National Conference on Artificial intelligence*, pages 508–512, 1986.
19. Jeffrey Schlimmer and Richard Granger, Junior. Beyond incremental processing: Tracking concept drift. In *Proceedings aaai-86 Fifth National Conference on Artificial intelligence*, pages 502–507, 1986.
20. Jeffrey Schlimmer and Richard Granger, Junior. Incremental learning from noisy data. *Machine Learning*, 1(3):317–354, 1986.
21. Gerhard Widmer and Miroslav Kubat. Effective learning in dynamic environments by explicit context tracking. In *Proceedings of the European Conference on Machine Learning (ECML-93*, Vienna, Austria, 1993.