A Fast Partial Memory Approach to Incremental Learning through an Advanced Data Storage Framework

Marenglen Biba, Stefano Ferilli, Floriana Esposito,
Nicola Di Mauro, Teresa M.A Basile

Department of Computer Science, University of Bari
Via E. Orabona, 4 - 70125 Bari, Italy
{biba, ferilli, esposito, ndm, basile}di.uniba.it

Abstract. Inducing concept descriptions from examples has been thoroughly tackled by symbolic machine learning methods. However, on-line learning methods that acquire concepts from examples distributed over time, require great computational effort. This is not only due to the intrinsic complexity of the concept learning task, but also to the full memory approach that most learning systems adopt. Indeed, during learning, most of these systems consider all their past examples leading to expensive procedures for consistency verification. In this paper, we present an implementation of a partial memory approach through an advanced data storage framework and show through experiments that great savings in learning times can be achieved. We also propose and experiment different ways to select the past examples paving the way for further research in on-line partial memory learning agents.

1 Introduction

Inductive concept learning concerns the inference of hypothesis from examples. There are two ways to process the examples: batch or incremental. The learning systems that adopt the batch paradigm simply store all the past examples and at each new incoming example just re-apply the method. This brings several disadvantages such as memory requirements for very large datasets, high computational learning times and difficulty in dealing with changing concepts over time. For these reasons, efforts have been made to built incremental learners that avoid the harmful effects of the batch approach. Incremental systems generally have lower memory requirements, lower learning time and are able to deal much better with drifting concepts.

However, in the incremental learning model the examples are distributed over the time, hence a good storage technique is of vital importance for the system efficiency. Incremental learning paradigm can be implemented using one of three memory models: *no instance memory*, *partial instance memory*, and *full instance memory*. In the *full instance memory*, such as IB1 [1], the system retains all the previous examined examples. In the *partial instance memory*, see for example FLORA [2], LAIR [3], HILLARY [4], DARLING [5], AQ-PM [6], the system memorizes only part of the past examples. Finally, the *no instance memory* model consists in

forgetting the examples as soon as the learning step is completed. Examples of such systems are Weighted Majority [7], Winnow [8], Stagger [9], Arch [10].

As pointed out in [11] there are three main reasons for studying partial memory learners. First, humans when learn from experience, store not only concept descriptions but also particular significant events and they may remember these events forever even though many new other events may happen. Second, the machine learning community has not dedicated much work to on-line partial memory systems. In fact, almost all learning systems either store all the encountered examples or none of them. Third, partial memory learners react more quickly to drifting concepts.

A fundamental issue for partial memory approach is the choice of the *representative* instances to memorize. In the partial memory incremental learning approach exploited in AQ-PM [6], the learned concepts are used to determine which training examples establish the outer bounds of a concept. The representative positive examples of a concept are those examples that lie at the boundaries of the concept in the event space, while the representative negative examples are those that constrain the concept in the event space (all other training examples can be discarded). Specifically, after the initial concepts have been learned they are used by the system for inference. If the system makes a wrong decision, then this is a signal that the concepts require refinement. The misclassified training example is included within the existing representative examples and inductive learning takes place. Once new concepts are learned, the training examples are evaluated and those determined to be representative are selected to form the new set of representative examples. The new concepts are used for inference and the new representative training examples are stored. This process repeats indefinitely.

Other systems that exploit the *partial instance memory* are: LAIR [3], that retains all the first positive examples available; HILLARY [4], that memorizes the negative instances only; DARLING [5] that associates a weight equal to 1 to each new incoming instance decreasing the weight of the past instances according to a neighbor policy and eliminating the instance from the memory as soon as the associated weight is lower than a threshold.

In this paper we focus on incremental systems that refine theories with a partial memory approach. The sub-field of machine learning where our work is positioned is that of learning logic programs from examples [12], i.e. concepts and examples are described in logic representation and in particular in first-order Horn clauses. The goal is to transform the multi-relational inductive learning system INTHELEX [13] so that it can adopt a partial memory approach. We also want to show through experiments that the partial memory implementation in learners of logic programs and systems that refine theories, can drastically reduce learning times.

To the best of our knowledge, this is the first attempt to implement a partial memory approach in a logic-based theory refining system. The most similar system is GEM-PM [11] that is an incremental learning system with partial memory that employs both refinement operators. However, the concept description language is not based on first-order logic such as INTHELEX language. Since GEM-PM is based on the AQ learning algorithm [14], it learns propositional concept descriptions, while INTHELEX learns first-order concept descriptions. Other systems such as FORTE [15] refine theories in a first-order setting but do not employ partial memory during the refinement process. Most of other work in the first-order setting [12] is related to

batch or incremental learning systems that either do not refine theories or do not employ partial memory.

The paper is organized as follows. Section 2 presents the field of inducing concept descriptions as logic programs and the incremental multi-relational learning system INTHELEX. Section 3 presents the advanced data storage framework through which the partial memory is implemented. Section 4 describes how the partial memory is implemented. Section 5 presents experiments and Section 6 concludes and presents future work.

2 Inducing Concept Descriptions as Logic Programs

First-order logic provides a theoretically robust background to reasoning and a powerful knowledge representation framework. Logic-based formalisms' semantics and their descriptive power have been exploited to induce concepts and reason about them in domains with structural and relational requirements. One of the fields that have witnessed much work is that of multi-relational learning [12]. In this research area, concepts are described as first-order Horn clauses which represent a sub-set of pure first-order logic. The learning task consists in inducing concepts given a set of positive and negative examples in the presence of an eventually available background knowledge. A concept is usually intended as a learned theory in the form of *if…then…* rules that compose it. The main operators in the learning task are those that generalize and specialize a certain rule in the theory. When a new positive example is not explained by the theory then the generalizing operator starts a search for a hypothesis (in the space of possible hypothesis usually ordered by θ-subsumption) that can explain the example. If a new coming negative example is explained by the theory, then the specializing operator identifies the responsible rule for explaining the example and tries to specialize it by searching the candidate hypothesis in the subsumption lattice. Usually multi-relational learning systems adopt one of the two operators to search the space of hypothesis. They, either start from the most possibly general hypothesis and try to specialize it (top-down) or from the most specific hypothesis and generalize it (bottom-up). Involving both operators is known as theory refinement [16] and is considered to be a very hard task.

There are three issues of multi-relational learning to be considered for the purposes of this paper. First, we are interested in the way examples are stored and considered for future learning processes. Second, how the learning process is performed, batch or incremental. Third, how the hypothesis space is searched. The current learning system INTHELEX that we refer to in this paper adopts a full memory approach, i.e. it maintains all the past positive and negative examples and whenever a hypothesis must be evaluated during search, it must be checked against all the past examples. INTHELEX is an incremental system, i.e. learning can start from scratch or from an existing theory. In the latter case, the existing theory can be generalized or specialized according to the arriving examples. Thus, the system implements refining operators. Moreover it can learn hierarchical concepts, when a concept depends on another concepts, it interrupts the search for the hypothesis for the main concept and starts searching for hypothesis for the composing concept.

To the best of our knowledge, there have not been other works on refining first-order logical theories through partial memory approaches. This is due to the fact that not many refining systems exists due to the high complexity of the refinement task. In the following we sketch the main algorithms in INTHELEX and show how the partial memory approach can be implemented.

```
Procedure Generalize (E: positive example; T: theory;
                      M: set of all negative examples)

  L := Take all clauses whose concept is the same as that of E.
  Consistency = false
  While there are clauses in L do
     Take a clause C from L
     While (Consistency = false and  there are no more generalization for C) do
          Candidate_Hypothesis = Generate a generalization of C and E
           Consistency = Check_Consistency(Candidate_Hypothesis,M)
     End While
  End While
 End Procedure
```

```
Procedure Specialize (E: negative example; T: theory;
                      M: set of all positive examples)

  L := Take all clauses from T that participate in the derivation of the negative
  example E.
  Completeness = false
  While there are clauses in L do
     Take a clause C from L
     While (Completeness = false and  there are no more specialization for C) do
          Candidate_Hypothesis = Generate a specialization of C and E
           Completeness = Check_Completeness(Candidate_Hypothesis,M)
     End While
  End While
 End Procedure
```

The most expensive procedures in theory refinement are those that check the consistency and completeness towards the past examples. When a positive example is not covered by the theory the procedure *Generalize* starts a search in the space of hypothesis by generating and testing each candidate hypothesis' consistency towards all the past negative examples. Since the hypothesis that can be generated are too many, checking every candidate towards all the past examples becomes a bottleneck. The idea of the partial memory is to check the consistency only against a part of the

negative examples. On the contrary, when a negative example is covered, the theory must be specialized. The procedure *Specialize* greedily generates specializations each of which must be checked for completeness against all the past positive examples. Again, this becomes overwhelming in terms of computational times and if we are unlucky we may check for completeness *n-1* candidates on all the available examples and then find that the *n-th* is the only one that is complete. With the partial memory, the number of examples to check is lower and, as we will show in the next sections, this can bring significant savings in time at no significant changes in the learning accuracy.

3 The Berkeley DB Framework and the Prolog API

BDB (Berkeley DB) [17] is an advanced framework for data management. It is an open-source project whose goal is that of producing a database library able to provide high performance data access and highly efficient data management services. It was initially developed at the University of California at Berkeley and later at University of Harvard. BDB is a tool for software developers, the only way to use it is to write code. There is no standalone server and no SQL query tool for working with BDB databases. It provides a powerful Application Programming Interface (API) that permits to develop fast and reliable embedded solutions. The basic structures of BDB are B-tree, hash table and persistent queues. Over these structures BDB offers advanced services such that multiple threads or processes can operate on the same collection of B-trees and hash tables at the same time without risk of data corruption or loss. One of the most appealing feature of BDB is the ease of deployment. The API supports many programming languages such as JAVA, C, C++ etc, so that it can be easily embedded in end-user or infrastructure applications. Moreover, since the database engine is literally in the same address space as the application, no database access needs to cross a process or network boundary. In many relational DBMSs, context switch time is an important bottleneck, and also moving data across network boundaries on different machines is even more expensive. BDB eliminates both these disadvantages. A comparison of BDB towards relational DBMSs can be found in [18].

For our purposes in this paper we are interested in some features of a data storage system. In particular, we refer to the Prolog API for BDB provided by SICStus [19]. The most important feature for storing logic terms is the possibility to store logic variables. Most DBMSs are not able to store non-ground terms or more than one instance of a term. The Prolog API for BDB permits to store variables with blocked goals as ordinary variables. This is a powerful feature in a logic-based programming system and since INTHELEX is a learning system written in Prolog we can use the above data storage Prolog API for a partial memory approach. However, as stated above, BDB does not offer a SQL-like query language. For every data management service, wrapping code must be written to provide data access. We have developed a wrapping layer in the Prolog language to implement a partial memory solution for the system INTHELEX.

In the following we describe the Prolog API of SICStus. This interface exploits the Concurrent Access Methods product of BDB. This means that multiple processes can open the same database. The environment and the database files are ordinary BDB entities that use a custom hash function. The idea is to obtain a behavior similar to the built-in Prolog predicates such as assert/1, retract/1 and clause/2, but having the terms stored on files instead of main memory. Some differences with respect to the Prolog database are:

- The functors and the indexing specifications of the terms to be stored have to be given when the database is created.
- The indexing is specified when the database is created. It is possible to index on other parts of the term than just the functor and first argument.
- Changes affect the database immediately.
- The database will store variables with blocked goals as ordinary variables.

The db-specification (*db-spec*) defines which functors are allowed and which parts of a term are used for indexing in a database. The *db-spec* is a list of atoms and compound terms where the arguments are either + or -. A term can be inserted in the database if there is a specification (*spec*) in the *db-spec* with the same functor. Multilevel indexing is not supported, terms have to be “flattened”. Every *spec* with the functor of the indexed term specifies an indexing. Every argument where there is a + in the *spec* is indexed on.

A *db-spec* has the form of a specification-list (*speclist*):

speclist = [spec1, . . . , specM]
spec = functor(argspec1, . . . , argspecN)
argspec = + | -

where *functor* is a Prolog atom. The case N = 0 is allowed. A *spec F(argspec1, . . . , argspecN)* is applicable to any ground term with principal functor F/N.

When storing a term T, it is generated a hash code for every applicable *spec* in the *db-spec*,and a reference to T is stored with each of them. (More precisely with each element of the set of generated hash codes). If T contains ground elements on each + position in the *spec*, then the hash code depends on each of these elements. If T contains some variables on + position, then the hash code depends only on the functor of T. When fetching a term Q we look for an applicable *spec* for which there are no variables in Q on positions marked +. If no applicable *spec* can be found a domain error is raised. If no *spec* can be found where on each + position a ground term occurs in Q an instantiation error is raised. Otherwise, we choose the *spec* with the most + positions in it breaking ties by choosing the leftmost one. The terms that contain ground terms on every + position will be looked up using indexing based on the principal functor of the term and the principal functor of terms on + positions. The other (more general) terms will be looked up using an indexing based on the principal functor of the term only.

4 Implementing Partial Memory in INTHELEX

In a recent work [20] the authors presented the design and implementation of an external storage solution of logic terms by integrating BDB in INTHELEX through the SICStus Prolog API. It was shown that this upgrade resulted in much better performance against the old version of INTHELEX that used only the internal memory of the SICStus Prolog interpreter. In this solution, logic terms were stored in BDB instead of being loaded in main memory. Moreover, it was implemented a relational schema linking every example to the clause that explains it. This helps a lot in the theory refinement process because permits to check the candidate clause refinements only against the examples already covered by the clause that is being refined. Experiments in [20] showed that using the fast access of BDB and the relational schema, outperformed the old version of the system in terms of learning time.

In Figure 1 it is shown the schema implemented in [20]. In the following we briefly explain the schema. It makes possible to link clauses to examples that are covered by these clauses. Whenever one of these clauses is refined, the relational schema permits a fast access only to those examples that are related to the clause. We chose to include examples in a database and clauses in another. We maintained two separate tables for examples and descriptions in order to have the possibility to preserve useful information about descriptions that can be used either afterwards to assess and study the learning process or during the learning task for accelerating the access to data. We also designed a table with only two fields that serves as a hash table to fetch all the examples covered by a clause. In fact, the two fields represent respectively the key of the clause and the key of the example. Another table was introduced in order to maintain all the versions of the theory during the refinement steps. This could be very useful for the learning task and the previous versions of the theory could be exploited for improving the searching on the space of the hypothesis. Regarding the information about the examples, we keep useful statistics about the examples and the observations.

We can also group the examples in order to distinguish between those that have been given in input to the system at a time and those have been given at another moment. This might be useful to study the learning task from an *example grouping* point of view, trying to analyse and distinguish those examples that have a greater impact on the learning task. As regards the clauses, we keep other information such as the generating example of the clause. This represents another precious fact that could be useful in understanding better the learning process.

The *db-spec* that results from the schema in Figure 1 is simple. The following specifications show how the tables are implemented through the *db-spec* of BDB. As it can be seen, the attributes which are marked with "+, represent the keys of the clauses and examples, thus the database is indexed on these attributes.

[examples(+,-,-,-,-,-), obs(+,-,-,-), max(+,-,-,-)]
[clauses(+,-,-,-,-,-,-,-,-), theory_changes(+,-,-,-,-), clauEx(+,-), pos_excep(+,-,-), neg_excep(+,-,-), max(+,-,-,-,-,-)]

We have used the indexed attributes for our purpose of implementing a relational schema between clauses and examples but however the database engine of BDB uses these attributes for efficient fetching of the terms. This is due to the schema adopted by this database engine whose fetching mechanism is based on the indexed terms.

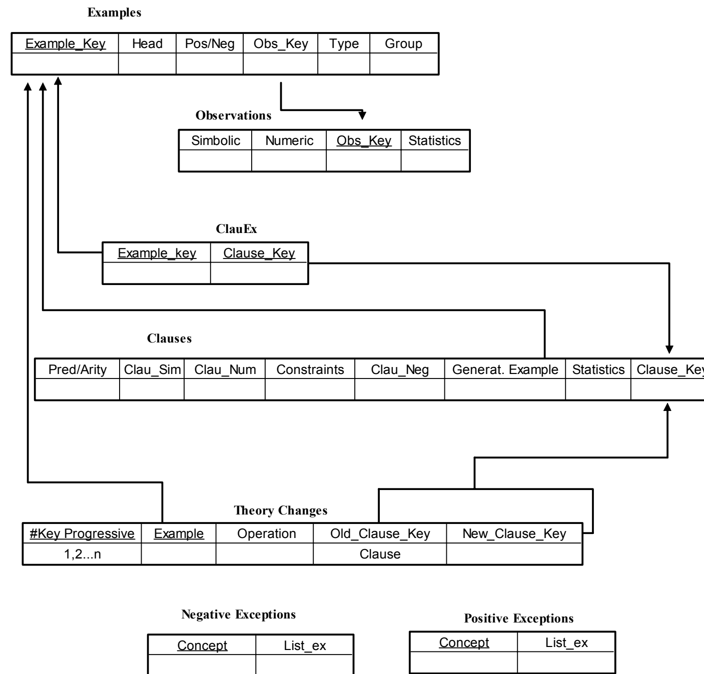

Figure 1. Linking examples to clauses and partial memory solution through the attributes *Example_Key, Type* and *Group*.

Concerning the implementation of the partial memory strategy, for each example we kept a progressive key, *Example_Key,* associated to each single training instance after being processed. Furthermore, the *Type* attribute was associated to each example to distinguish the observations that caused a change in the theory. With this information, the refinement operators could consider the last *n* observations in the order they arrived to the system, or take into account the last *n* observations among those that generated a theory refinement when they were considered by the system. Thus, there are different options for implementing the partial memory by simply modifying the procedures *Check_Consistency* and *Check_Completeness* of the refining operators of Section 2 such that only a portion of the past examples is considered for consistency and completeness checking.

5 Experimenting Partial Memory

The test of the performance of the system when the partial memory functionality is activated concerned the classification and understanding task on scientific paper documents [13]. In particular the classification task was performed on the class of documents belonging to three different series `ICML` (28- the International Conference on Machine Learning), `SVLN` (32- Springer Verlag Lectures Notes) and `IEEET` (36 – IEEE Transactions on Pattern Analysis and Machine Intelligence). Each positive example for a concept target was considered negative for all the other concepts to be learned. The understanding task was performed on the *Author* [36+, 332-], *Page Number* [27+, 341-] and *Title* [29+, 339-] logical components of the `ICML` class of documents. The dataset was randomly split 33 times in a Tuning set, containing 70% of the examples, and a Test set, containing 30%, both guaranteeing a fair proportion of positive and negative examples for each class. We combined different alternatives for the partial memory and modified the procedures Check_Completeness and Check_Consistency presented in Section 2. Initially three kinds of experiments were carried out with the partial memory: **1.** in the first one approximately the last 25% of the past observations were considered; **2.** in the second experiment only the observations that modified the theory were considered; **3.** in the last experiment we took into account the observations that modified the theory among the last 25% of all the past observations. All the experiments were performed on a Pentium 4, 2.4 GHz, using SICStus Prolog 3.11.1.

As shown in Tables 1 and 2, that report the mean predictive accuracy and mean learning time in seconds for the 33 fold, in all the experiments both for the classification and understanding process, the execution time decreases significantly; as regards the percentage of predictive accuracy, there is no significant difference in the case of full memory or partial memory with the option **1**, thus we could accurately learn the theory considering only a part of the learning set. However, with the partial memory options **2** and **3** the accuracy decreases significantly.

Table 1. Classification with Partial Memory Functionality

	ICML		**SVLN**		**IEEET**	
	Acc. %	Time(s.)	Acc. %	Time(s.)	Acc.%	Time(s.)
Full Memory	97,75	3,15	87,36	19,45	90,57	27,55
Partial Memory						
1	97,36	1,73	87,57	6,63	88,75	6,13
2	89,84	1,45	84,63	2,66	84,39	3,04
3	89,48	1,33	81,00	2,81	80,27	2,60
4	96,18	1,95	83,90	5,25	87,39	6,9

Table 2. Understanding with Partial Memory Functionality

	Author		***Page Number***		***Title***	
	Acc. %	Time(s.)	Acc. %	Time(s.)	Acc.%	Time(s.)
Full Memory	97,12	29,07	97,54	76,22	97,87	51,66
Partial Memory						
1	96,87	10,27	97,42	27,17	97,96	19,65
2	94,54	3,09	90,06	5,58	94,69	3,25
3	88,24	3,36	94,06	5,77	94,51	2,61
4	97,06	4,95	96,33	12,11	96,81	13,19

We performed a thorough analysis of the results with the partial memory options **2** and **3**. We discovered that on some of the 33 folds the predictive accuracy decreased drastically and this influenced the overall accuracy on the 33 folds. The reason of the decrease was that in these folds there were no negative examples that caused changes to the theory and thus the system refined the theory considering only positive examples that modified the theory. This resulted in over-generalization of the theory and in the test phase many negative examples were covered by the theory. Therefore, the natural solution to this problem was to include negative examples in the set of examples that represent the partial memory. To do this we initially included in the partial memory only the first negative example encountered. However, this did not change the situation and the predictive accuracy continued to be low. At this point, we empirically discovered that if there were some negative examples, also very few, the predictive accuracy did not decrease. For this reason, we decided to adopt the following solution. We decided to maintain in the partial memory a certain number of negative examples that constituted at every moment of the learning task a certain percentage of the overall number of examples. The procedure that updated the partial memory kept constantly at every arrival of each example, a set of negative examples. Among these, there were examples that had caused modifications of the theory and others that had not. For instance, if at a certain moment of the learning task were processed 100 examples, and there were two negative examples that had caused theory refinements and the procedure for updating the partial memory had received 5% as parameter for the percentage of negative examples, then the partial memory contained totally 5 negative examples. The remaining three negative examples were randomly chosen from those that did not modify the theory.

Due to the different number of available observations for each dataset, the percentage of negative observations was 10% for the classification process and 2% for the understanding one. The results reported in Tables 1 and 2, option **4.** for the partial memory, show an improvement of the predictive accuracy towards option **2** and **3**. From all the solutions for the partial memory, the combination of the modifying examples with a certain fixed percentage of negative examples, resulted the best strategy.

The experimental results suggest that there exists a certain number of examples that can guide the learning task and can avoid over-generalizations or under-

specializations. Therefore, it is not necessary to continue refining the theory against all the past examples if the same accuracy but at much lower time is achieved through the partial memory. It also seems highly likely that among the negative examples, there exists a sufficient set whose informative contents seems to optimally conduct the learning process. Distinguishing among the negative examples that do not modify the theory, those that more than others influence the learning task is an interesting task. In this paper, we choose them randomly to always keep in the partial memory a constant percentage of negative examples. A non-random selection would require a principled criteria. We believe information-theory based criteria such as information gain can be used. We plan to investigate this in the future.

6. Discussion and Future Work

We have implemented a partial memory solution to incremental learning and theory refinement and shown through experiments that the partial memory approach outperforms the traditional one in terms of learning times at no significant changes in predictive accuracy. The approach is implemented through an advanced term storage framework such as BDB and the relative Prolog API that is able to store variables. We have experimented various strategies for the partial memory showing that a small number of examples can guide the learning process towards optimal generalizations and specializations. Identifying this small set of examples is not always a straightforward task. In this paper, we have shown that combining a set of positive examples that modify the theory with a small number of negative examples, produces an enough training set to achieve very good results.

To the best of our knowledge, this is the first attempt to implement a partial memory approach in a logic-based theory refining system. The most similar system implementing an incremental approach with partial memory and that employs both refinement operators is GEM-PM [11] that, however learns propositional concept descriptions, while INTHELEX learns first-order concept descriptions. Most of other work in the first-order setting [12] is related to batch or incremental learning systems that either do not refine theories or do not employ partial memory.

The experiments in this paper for a refinement task, suggest that theory refinement in first-order settings can highly benefit from partial memory approaches leading to significant time savings at no significant loss of accuracy. As future work, we plan to investigate how we can identify among the modifying examples those that are more significant, i.e. examples that cause different refinements in the theory and, most probably, the ones that cause the most substantial changes are those that must be kept for future use in the learning task. Again, the implemented schema with BDB can be very useful, because it permits to keep every intermediate change in the theory and the example that caused it. In this way, we can weight an example based on its history of theory refinements and thus evaluate the "importance" of each example for the learning process.

References

1. Aha. D., Kibler. D., Albert. M. Instance-based learning algorithms, *Machine Learning* 6 37–66, (1991).
2. Widmer. G., Kubat. M. Learning in the presence of concept drift and hidden contexts, *Machine Learning* 23, 69–101 , (1996).
3. Elio. R., Watanabe. L. An incremental deductive strategy for controlling constructive induction in learning from examples, *Machine Learning* 7 7–44, (1991).
4. Iba. W., Woogulis. J., Langley. P. Trading simplicity and coverage in incremental concept learning, in: *Proceedings of the Fifth International Conference on Machine Learning*, Morgan Kaufmann, San Francisco, CA, pp. 73–79, (1988).
5. Salganicoff. M., Density-adaptive learning and forgetting, in: *Proceedings of the Tenth International Conference on Machine Learning*, Morgan Kaufmann, San Francisco, CA, pp. 276–283, (1993).
6. Maloof., M. Michalski. R. Selecting examples for partial memory learning, *Machine Learning* 41, 27–52. (2000).
7. Littlestone. N., Warmuth. M. The Weighted Majority algorithm, *Inform. and Comput*. 108 212–261, (1994).
8. Littlestone. N. Redundant noisy attributes, attribute errors, and linear-threshold learning using Winnow, in: *Proceedings of the Fourth Annual Workshop on Computational Learning Theory*, Morgan Kaufmann, San Francisco, CA , pp. 147–156, (1991).
9. Schlimmer. J., Granger. R. Beyond incremental processing: Tracking concept drift, in: *Proceedings of AAAI*- 86, Philadelphia, AAAI Press, pp. 502–507, (1986).
10. Winston. P. Learning structural descriptions from examples, in: P. Winston (Ed.), *Psychology of Computer Vision*, MIT Press, Cambridge, MA, (1975).
11. Maloof. M., Michalski. R. S. Incremental learning with partial instance memory. *Artificial Intelligence* 154, pp. 95-126, (2004).
12. Dzeroski. S., Lavrac. N. (Eds) *Relational Data Mining*. Springer, Berlin, (2001).
13. Esposito. F., Ferilli. S., Fanizzi. N., Basile. T.M.A., Di Mauro. N. Incremental Multistrategy Learning for Document Processing, *Applied Artificial Intelligence: An International Journal*, Vol. 17, n. 8/9, pp. 859-883, Taylor Francis, (2003).
14. Michalski. R. On the quasi-minimal solution of the general covering problem, in: *Proceedings of the Fifth International Symposium on Information Processing*, vol. A3, pp. 125–128, (1969).
15. Richards. B. L., Mooney. R.J. Refinement of first-order horn-clause domain theories. *Machine Learning Journal*, 19(2):95–131, (1995).
16. Mooney. R.J. Batch versus Incremental Theory Refinement. *Proceedings of the 1992 AAAI Spring Symposium on Knowledge Assimilation*, Standford, CA, (1992).
17. Berkeley DB reference: http://www.oracle.com/database/berkeley-db.
18. Oracle white paper: A Comparison of Oracle Berkeley DB and Relational Database Management Systems, http://www.oracle.com/database/docs/Berkeley-DB-v-Relational.pdf.
19. SICStus Prolog reference: http://www.sics.se/sicstus
20. Biba. M., Basile. T.M.A., Ferilli. S., Esposito. F. Improving Scalability in ILP Incremental Systems. *Proceedings of CILC 2006 - Italian Conference on Computational Logic*, Bari, Italy, June 26-27, (2006).