Improving scalability in ILP incremental systems

Marenglen Biba, Teresa Maria Altomare Basile, Stefano Ferilli, Floriana Esposito

Dipartimento di Informatica – Università degli Studi di Bari
via Orabona 4 - 70126 Bari
{biba, basile, ferilli, esposito}@di.uniba.it

Abstract. This paper presents an approach for extending an existing ILP system to deal with the scalability issue of inductive learning systems. Efficient data access in ILP has been tackled by previous approaches through the use of SQL, subsampling and local coverage test. The work presented here provides another method for efficient access of terms that allows also to generate and preserve precious information about the learning process that could be used during the learning task to guide the search for hypotheses. Although the learning algorithms in the ILP system do not change the way they work, coverage test of examples and theory refinement are employed using a hash based access to terms which are no longer loaded in main memory. Experiments conducted with the improved system indicate that it performs better for large datasets.

1. Introduction

Scalability is gaining increasing interest in ILP research in order to make it the main stream for multirelational data mining [1] [2]. In these domains, huge datasets must be explored in order to discover relationships among data, leading therefore to critical efficiency matters to be dealt with.

One of the approaches to the scalability problem could be using commercial database management systems (DBMS) that translate a goal query to the corresponding SQL query. These methods might not lead to scalable ILP systems because the internal efficiency of DBMSs should be taken into account. Recent attempts aim at ILP specific solutions. For instance, sub-sampling methods that focus on smaller sets of examples have produced good results [3]. Another ILP-tailored solution is local coverage test within a learning from interpretations setting where independence assumptions among examples are reasonable and coverage of each example can be tested without considering the entire background knowledge [4]. Furthermore, ad-hoc theta-subsumption techniques to speed-up the coverage test have been investigated in [5], [6].

In this paper, we present a powerful method for fast access to large datasets without changing the learning algorithms, applied to the ILP system INTHELEX (INcremental THEory Learner from EXamples) [7]. Such an approach stores the examples and the theory in an external database, that is then accessed for coverage tests of examples and for clause generation without loading any example into main memory.

All the terms in the database are indexed, this way enhancing the scalability and efficiency of the learning system.

To empirically demonstrate the efficiency gain, we extended the ILP system INTHELEX in order to fetch terms no longer from main memory but by calling predicates that interact with the external storage of terms. Such storage is a hash-based database based on the Berkeley-DB [8] data engine that provides fast access to data. While the two versions of INTHELEX produce almost the same set of rules, runtimes are quite different showing that the new version is more efficient on large datasets. For a solid comparison of the results, different datasets were used for the experiments.

The paper is organized as follows. The next section illustrates the learning system INTHELEX which loads data in main memory. Section 3 presents the scalable version of INTHELEX. Then, Section 4 reports the experimental results and Section 5 contains conclusions.

2 INTHELEX: an internal memory ILP system

Almost all ILP systems are based on an internal memory usage to process examples and clauses. INTHELEX is an ILP system for the induction of hierarchical theories from positive and negative examples [9], whose architecture is depicted in Figure 1. It is fully and inherently incremental: this means that, in addition to the possibility of taking as input a previously generated version of the theory to be refined, learning can also start from an empty theory and from the first available example; moreover, at any moment the theory is guaranteed to be correct with respect to all of the examples encountered so far. This is a fundamental issue, since in many cases deep knowledge about the world is not available. It adopts a *close loop* learning process, where feedback on performance is used to activate the theory revision phase. INTHELEX can learn simultaneously various concepts, possibly related to each other. The correctness of the learned theory is checked on any new example and, in case of failure, a revision process is activated on it, in order to restore completeness and consistency. It adopts a full memory storage strategy − i.e., it retains all the available examples, thus the learned theories are guaranteed to be valid on the whole set of known examples. It incorporates two inductive operators, one for generalizing definitions that reject positive examples, and the other for specializing definitions that explain negative examples. Both these operators, when applied, change the answer set of the theory, i.e. the set of examples it accounts for.

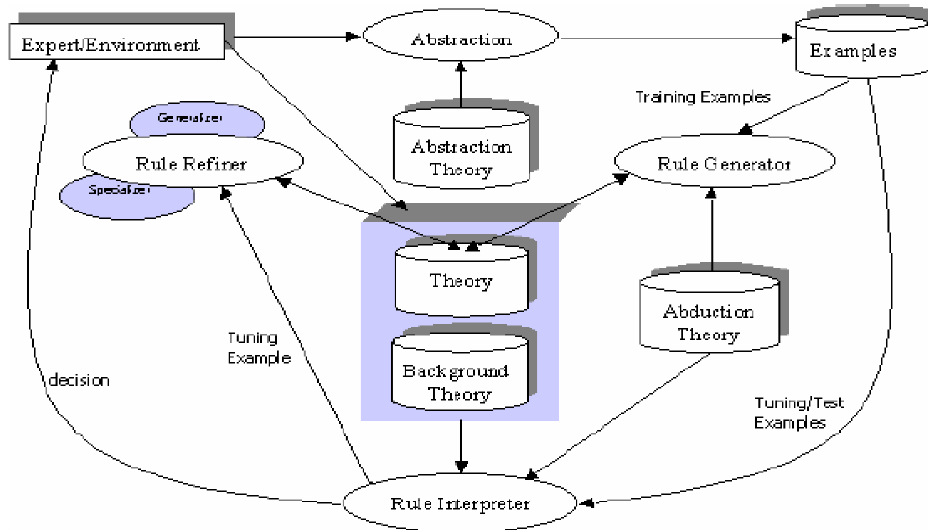

Figure 1. Architecture of INTHELEX

Algorithm 1

```
Procedure Tuning (E: example; T: theory; M: historical memory);
       If E is a positive example AND ¬ covers(T,E) then
              Generalize(T,E,M⁻)
       else
              if E is a negative example AND covers(T,E) then
                     Specialize(T,E,M⁺)
              end if
       end if
M := M U E.
```

Algorithm 1 shows the procedure implementing the Rule Refiner. It concerns the tuning phase of the system, where $M = M^+ \cup M^-$ represents the set of all negative (M^-) and positive (M^+) processed examples, E is the example currently examined, T represents the theory generated so far according to M plus the background knowledge BK. When a positive example is not covered by the theory, the procedure Generalize is called in order to restore the completeness of the theory.

Algorithm 2.

```
Procedure Generalize (E: positive example, T: theory, M⁻: negative
examples);
L := (non BK)clauses in the definition of the concept which
      E refers to
generalized := false; gen_lgg := 0
while ((¬ generalized) AND (L ≠ ∅)) do
       C := choose_clause(L) ; L := L\{C}
       while ((∃ another l ∈ lgg_OI(C,E)) AND
               (¬generalized) AND (gen_lgg ≤ max_gen)) do
              gen_lgg + 1
```

```
            if (consistent({T\{C}} U {l},M⁻)) then
                  generalized := true ; T := {T\{C}} U {l}
            end if
      end while
end while
if (¬generalized) then
      G := turn_constants_into_variables(E)
      if consistent(T U {G},M⁻) then
            T := T U {G}
      else
            T := T U {E} {positive exception}
      end if
end if
```

Algorithm 2 shows the procedure *Generalize*. When searching for a *least general generalization* (lgg), the algorithm must check whether the generated *lgg*s are consistent with the negative examples in the main memory. To do this, it must scan the entire set of previous examples, that have been loaded in main memory, in order to ensure that no negative example is covered by the generated *lgg*. This makes the generalization process heavily dependent on the number of examples. Since the system loads examples and theory into main memory, picking up examples and coverage test are simply obtained by calling the corresponding goals where goal proving is done automatically within a Prolog runtime system. Therefore it is not possible to access only the examples related to a certain clause, and every time a clause is being generalized, all the examples in the main memory must be considered instead of checking only those referred to the given clause. For instance, when a clause that covers some positive examples is generalized, it will obviously still cover those examples after generalization, so completeness of the theory is trivial and it becomes useless checking all positive examples in the main memory. However, the generalized clause could cover also other previous examples of the same concept, in addition to the new one, and remembering such examples could help to ensure that, after future refinements, each positive example has at least one clause covering it. As for negative examples checking, only those belonging to the same concept as the generalized clause should be tested and this could be obtained through a relational schema that links together clauses and examples that have in common the target concept.

The same problems arise when the system tries to specialize a clause that covers a negative example (Algorithm 3 shows the procedure). The consistency check scans all the available positive examples in main memory in order to check whether the specialized clause still covers the examples that were covered before specialization. In order to speed up such a verification, the coverage test could take advantage from a data organization allowing to link clauses to the examples they cover, so that once a clause is specialized, only the examples that were covered by the clause before being specialized need to be checked for consistency.

Algorithm 3

```
Procedure Specialize(E: negative example; T: theory; M⁺:positive
examples);
specialized := false ; gen_spec := 0
while ∃ a (new) derivation D of E from T do
      L := Input program clauses in D sorted by decreasing
           depth in derivation
```

```
        while ((¬ specialized) AND (∃C ∈ L) and
               (gen_spec ≤ maxgen)) do
               while ((∃ another S ∈ rho_OI_pos(C,E)) AND
                      (¬ specialized)) do
                      gen_spec + 1
                      if (complete({T\{C}} U {S},M+)) then
                             T := {T\{C}} U {S}
                      end if
               end while
        end while
        if (¬ specialized) then
               C := first clause in the derivation of E
               while ((∃ another S ∈ rho_OI_neg(C,E)) AND
                     (¬ specialized)) do
                      if (complete({T\{C}} U {S},M+)) then
                             T := {T\{C}} U {S}
                      end if
               end while
        end if
        if (¬ specialized) then
               T := T U {E} {negative exception}
        end if
end while
```

Inspired by these considerations, a possible integration of an external terms storage was taken into account in order to increase the system performance. Additionally, developing a more structured framework for handling data, it becomes possible to maintain and exploit during the learning process precious information about the examples and the clauses.

3. Scalable version of INTHELEX

This section presents the extension of INTHELEX with an external storage of terms. Instead of choosing a general DBMS for the external handling of terms, a tailored embedded solution was developed using the Sicstus Prolog interface [10] to the Berkeley DB toolset for supporting persistent storage of Prolog terms. 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.

Some commercial databases can't store non-ground terms or more than one instance of a term. This interface can however store terms of either kind.

3.1 Interface to Berkeley DB

The interface we exploit in this work uses the Concurrent Access Methods product of Berkeley DB [10]. This means that multiple processes can open the same database, but transactions and disaster recovery are not supported. The environment and the database files are ordinary Berkeley DB entities that use a custom hash function.

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.

3.2 Designing storage of terms

In order to relate clauses to examples, a simple relational schema was designed. 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. The following schema in Figure 2 shows how the logical model of the tables was designed. 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. For example, the attribute *type* permits to distinguish between the examples that have modified the theory and those that have not. This could be useful in a partial memory context, in which examples could be considered based on their history of modifications of the theory, thus during the learning task only the examples that have modified the theory could be considered. For this purpose, we also planned to 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 analyze 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.

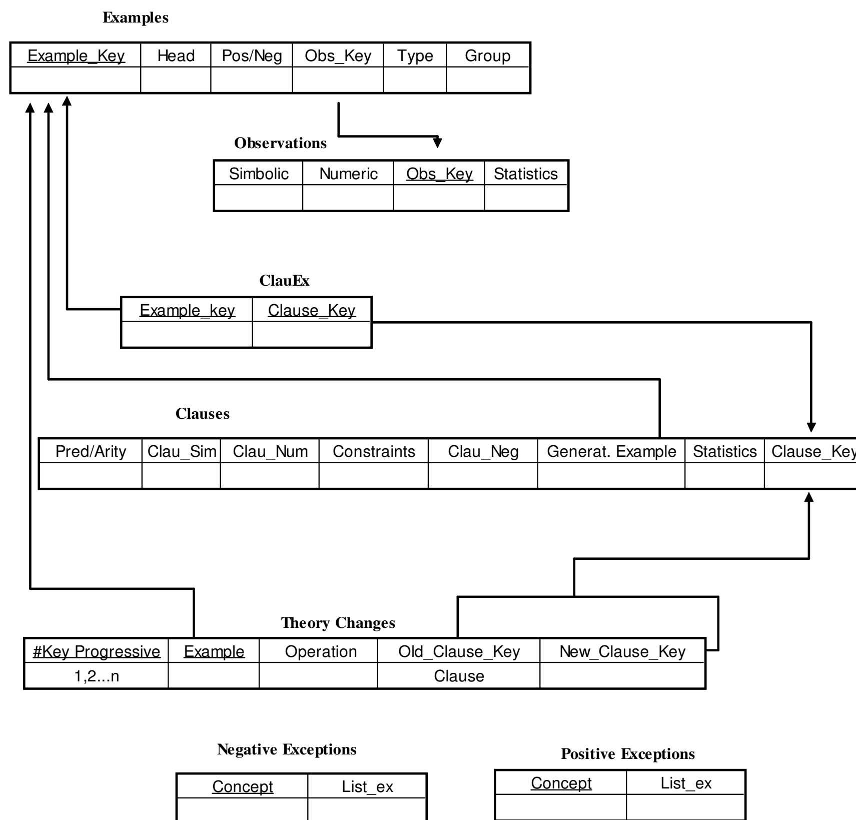

Figure 2. Logical model of the database of terms

Another similar information kept in the table *theory_changes* is the key of the example that have caused the refinement of a certain clause. This information could be exploited by another improved version of INTHELEX, named INTHELEX_back, [11], [12], able to handle the problem of ordering effects [13] in incremental learning.

The *db-spec* that resulted form the logical model was very simple. The following specifications show how the tables were implemented through the *db-spec* of Berkeley DB. 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 Berkeley DB 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.

4. Experimenting scalability

The framework proposed was tested on the domain of document processing. The exploited datasets belong to two learning tasks: document understanding and document classification. Both these tasks are quite hard to deal with, in terms of computational effort and we chose them because of the very high times of elaboration of the first version of INTHELEX in order to process the datasets involved. The datasets contain descriptions of documents formatted according to different conference and formatting styles, specifically ISMIS (the International Symposium on Methodologies for Intelligent Systems), TPAMI (Transactions on Pattern Analysis and Machine Intelligence) and ICML (the International Conference on Machine Learning) [7]. The task for the learning system is to learn theories able to classify the new incoming documents and to recognize the significant logical components for each class of document. In particular, the interesting logical components identified by the expert in the classes were: *title*, *abstract*, *author* and *page number* for the documents that belong to ICML; *title*, *abstract*, *author* and *affiliation* for ISMIS documents and *title*, *abstract*, *affiliation*, *index terms*, *page number* and *running head* for TPAMI. All the experiments were performed on a Pentium 4, 2,4 GHz, using Sicstus Prolog 3.11.1. The first experiments were conducted between the two versions of INTHELEX with database. In the first version nothing is changed but the simple fetching of terms. Whereas in the second version we consider during the refinement steps only examples covered by a clause during the clause refinement. Table 1 reports the results of these experiments. As it can be seen, the second version of INTHELEX that does not simply use the database as a fast-fetching mechanism but makes use also of the relational schema between clauses and examples has better results than the first one. We have reported here the mean time in seconds that resulted from the experiments on 33 folds.

Finally, experiments were performed between the initial version of INTHELEX that uses main memory for loading examples and clauses and is based on text files and the second version of INTHELEX with database. As it is possible to note in Table 2 there is a far better performance for the scalable version of INTHELEX. This is due to not only the relational schema that makes possible considering only the examples covered by a clause but can be attributed also to the fast access that the database engine guarantees comparing it with the main memory approach of Prolog. As regards the dataset dimension, the results showed that for large datasets the gain in elaboration time is considerable. For small datasets the scalable version has a slightly better performance. This enforces the idea that the scalable version is oriented to large datasets where the elaboration times are very high. As for the theories' accuracy in both experiments the accuracy of the theories was similar.

Table 1. Comparison of the two versions of INTHELEX with database

Class of documents	Initial version Inthelex-Berkeley Runtime (secs)	Second version Inthelex-Berkeley Runtime (secs)
Classification		
ICML	3.61	3.15
ISMIS	25.23	19.45
TPAMI	37.26	27.55
Understanding		
ICML logical components		
Abstract	167.86	111.13
Author	31.35	29.07
Page Number	77.65	76.22
Title	55.73	51.66
ISMIS logical components		
Abstract	173.99	133.70
Affiliation	65.80	50.94
Author	64.83	47.88
Title	54.11	33.46
TPAMI logical components		
Absctract	1276.09	1089.88
Affiliation	476.12	387.76
Page Number	1652.31	1265.47
Running Head	1161.31	1148.17
Title	1272.71	1234.43

5. Conclusions and future work

This paper described an approach for implementing scalable ILP systems. It is based on the idea that loading datasets in main memory is an operation which causes the ILP systems to have low performance on large datasets. For this reason it was

implemented a scalable version of the ILP system INTHELEX in order to work without loading data in main memory. There has been some work in this direction such as that in [3] which proposed sub-sampling as a possible approach for implementing scalability. The work presented here is based on the use of high performance database engine for storing terms. The results were very promising since for large datasets it was evident that the scalable version has better results in terms of elaboration time. Our aim was that of showing how the scalability extension of ILP systems is possible through the integration of valid engines for terms retrieval.

As future work we intend to use the information we stored during the learning task in order to guide the search for the hypotheses. This will render the system not only scalable for dealing with large datasets but will also provide a valid mechanism for improving the learning process through the use of information such as the different versions of theory and the modifying examples for each couple of clauses, the old and the refined one.

Table 2. Comparison between the original version of INTHELEX and the scalable one

Class of documents	INTHELEX Text Files Mean time (secs.) on 33 folds	INTHELEX BERKELEY DB Mean time (secs.) on 33 folds	Coefficient of time gain
Classification			
ICML	15.67 sec	3.15 sec	4,96
ISMIS	67.53 sec	19.45 sec	3,47
TPAMI	64.20 sec	27.55 sec	2.32
Understanding			
ICML logical components			
Abstract	131.54 sec	111.13 sec	1,18
Author	130.41 sec	29.07 sec	4,48
Page Number	270.23 sec	76.22 sec	3,54
Title	249.04 sec	51.66 sec	4,82
ISMIS logical components			
Abstract	178.75 sec	133.70 sec	1,33
Affiliation	129.50 sec	50.94 sec	2,54
Author	159.35 sec	47.88 sec	3,32
Title	65.52 sec	33.46 sec	1,95
TPAMI logical components			
Abstract	1687.72 sec	1089.88 sec	1,54
Affiliation	1155.10 sec	387.76 sec	2,97
Page Number	2669.03 sec	1265.47 sec	2,10
Running Head	1798.31 sec	1148.17 sec	1,56
Title	2027.57 sec	1234.43 sec	1,64

References

[1] A. Knobbe, H. Blockeel, A. Siebes and D. Van der Wallen, Multi-Relational Data Mining, *Technical Report INS-R9908*, Maastricht University, 9, 1999.

[2] H. Blockeel and M. Sebag, Scalability and efficiency in multi-relational data mining. *SIGKDD Explor. Newsl*, Vol 5, pp. 17-30, 2003.

[3] A. Srinivasan, A study of Two Sampling Methods for Analyzing Large Datasets with ILP, *Data Mining and Knowledge Discovery,* Vol. 3, pp. 95-123, 1999.

[4] H. Blockeel, L. De Raedt, N. Jacobs and B. Demoen. Scaling up Inductive Logic Programming by Learning from Interpretations. *Data Mining and Knowledge Discovery,* Vol. 3, No. 1, pp. 59-64, 1999.

[5] N. Di Mauro, T.M.A. Basile, S. Ferilli, F. Esposito and N. Fanizzi. An Exhaustive Matching Procedure for the Improvement of Learning Efficiency. In T. Horváth and A. Yamamoto, Inductive Logic Programming: 13th International Conference (ILP03), 2003, Vol. 2835, LNCS, pp. 112-129. Springer Verlag.

[6] J. Maloberti and M. Sebag. Fast Theta-Subsumption with Constraint Satisfaction Algorithms. Machine Learning, Vol 55, N. 2, pp. 137-174, Kluwer Accademic Publishers, 2004.

[7] F. Esposito, S. Ferilli, N. Fanizzi, T. M.A. Basile and N. Di Mauro, Incremental Multistrategy Learning for Document Processing, *Applied Artificial Intelligence: An International Journal*, 2003, Vol. 17, n. 8/9, pp. 859-883, Taylor Francis.

[8] Berkeley DB reference: http://www.sleepycat.com

[9] F. Esposito, S. Ferilli, N. Fanizzi, T.M.A Basile and N. Di Mauro. Incremental Learning and Concept Drift in INTHELEX. *Intelligent Data Analysis Journal, ISSN 1088-467X, Special Issue on Incremental Learning Systems Capable of Dealing with Concept Drift,* 8(3):213-237, IOS Press, Amsterdam, 2004.

[10] Sisctus Prolog reference www.sics.se/sicstus

[11] N. Di Mauro, F. Esposito, S. Ferilli and T.M.A. Basile. A Backtracking Strategy for Order-Independent Incremental Learning. In R. Lopez de Mantaras and L. Saitta (Eds), *Proceedings 16th European Conference on Artificial Intelligence* (ECAI 2004), 2004, pp. 460-464, IOS Press.

[12] N. Di Mauro, F. Esposito, S. Ferilli and T. M.A. Basile. Avoiding Order Effects in Incremental Learning. In S. Bandini and S. Manzoni (Eds.), *Advances in Artificial Intelligence* (AI*IA05), 2005 LNCS, Vol. 3673, pp. 110-121, Springer-Verlag

[13] P. Langley. Order effects in incremental learning. In: P. Reimann, H. Spada (Eds.), *Learning in Humans and Machines: Towards an Interdisciplinary Learning Science*, Elsevier, Amsterdam, 1995.