Mach Learn (2006) 64:149–182
DOI 10.1007/s10994-006-8713-9

First order random forests: Learning relational classifiers with complex aggregates

Anneleen Van Assche · Celine Vens · Hendrik Blockeel · Sašo Džeroski

Received: 8 April 2005 / Revised: 24 February 2006 / Accepted: 29 March 2006 / Published online: 21 June 2006
Springer Science + Business Media, LLC 2006

Abstract In relational learning, predictions for an individual are based not only on its own properties but also on the properties of a set of related individuals. Relational classifiers differ with respect to how they handle these sets: some use properties of the set as a whole (using aggregation), some refer to properties of specific individuals of the set, however, most classifiers do not combine both. This imposes an undesirable bias on these learners. This article describes a learning approach that avoids this bias, using first order random forests. Essentially, an ensemble of decision trees is constructed in which tests are first order logic queries. These queries may contain aggregate functions, the argument of which may again be a first order logic query. The introduction of aggregate functions in first order logic, as well as upgrading the forest's uniform feature sampling procedure to the space of first order logic, generates a number of complications. We address these and propose a solution for them. The resulting first order random forest induction algorithm has been implemented and integrated in the ACE-ilProlog system, and experimentally evaluated on a variety of datasets. The results indicate that first order random forests with complex aggregates are an efficient and effective approach towards learning relational classifiers that involve aggregates over complex selections.

Editor: Rui Camacho

A. Van Assche (✉) · C. Vens · H. Blockeel
Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium
e-mail: anneleen.vanassche@cs.kuleuven.be

C. Vens
e-mail: celine.vens@cs.kuleuven.be

H. Blockeel
e-mail: hendrik.blockeel@cs.kuleuven.be

S. Džeroski
Department of Knowledge Technologies, Jožef Stefan Institute, Jamova 39, 1000 Ljubljana, Slovenia
e-mail: saso.dzeroski@ijs.si

Keywords Relational learning · Random forests · Aggregation · Decision tree learning

1. Introduction

In relational learning, an individual to be classified may be linked to a set of other objects. Properties of this set, or of some of the objects it contains (or perhaps both) may be relevant for the classification. Among the many approaches to relational learning that currently exist, an important distinction can be made with respect to how they handle these one-to-many and many-to-many relationships. Whereas propositionalisation approaches usually handle sets by aggregating over them, inductive logic programming (ILP) (Muggleton, 1992) techniques select specific elements. This imposes an undesirable bias on both kinds of learners (Blockeel & Bruynooghe, 2003).

A combination of both would involve aggregating over a subset of elements fulfilling specific conditions ("aggregating over a selection"). Such combinations might be expected to naturally appear in certain patterns, but they are very difficult to construct for machine learning systems, both because the feature space explodes and because it becomes more difficult to search it in a structured and efficient way due to, e.g., non-monotonicity (Knobbe, Siebes & Marseille, 2002).

In this article we present an approach based on random forests (Breiman, 2001) for combining aggregates and selections. Random forest induction is a bagging method that builds decision trees using only a random subset of the feature set at each node. In our approach, the decision trees that are constructed contain tests with first order logic queries that may involve aggregate functions. The argument of these aggregate functions may again be a first order logic query.

The motivation for using random forests is based on two observations. First, random forests seem to be very suitable to tackle the problems mentioned above. For example, the problem of feature space explosion is handled by sampling the feature space for good features, instead of exhaustively searching it, and by combining multiple trees built using such random sampling. Also, because of the search strategy used by decision trees, problems regarding an unstructured feature space disappear. Second, in propositional learning, random forest induction has been shown to work well when many features are available (Breiman, 2001). Because ILP typically deals with large feature sets, it seems worthwhile to investigate whether random forests perform well in this context.

The paper is organised as follows. Section 2 illustrates the problem of combining aggregates with selection conditions in ILP. Section 3 discusses how aggregate functions can be included in the search space of first order decision trees. Section 4 describes random forests and how they can be upgraded to the first order case. In Section 5 we experimentally evaluate our method on some well-known real world data sets and on an artificially generated data set. Section 6 discusses related work. Finally, we formulate conclusions and some ideas for future research in Section 7.

2. Aggregates and selection in ILP

In this section, we discuss the combination of aggregates and selection in ILP. We start by stating the problem and describing a few attempts at combining selection and aggregation. Then, we give some definitions that we use in a technical discussion afterwards. Finally, we discuss the complexity of finding patterns that combine aggregation and selection.

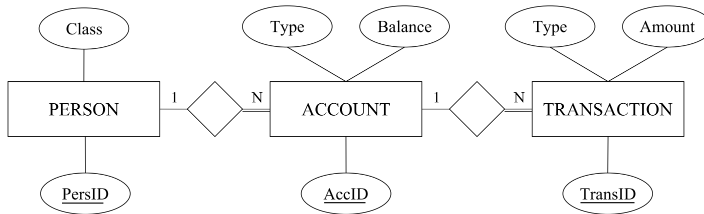

Fig. 1 ER-diagram of the *Account* example

2.1. Aggregates and selection

As mentioned in the introduction, in relational learning an individual may be linked to a set of other objects, via one-to-many or many-to-many relationships. Properties of either the set itself or of some of the objects it contains may be relevant for the classification.

Current relational learners either use aggregate functions, which reflect properties of the set as a whole, or build patterns that express the existence of one or more elements with specific properties in that set (Blockeel & Bruynooghe, 2003). They usually do not combine both. Such a combination would involve aggregating over a subset of elements fulfilling specific conditions ("aggregating over a selection"). For example, in the context of the database shown in Fig. 1, a relevant criterion to determine whether a person is, for example, a good customer, could be related to the sum *(aggregation)* of the balance on his savings *(selection)* accounts.

In terms of relational algebra, combining aggregation and selection boils down to constructing features of the form $\mathcal{F}(\sigma_C(R))$ with $\mathcal{F}$ an aggregate function, $\sigma_C(R)$ a selection function based on a condition C, and R a set of tuples somehow connected to the tuple we want to classify. From this viewpoint, ILP systems (inductive logic programming, (Muggleton, 1992)) typically build a complicated selection condition C but the aggregate function $\mathcal{F}$ is always the "there exists" function, returning *true* if at least one element of R fulfills C, and *false* otherwise. Other relational learners use features of the form $\mathcal{F}(R)$, where $\mathcal{F}$ is taken from a predefined set of aggregate functions. Examples of this approach include probabilistic relational models (Koller, 1999) and relational probability trees (Neville et al., 2003).

A few attempts at combining selection and aggregation exist. Knobbe, Siebes & Marseille (2002) propose a method for aggregating over selections. The patterns they learn are represented as selection graphs. They can perform a general-to-specific search through a hypothesis space, at the cost of disallowing refinement of certain aggregate functions (so-called non-monotone functions, see further). Krogel and Wrobel (2003) propose a propositionalisation approach, called RELAGGS, where features can be aggregates over selections, but the selection conditions have a limited complexity and are not refined during the search. Uwents and Blockeel (2005) present a non-symbolic approach towards combining aggregates and selection. Their method is not constrained to using predefined aggregate functions. They train a so-called relational neural network that reflects the structure of the relational database. Of course, as in all non-symbolic approaches, the obtained model is not easily interpretable.

Perlich and Provost (2003) present a hierarchy of relational concept classes in order of increasing complexity, where the complexity depends on that of any aggregate functions used. Their "category 3 and 4" concepts rely on so-called multi-dimensional aggregation, which corresponds to what we call "combining selection and aggregation". Perlich and Provost's analysis is very similar to ours: They point out that ILP-like systems are the only ones that can handle concepts of categories 3 or 4, but they have the disadvantage that they typically do

not use numerical aggregation, and the latter is identified as a crucial weakness. It is exactly that weakness that we try to eliminate with this work.

An important reason why few current systems combine aggregation and selection is that this combination causes a substantial expansion of the feature space and at the same time makes it more difficult to search that space in a structured and efficient way. In the following, we will analyse the difficulties that arise with aggregation and selection in the context of first order logic. We start with some terminology and definitions, which will be useful for describing exactly what complications arise when learning first order logic clauses with arbitrarily complicated aggregates.

2.2. Preliminaries

This section reviews refinement under theta-subsumption, coverage, and monotonicity, and adapts these concepts to our context where needed.

2.2.1. θ-subsumption for clauses with aggregates

Many ILP systems perform a general-to-specific search by repeatedly refining an overly general clause until its quality is sufficient. For this refinement, ILP systems typically use a refinement operator based on θ-subsumption. The classical definition of the θ-subsumption relation between clauses is as follows (Plotkin, 1969).

Definition 1. A clause c_1 θ-subsumes a clause c_2 (denoted $c_1 \leq_\theta c_2$) if there exists a substitution θ such that $c_1\theta \subseteq c_2$.

The kind of clauses that we are interested in are not pure logical clauses: they may contain aggregate literals of the form $F(V, Q, R)$ where F is an aggregate function (e.g., count), V is an *aggregate variable* occurring in the *aggregate query* Q, and R is the result of applying F to the set of all answer substitutions for V that Q results in (we will call this set the *result set* of Q). We will use the term *standard clauses* to refer to clauses without aggregates.

As we are interested in learning classifiers with extended aggregate queries, it is useful to be able to gradually refine aggregate queries, in a similar way as standard clauses would be refined. We therefore define an extension of the classical θ-subsumption relation on clauses with aggregation, and call it A-subsumption. A *standard literal* is any literal in a clause that is not an aggregate literal and does not occur inside an aggregate query. The *standard part* of a clause c, denoted $S(c)$, is the clause consisting of all standard literals of c (and only those).

Definition 2. A clause c_1 A-subsumes a clause c_2 (denoted $c_1 \leq_A c_2$) if and only if $S(c_1) \leq_\theta S(c_2)$, and for each aggregate literal $F(V_1, Q_1, R_1) \in c_1$, there exists an aggregate literal $F(V_2, Q_2, R_2) \in c_2$ such that $Q_1\theta \leq_A Q_2$, and the latter A-subsumption only involves a set of substitutions σ over locally defined variables in Q_1, such that $V_1\sigma = V_2$.

In other words, a clause A-subsumes another clause if, after applying the right variable substitutions to the standard part of the clause as well as to its aggregate queries, the standard part of the first clause becomes a subset of the standard part of the second clause, and each aggregate query becomes a subset of the corresponding aggregate query in the second clause.

2.2.2. Specialisation and generalisation

We define the concepts of generalisation and specialisation of a clause in terms of its coverage. Generally, we say that a clause *covers* an example if the body of the clause succeeds for the example. Clause c_1 is a *specialisation* (*generalisation*) of c_2 if and only if the set of examples it covers is a subset (superset) of the examples covered by c_2.

For clauses without aggregates, it holds that whenever $c_1 \leq_A c_2$ (or equivalently, $c_1 \leq_\theta c_2$), the coverage of c_1 must be a superset of the coverage of c_2, i.e., c_2 is a specialisation of c_1. We will see later that for clauses with aggregates and the A-subsumption relation, this property is lost.

We will call a refinement *valid* if and only if it constitutes a specialisation.

2.2.3. Monotonicity

We will also use the concept of monotonicity. A condition on a set is monotone if and only if whenever the condition holds on a set S, it holds also on all supersets of S. It is anti-monotone if and only if whenever it holds on S, it holds also on all subsets of S. It is non-monotone if it is neither monotone nor anti-monotone. The concept can easily be extended towards bags (multi-sets), using definitions of subbag and superbag as follows: A is a subbag of B (denoted $A \subseteq B$) if and only if each element of A is also in B and its multiplicity in B is at least as high as in A; and A is a superbag of B if and only if B is a subbag of A.

In the context of aggregate functions, the concept of monotonicity can be specialised to the following definitions, which we borrow from Knobbe, Siebes and Marseille (2002).

Definition 3. An *aggregate condition* is a pair (f, o) where f is an aggregate function and o is a comparison operator.

Definition 4. An aggregate condition (f, o) is *monotone* if and only if for any sets (bags) S and S' such that $S' \subseteq S$, and for any value v in the range of f, $f(S')\ o\ v \Rightarrow f(S)\ o\ v$.

Definition 5. An aggregate condition (f, o) is *anti-monotone* if and only if for any sets (bags) S and S' such that $S' \subseteq S$, and for any value v in the range of f, $f(S)\ o\ v \Rightarrow f(S')\ o\ v$.

Definition 6. An aggregate condition is *non-monotone* if and only if it is neither monotone nor anti-monotone.

It is easy to see that the following aggregate conditions are monotone: $(count, \geq)$, $(max, \geq)$, and $(min, \leq)$. Anti-monotone conditions are $(count, \leq)$, $(max, \leq)$ and $(min, \geq)$. The aggregate conditions $(mode, =)$, $(sum, \leq)$, $(sum, \geq)$, $(avg, \geq)$ and $(avg, \leq)$ are non-monotone.[1]

2.3. Refining clauses with aggregates

We can now discuss in what way refinement of clauses with aggregates is more complicated than refinement of standard clauses.

[1] The aggregate functions $count$, max, min, sum, and avg are defined as in SQL. The function $mode$ returns the most frequent value of a set.

In the context of standard clauses, whenever a clause c_1 θ-subsumes a clause c_2, c_2 must be a specialisation of c_1. Hence, by using a refinement operator ρ that, given a clause c, yields only clauses θ-subsumed by c, a general-to-specific search through the hypothesis space is obtained. Such a refinement operator typically employs one of the following basic operations on a clause:

- apply a substitution to the clause, and
- add a literal to the body of the clause.

Our definition of A-subsumption is syntactically very similar to the original θ-subsumption definition, but the property that $c_1 \leq_A c_2$ implies that c_2 is a specialisation of c_1, is lost. Refinement under A-subsumption may yield a specialisation or a generalisation, or even none of both.

To some extent, this behaviour can be related to the monotonicity properties of the aggregate conditions, an observation also made by Knobbe, Siebes and Marseille (2002) in the context of refining selection graphs. But it turns out that in the first order logic context, monotonicity is not the only factor. Also the semantics of the aggregate conditions plays a role. We discuss these two issues in more detail.

Take the following example clause (see Fig. 1 for the *Account* database).

$person(PersID, pos) \leftarrow$
$count(AccID, account(PersID, AccID, Type, Balance), C), C \geq 4$

Applying a substitution to the aggregate query gives the following refinement

$person(PersID, pos) \leftarrow$
$count(AccID, account(PersID, AccID, savings, Balance), C), C \geq 4$

which must have at most the same coverage (people with at least four savings accounts must be a subset of people with at least four accounts), so the refinement is valid.

However, if we consider the following query

$person(PersID, pos) \leftarrow$
$count(AccID, account(PersID, AccID, Type, Balance), C), C < 4$

and its refinement

$person(PersID, pos) \leftarrow$
$count(AccID, account(PersID, AccID, savings, Balance), C), C < 4$

then the result yields a generalisation (people with less than four savings accounts may have more than four accounts).

The fact that the refinement operator yields a specialisation in the first case, and a generalisation in the second case, is related to the first clause having a monotone aggregate condition, and the second clause having an anti-monotone one. Indeed, applying a substitution to the aggregate query causes the count to go down, which may cause the $C \geq 4$ condition to fail when it was true for the original clause (hence, we obtain a more specific clause), and may cause the $C < 4$ condition to become true when it was false for the original clause (hence, we obtain a more general clause).

The situation becomes more complex, however, when we add literals to the aggregate query (instead of just applying substitutions). For example, consider the following refinement:

Table 1 Extension of the *Account* and *Transaction* tables of the *Account* database

Account	PersID	AccID	Type	Balance
	John	123456	Checkings	100
	John	987654	Checkings	200
	John	789123	Savings	200
Transaction	**Acc**	**TransID**	**Type**	**Amount**
	123456	tr090	Withdrawal	50
	123456	tr091	Deposit	30
	987654	tr098	Deposit	70
	789123	tr100	Withdrawal	100
	789123	tr101	Deposit	80

$$\begin{aligned}&person(PersID, pos) \leftarrow\\&count(AccID, (account(PersID, AccID, Type, Balance),\\&\qquad transaction(AccID, TransID, Type, Amount)),\\&\quad C), C < 4\end{aligned}$$

The count aggregate function counts the number of times the aggregate query succeeds, which may be larger than the number of accounts if there are multiple transactions per account, or smaller if some accounts have no transactions. In other words, this refinement may lead to generalisation or specialisation, or none of both, even though the aggregate condition is anti-monotone. The reason for this is that the count function computes the cardinality of the bag, rather than the set, of *AccID* values returned by the aggregate query. While the set of *AccID* values returned by the refined query must be a subset of the set returned by the original one, the bag of *AccID* values returned by the refined query is not guaranteed to be a subbag or superbag of the original one: some accounts may have disappeared, others may have been duplicated.

The situation is similar to what one would get with a SQL query for relational databases along the lines of

```
SELECT COUNT(A.AccID)
FROM account AS A, transaction AS T
WHERE A.AccID = T.AccID
```

A solution in the relational database case is to use the COUNT DISTINCT construct. This is semantically meaningful if *AccID* is a key attribute for the *Account* relation.

In general, there are thus two possible outcomes for an aggregate function: applying the function to the bag or to the set of variable substitutions returned by the aggregate query. For example, consider the simple extension of the *Account* database in Table 1. The query

$$\begin{aligned}&sum(Balance, (account(PersID, AccID, Type, Balance),\\&\qquad transaction(AccID, TransID, Type, Amount)), C)\end{aligned}$$

leads to two possible results:

- $100 + 100 + 200 + 200 + 200 = 800$ (when aggregating over the bag of balances)
- $100 + 200 = 300$ (when aggregating over the set of balances).

However, taking the set over the balance values is often not intuitive. Instead, one most likely wants to take the set of the account objects and then take the sum of the balances. More generally, this corresponds to taking the set over the first predicate in the aggregate

query, and applying the aggregate function to the corresponding aggregate variable values. This leads to a third possible result:

– $100 + 200 + 200 = 500$ (when aggregating over the set of accounts)

These solutions correspond to the following queries in relational algebra (where $\star$ denotes the natural join operator):

– $\mathcal{F}_{SUM(Balance)}(account \star transaction)$
– $\mathcal{F}_{SUM(Balance)}(\pi_{Balance}(account \star transaction))$
– $\mathcal{F}_{SUM(Balance)}(\pi_{AccountId,Balance}(account \star transaction))$

In our system, we chose not to include the second interpretation (note that when aggregating over a key attribute, the second and third interpretation are the same). Next to aggregating over the bag of balances (or equivalently, the bag of accounts), we provide the possibility to aggregate over the set of accounts, the latter being consistent with the way of executing aggregates by Knobbe, Siebes and Marseille (2002). Therefore, we introduce in our approach a number of "distinct" versions of aggregate functions, which correspond to this set semantics: *count_dist, avg_dist, sum_dist*, and *mode_dist*. As the minimum (maximum) over a set is the same as the minimum (maximum) over a bag defined over the set, there is no need to have a *min_dist* (*max_dist*).

We can summarise all this as follows. (Anti-)monotonicity guarantees that a condition that is true for some set of examples is also true for its supersets (subsets). Refining an aggregate query under A-subsumption causes the result *set* of the aggregate query to decrease (to become a subset of what it originally was), but may have any effect on the result *bag* of the aggregate query. Hence, *for the set semantics* we can say that refining an aggregate query inside a (anti-)monotone aggregate condition can only yield a specialisation (generalisation), but *for the bag semantics* no such statements are possible.

A consequence of this is that searching the hypothesis space in a general-to-specific manner becomes more complicated. There is no obvious refinement strategy for aggregate queries that guarantees that the refinement will yield a specialisation, unless we limit the hypothesis space to patterns involving refinements of monotone aggregate functions only, and use a set semantics. This is essentially what Knobbe, Siebes and Marseille (2002) do.

In our system, we choose not to exclude any aggregate conditions from being refined, but instead use a search method that ensures valid refinements. Our approach is based on decision trees. While rule induction is more common in ILP than tree induction, ILP tree learners have been around for several years now. Because of the divide and conquer search strategy used by tree induction methods, a refined aggregate condition can not become true when the original condition failed. Hence, bag-defined aggregate conditions can only yield valid refinements. Considering the aggregate functions defined over sets, we know that anti-monotone aggregate conditions will always yield refinements with the same coverage as the original condition, and as such, these refinements will never be chosen by a decision tree inducer. Discarding these anti-monotone conditions leads to a reduction in size of the search space, while its expressiveness remains the same. Therefore, we can dispose of the following aggregate conditions: *(count_dist,*$\leq$*), (max,*$\leq$*), (min,*$\geq$*)*.[2] The aggregate conditions that are used in our system are listed in Table 2.

[2] In decision trees, to dispose of (max,$\leq$) and (min,$\geq$) while being equally expressive, $\leq$ and $\geq$ need to be equivalent up to switching branches. Therefore, it is necessary to define the aggregate functions over empty sets. We define $min(\emptyset) = +\infty$ and $max(\emptyset) = -\infty$.

Table 2 Aggregate conditions included in our system. For each condition, the monotonicity is given

		bag-based		set-based	
aggr. cond.	mon.	aggr. cond.	mon.	aggr. cond.	mon.
(min,≤)	mon.	(count,≥)	mon.	(count_dist,≥)	mon.
(max,≥)	mon.	(count,≤)	anti-mon.	(avg_dist,≤)	non-mon.
		(avg,≤)	non-mon.	(avg_dist,≥)	non-mon.
		(avg,≥)	non-mon.	(sum_dist,≤)	non-mon.
		(sum,≤)	non-mon.	(sum_dist,≥)	non-mon.
		(sum,≥)	non-mon.	(mode_dist,=)	non-mon.
		(mode,=)	non-mon.		

2.4. The complexity of finding patterns that combine aggregation and selection

ILP systems explore large search spaces. They often do this in a greedy manner: from the current best clause, they generate a number of refinements, take the best among these, and continue the process. The computational complexity of this process depends on the branching factor of the search (how many refinements are generated from a clause).

By introducing aggregates in clauses, and allowing the aggregate queries to be refined as well, the branching factor is multiplied. Assume that a standard clause can be refined in C ways by adding a standard literal to it, and we now also allow the addition of an aggregate literal with any of those C literals as aggregate query, and any variable occurring in the new literal as the aggregate variable. If the new literal has V variables and we consider N possible aggregate functions, the branching factor increases with $V \cdot N \cdot C$. The multiplication factor $V \cdot N$ can easily be one or two orders of magnitude (Table 2 already lists 15 aggregate functions). Greedy searches slow down with the same factor.

A natural way to avoid the explosion of the feature space in a decision tree context is to use random forests. Random forests are collections of trees, where each tree has been built by considering in each node only a random sample of the possible tests for that node. In our ILP setting, this boils down to making a random selection of the refinements of a query. This compensates for the increase of the branching factor. Hence, the exploration of first order random forests as a means of learning classifiers with aggregations seems a natural choice.

3. First order decision trees with complex aggregates

This section describes how selection and aggregation are combined in first order decision trees. First, an introduction to first order decision trees is given (Section 3.1). Next, we continue by explaining how (complex) aggregate conditions are added to the feature space (Section 3.2).

3.1. First order decision trees

We consider the use of aggregates in TILDE (Blockeel & De Raedt, 1998), which is included in the ACE-ilProlog data mining system (Blockeel et al., 2002). TILDE is a relational top-down induction of decision trees (TDIDT) instantiation, and outputs a first order decision tree.

Table 3 TILDE algorithm for first order logical decision tree induction (Blockeel & De Raedt, 1998)

```
procedure GROW_TREE (E: examples, Q: query):
    candidates := ρ(← Q)
    ← Q_b := OPTIMAL_SPLIT(candidates, E)
    if STOP_CRIT (←Q_b, E)
    then
        K := PREDICT(E)
        return leaf(K)
    else
        conj := Q_b − Q
        E_1 := {e ∈ E|←Q_b succeeds in e ∧ B}
        E_2 := {e ∈ E|←Q_b fails in e ∧ B}
        left := GROW_TREE (E_1, Q_b)
        right := GROW_TREE (E_2, Q)
        return node(conj, left, right)
```

A first order decision tree (Blockeel & De Raedt, 1998) is a binary decision tree that contains conjunctions of first order literals in the internal nodes. Classification with a first order tree is similar to classification with a propositional decision tree: a new instance is sorted down the tree. If the conjunction in a given node succeeds (fails), the instance is propagated to the left (right) subtree. The predicted class corresponds to the label of the leaf node where the instance arrives. A given node n of the tree may introduce variables that can be reused in the nodes of its left subtree, which contains the examples for which the conjunction in n succeeds (with certain bindings for these variables).

In TILDE, first order decision trees are learned with a divide and conquer algorithm similar to C4.5 (Quinlan, 1993). The main point where it differs from propositional tree learners is the computation of the set of tests to be considered at a node. The algorithm to learn a first order decision tree is given in Table 3.

The OPTIMAL_SPLIT procedure returns a query Q_b, which is selected from a set of candidates generated by the refinement operator ρ, by using a heuristic, such as information gain for classification problems, or variance reduction for regression. The refinement operator typically operates under θ-subsumption and generates candidates by extending the current query Q (the conjunction of all succeeding tests from the root to the leaf that is to be extended) with a number of new literals that are specified in the language bias.[3] The conjunction put in the node consists of $Q_b - Q$, i.e., the literals that have been added to Q in order to produce Q_b. In the left branch, Q_b will be further refined, while in the right branch Q is to be refined. When the stop criterion holds (typically, this is when a predefined minimum number of examples is reached), a leaf is built. The PREDICT procedure returns the most frequent class of the examples in E in case of classification, or the mean target value in case of regression.

3.2. First order decision trees with complex aggregates

TILDE was modified to include (complex) aggregate conditions. The feature set considered at each node in the tree was expanded to consist of the original features, augmented with aggregate conditions (both simple and complex ones). A *simple* aggregate condition is an aggregate that is constructed directly from the language bias, without having selection conditions. In

[3] See Section 3.2 for more details about the language bias.

Table 4 Language bias for the *Account* example. The arguments of the *account* and *transaction* relations correspond to the attributes in the *Account* and *Transaction* tables specified in Table 1. Lists of constants can be provided by the user, or can be generated using a discretization procedure

```
% prediction
predict(person(+persid,-class)).

% types
typed_language(yes).
type(account(persid, accid, acctype, balance)).
type(transaction(accid, trans, transtype, amount)).
type(person(persid,class)).

% rmodes
rmode(account(+PersID, -AccID, -Tp, -Bal)).
rmode(account(+PersID,+-AccID, #["savings","checkings"],-Bal)).
rmode((account(+PersID, +-AccID, -Tp,Bal), Bal ≤ #[500,2000,5000,10000])).
rmode(transaction(+AccID, -Tr, -Tp, -Am)).
rmode(transaction(+AccID, +-Tr,#["deposit","withdrawal"], -Am)).
rmode((transaction(+AccID, +-Tr, -Tp, Am), Am ≤ #[500,1000,2000])).

% aggregates
aggcondition([max], account(+PersID, -AccID, -Tp, -Bal), Bal, [≥],[2000,5000]).
aggcondition([min], account(+PersID, -AccID, -Tp, -Bal), Bal, [≤],[-100,0,100]).
aggcondition([sum], account(+PersID, -AccID, -Tp, -Bal), Bal, [≥,≤],[0,5000]).
aggcondition([count_dist],account(+PersID, -AccID, -Tp, -Bal), AccID, [≥],[2,5]).
aggcondition([count_dist],transaction(+AccID, -Tr, -Tp, -Am), Tr, [≥],[5,10,50]).
aggcondition([mode_dist],transaction(+AccID, -Tr, -Tp, -Am), Tp, [=],
    ["deposit","withdrawal"]).
```

terms of relational algebra, it would be denoted by $\mathcal{F}(R)$, with $\mathcal{F}$ an aggregate function and R a set of tuples connected to the tuple under consideration. By *complex* aggregate conditions, we mean aggregate conditions that have been refined with selection conditions. In relational algebra, these would be expressed as $\mathcal{F}(\sigma_C(R))$ with $\sigma_C(R)$ a non-empty selection condition on R. It is practically impossible to declare the complex aggregate conditions as intensional background knowledge, if the relevant ones are not known in advance. The main difficulty is that the aggregate queries themselves are the result of a search through some hypothesis space, hence we want to learn them.

To illustrate how our method works, an example language bias for the *Account* example is given in Table 4. The language bias consists of two important constructs, namely *rmode* and *aggcondition*, to specify the candidates that can be generated. In the language bias, '$+$' in front of a variable means that the variable is an input variable; i.e., it has to be bound when adding this literal. To this end, the variable is unified with a variable already occurring in the query. '$-$' means this is a new variable; no unification is performed with already existing variables (though variables that are introduced later on may be unified with this variable). '$+-$' means that the variable can, but does not need to be bound (unification with other variables is possible but not mandatory). In the rmodes the #-sign is always a placeholder for a constant. When it is followed by a list, the # symbol will be replaced by one element of the list. When it stands alone, it will be replaced by any constant that appears in that place

anywhere in the data. To include aggregate conditions next to the original features, the user needs to specify the basic ingredients in the *aggcondition* construct: the aggregate functions, the aggregate query, aggregate variables, and comparison operators. A number of values to compare the result with can be provided by the user, or can be obtained using discretization (Blockeel & De Raedt, 1997).

The system then constructs simple aggregate conditions, using these components. The refinement operator ρ includes the aggregate conditions in the set of candidate queries it generates. A simple aggregate condition that will be generated from the first *aggcondition* construct in Table 4 is for instance:

$$\begin{aligned}&max(Balance, account(PersID, AccID, Type, Balance), Max),\\&\quad Max \geq 2000,\end{aligned}$$

with *PersID* bound to *PersID* in *person*(*PersID*, *Class*). This query states that the maximum balance of the accounts of a person exceeds 2000.

When also considering complex aggregates, a local search has to be conducted within the aggregate condition. Therefore, ρ constructs an inner refinement operator ρ_{inn}, which generates candidates by extending the current aggregate query with all features specified in either the *rmode* or *aggcondition* constructs. Each candidate generated by ρ_{inn} is included in an aggregate condition, which is then considered as a candidate of ρ. This adapted operator ρ is now operating under A-subsumption. Note that it allows to refine an aggregate condition with an aggregate condition. An example is given by the following query

$$\begin{aligned}&max(Bal, (account(PersID, AccID, AType, Balance),\\&\qquad count_dist(TransID,\\&\qquad\qquad transaction(AccID, TransID, TType, Am),\\&\qquad\qquad Cnt), Cnt \geq 5),\\&\quad Max), Max \geq 2000,\end{aligned}$$

which states that the maximum balance of the accounts that have more than 5 transactions associated, exceeds 2000.

There are two ways to use complex aggregate functions. The first one is to refine a (simple or complex) aggregate condition, occurring in the current query Q. For example, if the current query at a given node n is

$$person(P, Cl), count_dist(A, account(P, A, Tp, B), C), C \geq 5$$

then one of the refinements generated by ρ might for example be

$$\begin{aligned}&person(P, Cl), count_dist(A, account(P, A, Tp, B), C), C \geq 5,\\&count_dist(AccID, (account(P, A, Tp, B),\\&\qquad transaction(A, Tr, TrTp, Am)), C'), C' \geq 5\end{aligned}$$

This query states that a person has at least five accounts that have a transaction associated with it. If the query above is chosen by the OPTIMAL_SPLIT procedure, then

$$\begin{aligned}&count_dist(A, (account(P, A, Tp, B),\\&\qquad transaction(A, Tr, TrTp, Am)), C'), C' \geq 5\end{aligned}$$

is the conjunction added in the left child node of n.

The second way to build complex aggregates is based on lookahead (Blockeel & De Raedt, 1997), a technique commonly used in ILP to make the learner look ahead in the refinement

lattice. In most cases, the refinement operator ρ adds only one literal (i.e., the new node contains only one literal—not a conjunction). In some cases, however, it is interesting to add more literals at once, e.g., if the first literal yields no gain, but introduces interesting variables that can be used by other literals. If the refinement operator adds up to $k + 1$ literals, one says that it performs a lookahead of depth k. We extend this mechanism to be directly applied to the aggregate query. When the aggregate lookahead setting is turned on in the language bias description, ρ_{inn} is called and aggregate queries are built with up to a predefined depth of literals. This way, the query

$$count_dist(A, (account(P, A, Tp, B), \\ transaction(A, Tr, TrTp, Am)), C'), C' \geq 5$$

could immediately be inserted, without having the query

$$count_dist(AccID, account(P, A, Tp, B), C), C \geq 5$$

in one of its ancestor nodes. Obviously, this technique is computationally expensive, but it may yield significant improvements.

4. First order random forests with complex aggregates

4.1. Random forests

Random forest induction (Breiman, 2001) is an ensemble method. An ensemble learning algorithm constructs a set of classifiers, and then classifies new data points by combining the predictions of each classifier. A necessary and sufficient condition for an ensemble of classifiers to be more accurate than each of its individual members, is that the classifiers are accurate and diverse (Hansen & Salamon, 1990). An accurate classifier does better than random guessing on new examples. Two classifiers are diverse if they make different errors on new data points.

There are different ways to construct ensembles: bagging (Breiman, 1996a) and boosting (Freund & Schapire, 1996) for instance, introduce diversity by manipulating the training set. Several other approaches attempt to increase variability by manipulating the input features or the output targets, or by introducing randomness in the learning algorithm (Dietterich, 2000).

Random forests increase diversity among the classifiers by changing the feature sets over the different tree induction processes, and additionally by resampling the data. The exact procedure to build a forest with k trees is as follows:

– **for** $i = 1$ **to** k **do**:

- build training set D_i by sampling (with replacement) from data set D
- learn a decision tree T_i from D_i *using randomly restricted feature sets*

The part of the algorithm where random forests differ from the normal bagging procedure is emphasized. Normally, when inducing a decision tree, the best feature is selected from a fixed set of features F in each node. In bagging, this set of features does not vary over the different runs of the induction procedure. In random forests however, a different random subset of size $f(|F|)$ is considered at each node (e.g., $f(x) = 0.1x$ or $f(x) = \sqrt{x}, \ldots$), and the best feature from this subset is chosen. This obviously increases variability. Assume for instance that $f(x) = \sqrt{x}$, and that two tests t_1 and t_2 are both good features for the root of each tree,

say t_1 is the best and t_2 is the second best feature on all the training bags considered. With a regular bagging approach t_1 is consistently selected for the root, whereas with random forests both t_1 and t_2 will occur in the root nodes of the different trees, with frequency $1/\sqrt{|F|}$ and $1/\sqrt{|F|} - 1/|F|$ respectively. Thus t_2 will occur with a frequency only slightly lower than t_1.

Consider now a classification problem where a new example is to be assigned one of the m possible classes $(\omega_1, \ldots, \omega_m)$. Each decision tree T_i from the random forest gives a class label C_i to the new example. The label given by the random forest to the new example will then be

$$C^* = \arg\max_{\omega_j} \sum_{i=1}^{k} I(C_i = \omega_j)$$

where $I(x) = 1$ if x is true and $I(x) = 0$ otherwise. Hence the majority vote of the predicted class labels of the set of k trees in the random forest is the label predicted.

An advantage of using bagging is that out-of-bag error estimates (Breiman, 1996b) can be used to estimate the generalisation errors. This removes the need for a set-aside test set or cross-validation. Out-of-bag error estimation proceeds as follows: each tree is learned on a training set D_i drawn with replacement from the original training set D. For each example d in the original training set, the predictions are aggregated only over those classifiers T_i for which D_i does not contain d. This is the out-of-bag classifier. The out-of-bag error estimate is then the error rate of the out-of-bag classifier on the training set. Note that in each resampled training set, about one third of the instances are left out (actually $1/e$ in the limit). As a result, out-of-bag estimates are based on combining only about one third of the total number of classifiers in the ensemble. This means that they might overestimate the error rate, certainly when a small number of trees is used in the ensemble.

Random forests have some other interesting properties (Breiman, 2001). They are efficient since only a sample of $f(|F|)$ features needs to be tested in each node, instead of all features. They do not overfit as more trees are added. Furthermore, they are relatively robust to outliers and noise, and they are easily parallelised.

The efficiency gain makes random forests especially interesting for relational data mining, which typically has to deal with a large number of features, many of which are expensive to compute. On the other hand, relational data mining offers an interesting test suite for random forests, exactly because the advantage of random forests is expected to become more clear for very large feature spaces. In relational data mining, data sets with very large feature spaces abound. Moreover, using random forests allows us to enlarge the feature set by including aggregate functions, possibly refined with selection conditions, as discussed in the previous section.

4.2. First order random forests with complex aggregates

In order to upgrade TILDE with complex aggregates to a first order random forest (FORF), we proceeded as follows. First, we built a wrapper around the algorithm in order to get bagging. We made some adaptations to get out-of-bag error estimates.

Next, we built in a filter that allows only a random subset of the tests to be considered at each node.[4] As a result, constructing a new node proceeds as follows: first all possible

[4] This actually differs from the definition in Breiman (2001) where a random subset of the attributes, instead of the tests, is chosen. Note that one attribute may yield different tests.

Table 5 Algorithm for first order random forest induction. The parts of the algorithm that are in boxes show the differences with the algorithm GROW_TREE from Table 3

```
procedure GROW_FOREST (N: nb of trees, f: function, E: examples):
    for i = 1 to N
        E_i := SAMPLE(E)
        T_i := GROW_TREE_2(E_i, true, f)
    return forest(T_1, T_2, ..., T_N)

procedure GROW_TREE_2 (E: examples, Q: query, [f: function]):
    candidates := ρ(← Q)
    [subsetsize := f(|candidates|)]
    [candidates_subset := SUBSET(candidates, subsetsize)]
    ← Q_b := OPTIMAL_SPLIT([candidates_subset], E)
    if STOP_CRIT (←Q_b, E)
    then
        K := PREDICT(E)
        return leaf(K)
    else
        conj := Q_b − Q
        E_1 := {e ∈ E|←Q_b succeeds in e ∧ B}
        E_2 := {e ∈ E|←Q_b fails in e ∧ B}
        left := GROW_TREE_2 (E_1, Q_b, [f])
        right := GROW_TREE_2 (E_2, Q, [f])
        return node(conj, left, right)
```

refinement candidates $\rho(\leftarrow Q)$ from the current query Q are generated, then a random subset of approximate size $f(|\rho(\leftarrow Q)|)$ (where $f(x)$ is a function given by the user, e.g., $f(x) = 0.1x$ or $f(x) = \sqrt{x}, \ldots$) is chosen. For each query in this subset, a heuristic is computed and the optimal split is placed in the new node. Consequently, only a part of all generated queries needs to be executed on the examples to calculate the heuristics, which obviously results in an efficiency gain.

To summarise, we provide an overview of the resulting algorithm in Table 5. The procedure GROW_FOREST takes the number of trees to grow as one of its input parameters. For each tree, it first builds a new set of examples, sampled with replacement from the original set E. Then the procedure GROW_TREE_2 is called, which is an adaptation of GROW_TREE (see Table 3), differences with this algorithm are denoted in boxes. The refinement operator ρ includes (complex) aggregate conditions in the set of candidate splits it generates, as discussed in Section 3.2. The SUBSET procedure generates a random subset of the candidate set. The size of the subset is a function f of the number of candidates. Hence, each candidate has probability $\frac{f(|\rho(\leftarrow Q)|)}{|\rho(\leftarrow Q)|}$ to be selected. The OPTIMAL_SPLIT procedure returns the optimal split among a set of candidate splits.

4.3. Forf: A more efficient approach

An important difference between propositional random forests and first order random forests is the generation of tests at each node. In a propositional tree the possible tests are the same at each node. In first order trees however, a node may introduce variables that can be reused

in the nodes of its left subtree. Hence, the number of candidate tests depends on the number of variable bindings in the conjunction of all succeeding tests on the path from the root to the node that is to be extended (this conjunction was called the current query in Section 3.1).

Query sampling reduces the time used for query evaluation in random forests. Still, in FORF, query generation also takes a substantial amount of time, certainly when lookahead within the aggregate queries is performed and a huge amount of queries needs to be generated. In that case, a lot of time is spent on generating queries that may not be evaluated in the end. As such, the algorithm described in Table 5 is still performing a lot of redundant actions. A more efficient version of the GROW_TREE_2 procedure would directly generate a random sample of queries, instead of generating them all. This is not trivial since the number of queries in the sample is a function of the total number of possible queries, which is hard to calculate in advance. Moreover, if we iteratively take a random literal[5] from the language bias to produce a random candidate query, the resulting sample will not be drawn from a uniform distribution over all possible candidates, as the number of candidates a literal from the language bias produces depends on the current query. Therefore, such an efficient sample generator would consist of two steps. First, for each literal in the language bias, the number of candidates that can be generated from it would have to be determined (without generating them all). Second, using the uniform distribution over candidates, obtained from the first step, the query sample could be randomly generated.

In TILDE, the refinement operator ρ is implemented as follows: for each literal in the language bias, variable instantiation is performed and for each of these variable instantiations all possible constants are generated (in case of lookahead, these are again refined in the same way). Let us illustrate this with an example where we use the same example database from Fig. 1. A possible language bias for this database is given in Table 4.

Suppose we want to refine a node where the current query Q is

$$person(P, C), account(A_1, P, T_1, B_1), account(A_2, P, checkings, B_2).$$

Then TILDE will generate all candidate refinements for this query according to the tree in Fig. 2. We only used the standard literals for simplicity (the aggregate conditions are simply too large to fit in Fig. 2 and are not treated differently by the sampling procedure). At depth 1 of the tree all literals that occur in the language bias (so both the ones specified by the *rmode* constructs as those specified by the *aggcondition* constructs) are added, at depth 2 variable instantiations with respect to query Q are performed and at depth 3 possible constants are filled in. If lookahead is used, some leaves of this tree are again expanded as if they were the root of the tree.

In order to count, in an efficient way, the number of candidate tests an rmode (or aggregate condition) produces, we proceeded as follows. The tree from Fig. 2 is only built partially, in the sense that each instantiated literal of depth two only has a single child node, representing the constants for that literal. Hence, in case of lookahead (or in case of complex aggregates), the corresponding candidates are generated only once, instead of once per available constant. We can assign probabilities to each literal in the language bias by counting how much offspring it yielded, thereby multiplying each node representing the constants with the number of constants available for that literal (this number can be obtained from the language bias). For example, for the last rmode in Fig. 2 we find in the language bias from Table 4 that there are three constants. While generating the search tree, we find two instantiations for this rmode.

[5] Such literals correspond to the literals that are specified by the *rmode* and *aggcondition* constructs in the language bias. See Table 4.

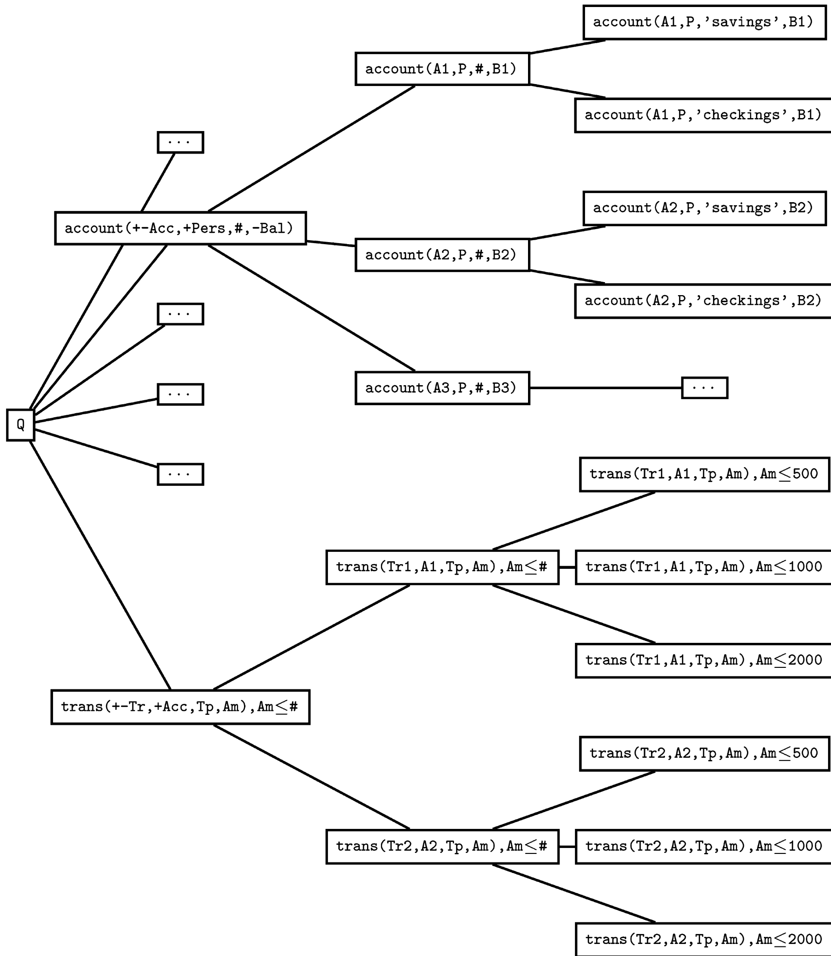

Fig. 2 Generation of candidates to refine the query Q: $person(P, C)$, $account(A_1, P, T_1, B_1)$, $account(A_2, P, checkings, B_2)$ in TILDE. At depth 1 all literals occurring in the language bias from Table 4 are added, at depth 2 variables are instantiated and at depth 3 constants are added

Thus, a total of 6 refinements are obtained from this rmode. Doing this for all rmodes we get a distribution over the different rmodes. Using this distribution a sample of queries is randomly generated.

This approach will be especially rewarding if the tree of Fig. 2 contains many levels, e.g., because of the use of lookahead or complex aggregates, and if its branching factor can be largely reduced (i.e. when a lot of constants need to be filled in). The largest gain over the naive algorithm will occur when using small sample ratios. Obviously, when one would use a ratio of, say 90%, the extra work for counting the number of candidates will not pay off.

5. Experiments

In this section we experimentally evaluate our method. We first describe what we want to learn from the experiments and how we will assess this. Then experimental results are reported

on different real world applications and one artificially generated data set. Afterwards, we discuss conclusions that can be drawn from the experiments.

5.1. Experimental setup

We want to investigate the strength of first order random forests (FORF) in a setting where the feature set is expanded with aggregates, both simple and complex ones. The precise questions we want to answer are the following:

1. Does the use of aggregates (both simple and complex ones) improve the performance?
2. How do first order random forests perform compared to first order decision trees?
3. What is the influence of the number of trees in a random forest?
4. What is the influence of the size of the sample of features that is taken in each node of the random forest? And related, is the optimal sample ratio (i.e., the minimal sample ratio that does not hurt the performance of FORF) influenced by the size of the feature space?
5. How does the performance of FORF relate to other available relational learners?

For each of these five questions, we describe the methodology used in the experiments:

1. We have investigated the performance of first order random forests according to different levels of aggregation. In the first level, we did not use any aggregates (afterwards, this setting is called FORF-NA). In the second level, simple aggregate conditions were introduced (FORF-SA). The third level includes refinement of aggregate queries (FORF-RA) and the fourth level allows lookahead up to depth 1 within the aggregate queries (FORF-LA). We do not allow to refine aggregate queries with new aggregate conditions, since otherwise the search space becomes too large for some of our experimental settings. For FORF-LA we were not able to show results with the TILDE algorithm since it required too much memory (in the tables this will be denoted with ‘out of memory’ or ‘O.O.M.’). We report accuracy as well as complexity of the trees in FORF and the time needed to refine the nodes.
2. For all experiments we ran both FORF and TILDE and compared their predictive accuracies. The trees output by TILDE were pruned using C4.5’s post-pruning method. In FORF no pruning was used, since pruning decreases the diversity among the trees in our random forest. For the two-class data sets we also compared the ROC behaviour (Provost & Fawcett, 2001) of FORF (with 33 trees and a sample ratio of 25%) to that of its base classifiers and that of TILDE. Area under the ROC-curve (AUC) is reported.
   We obtain a ROC-curve for a single tree in the standard way: a prediction is positive if the proportion of positives in the leaf of the instance being predicted is above some threshold; by varying this threshold a ROC-curve is obtained. For a forest, it is the mean proportion found in the different trees that is compared to the (varying) threshold. For both FORF and TILDE we obtain the ROC-curves by doing fivefold cross-validation with the same folds.
3. We examined the influence of the number of trees in the random forests, experimenting with 3, 11, and 33 trees.
4. We considered random subsets of 100%, 75%, 50%, 25%, 10%, and the square root of the number of tests at each node in the trees to test the influence of the size of the feature sample. We report accuracy, but also the complexity of the trees and the time needed to refine one node in a tree.
5. We compare the performance of FORF on the real world data sets to that of other systems available in the literature.

We now discuss the error assessment for the different questions. In all experiments where the different parameter settings of FORF are compared (questions 1, 3 and 4) and where FORF is compared to TILDE (question 2), the accuracy of FORF was computed using out-of-bag estimation and this was carried out five times and averaged, in order to obtain a more reliable estimate of the performance. The predictive performance of TILDE in this comparison is obtained by averaging five full threefold cross-validations with different folds. We use threefold cross-validation for TILDE since this error assessment approach is closest to out-of-bag estimation, as out-of-bag estimation also uses about 66% of the data as training data. The accuracy results concerning the questions 1, 2 and 4 are reported together in one table for each of the application domains.

To compare FORF to other available systems (question 5) on the other hand, we reran FORF with one particular parameter instantiation using tenfold cross-validation and did this five times with different folds to get more reliable estimates. We follow this approach since the results of the other systems were also obtained by doing five times tenfold cross-validation.

In the next sections, we present results for these experiments on three well-known real world data sets: Mutagenesis (Srinivasan, King & Bristol, 1999), Diterpenes (Džeroski et al., 1998), and Financial (Berka, 2000). The first two data sets contain complicated structures and have been widely used as ILP benchmarks. The latter is a business domain data set with high degree of non-determinacy. We also performed some experiments on artificially generated Trains data (Michalski, 1980) with a predefined concept containing complex aggregate functions (Section 5.3). We discuss the results, related to the questions formulated above, in Section 5.4.

5.2. Real world data

5.2.1. Mutagenesis

For our first experiment we used the Mutagenesis data set. This ILP benchmark data set, introduced to the ILP community by Srinivasan, King and Bristol (1999), consists of structural descriptions of 230 molecules, of which 188 are called "regression-friendly". The molecules are to be classified as mutagenic (60%) or not. The description consists of the atoms and the bonds that make up the molecule, i.e., the so called background B1. The aggregate functions used were *count*, *count_dist*, *mode*, *mode_dist*, and *min*. Predictive accuracies on the full data set related to the questions 1, 2 and 4 from Section 5.1 are shown in Table 6. As can be seen from this table the use of aggregates is clearly beneficial: the accuracy increases by adding (more and more complex) aggregates. Performance tends to decrease slightly when considering fewer features. But still, a sampling of, for instance, 25% is not significantly worse than sampling of 100% and comes with a substantial efficiency gain. FORF also outperforms TILDE.

An example of a test that was frequently found at high levels in the different trees was the following aggregate condition (with the range of Mol bound to a molecule)

$$count(Bnd, (bond(Bnd, Mol, At_1, At_2, Tp)), C), C > 28.$$

with the following refinement in its left child

$$count(Bnd, (bond(Bnd, Mol, At_1, At_2, Tp), atom(Mol, At_1, carbon, Ch)), C'), C' > 28.$$

The first part of the example represents the set of all molecules that have at least 28 bonds. This aggregate was also found to be a good test by Knobbe, Siebes and Marseille (2002). The refinement of the aggregate describes all molecules that have at least 28 bonds connected

Table 6 Accuracy results on the full Mutagenesis data set. The rows indicate the sample ratio at each node. The columns compare predictive accuracies for the different aggregation levels (LA, RA, SA, and NA) for FORF with 33 trees and TILDE. The standard deviation is indicated between parentheses. Results for FORF are obtained by averaging over 5 runs of out-of-bag estimation, while results for TILDE are computed by averaging over 5 full threefold cross-validations. Averages over the different aggregate settings (LA left out, because of 'O.O.M' for sampling 1 and 0.75) and different sample ratios (1 and 0.75 left out because of 'O.O.M' for LA setting) are also provided for FORF

Sample ratio	FORF-LA	FORF-RA	FORF-SA	FORF-NA	avg
1	O.O.M.	0.770 (0.016)	0.773 (0.012)	0.729 (0.013)	0.757
0.75	O.O.M.	0.771 (0.025)	0.757 (0.014)	0.723 (0.012)	0.750
0.50	0.795 (0.010)	0.760 (0.013)	0.762 (0.014)	0.710 (0.022)	0.744
0.25	0.793 (0.012)	0.756 (0.012)	0.768 (0.015)	0.717 (0.014)	0.747
0.10	0.790 (0.013)	0.751 (0.017)	0.743 (0.017)	0.677 (0.026)	0.724
sqrt	0.797 (0.013)	0.739 (0.010)	0.757 (0.016)	0.702 (0.012)	0.733
avg	0.794	0.752	0.758	0.702	
	TILDE-LA	TILDE-RA	TILDE-SA	TILDE-NA	
1	O.O.M.	0.731 (0.029)	0.733 (0.005)	0.690 (0.021)	

Table 7 Complexity and timing results on the full Mutagenesis data set. The top table shows the average complexity (number of nodes) of one tree in a forest. The bottom table shows the average time needed to refine a single node in a tree of the forest

	Complexity			
	FORF-LA	FORF-RA	FORF-SA	FORF-NA
1.0	16.6	21.4	21.8	18.4
0.75	19.8	19.4	16.2	16.4
0.50	18.8	20.4	19.6	14.2
0.25	23.2	18.2	13.6	15.8
0.10	20.2	15.0	9.6	11.4
sqrt	19.2	9.4	6.8	6.8

	Time per node			
	FORF-LA	FORF-RA	FORF-SA	FORF-NA
1.0	15549.952	211.907	70.294	17.272
0.75	25684.434	295.340	56.222	15.195
0.50	12973.660	162.569	35.102	12.746
0.25	5411.500	64.088	22.382	11.038
0.10	2096.267	27.200	11.042	7.526
sqrt	117.969	28.702	12.882	2.118

to an atom of type carbon. Unfortunately, we cannot compare our accuracy results to those given by Knobbe, Siebes and Marseille (2002), since they only report predictive accuracies for one best rule, covering only a part of the positive examples.

Table 7 shows the average complexity (number of nodes) of one tree in the random forests. To get the average number of nodes in the forest, these results need to be multiplied by the number of trees in the forest, which is 33 in this case. Complexity of the trees seems to decrease slightly when taking smaller samples. The bottom of Table 7 shows the average time that is used to refine one node in a tree. Sampling up to the square root of the number of

Table 8 Accuracy results on the "regression-friendly" Mutagenesis data set compared to other systems. The results for FOIL, PROGOL are obtained from Srinivasan, Muggleton and King (1995), TILDE from Blockeel & De Raedt (1998) and ROLLUP from Knobbe, de Haas and Siebes (2001). All results were obtained by averaging over five tenfold cross-validations

FORF-LA	FOIL	PROGOL	TILDE	ROLLUP
0.860 (0.014)	0.610 (0.060)	0.760 (0.030)	0.750	0.860

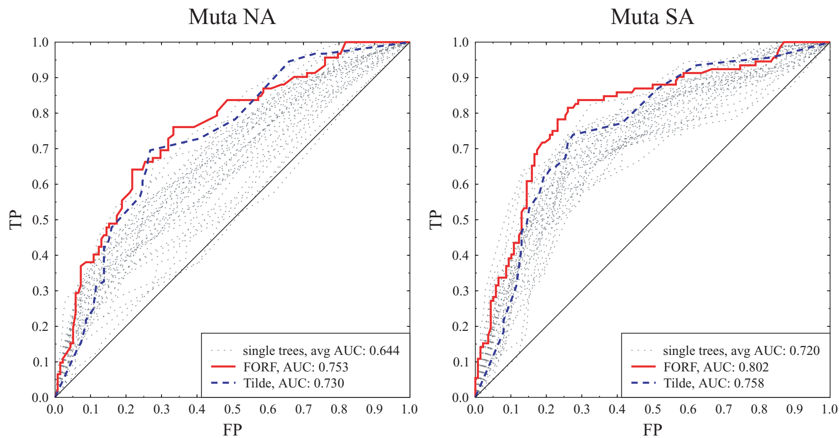

Fig. 3 ROC-curves for Mutagenesis obtained by fivefold cross-validation. The left figure shows results without using aggregates, the right figure with using simple aggregates. FORF is built using 33 trees and a sample ratio of 25%

features seems to give an efficiency gain of one order of magnitude for FORF-NA up to two orders of magnitude for FORF-LA.

Figure 3 provides the ROC-curves for FORF, all its base classifiers and TILDE. As shown in the figure, the AUC of FORF is considerably higher than the average AUC of its base classifiers or that of TILDE.

Table 8 shows predictive accuracies of FORF compared to other systems (Srinivasan, Muggleton & King, 1995; Blockeel & De Raedt, 1998). These results though, were obtained using tenfold cross-validation only on the "regression-friendly" part of the Mutagenesis data set. FOIL (Quinlan, 1990) induces concept definitions represented as function-free Horn clauses, from relational data. PROGOL (Muggleton, 1995) is an ILP learner capable of learning in structurally very complex domains. ROLLUP (Knobbe, de Haas & Siebes, 2001) is a propositionalisation approach that makes use of aggregates. We used the LA-setting since it gives clear improvements over the other aggregate levels. As a sample function we used $sqrt(x)$. From Table 8, we see that FORF performs at least as good as the other systems.

5.2.2. *Diterpenes*

In our second experiment we used the Diterpenes data set (Džeroski et al., 1998). The data contains information on 1503 diterpenes with known structure. The $red(Mol, Mult, Freq)$ relation stores the measured NMR-Spectra. For each of the 20 carbon atoms in the diterpene skeleton, it contains the multiplicity and frequency. The $prop(Mol,$

Table 9 Accuracy results on the Diterpenes data set. The rows indicate the sample ratio at each node. The columns compare predictive accuracies for the different aggregation levels (LA, RA, SA, and NA) for FORF with 33 trees and TILDE. The standard deviation is indicated between parentheses. Results for FORF are obtained by averaging over 5 runs of out-of-bag estimation, while results for TILDE are computed by averaging over 5 full threefold cross-validations. Averages over the different aggregate settings (LA left out, because of 'O.O.M' for sampling 1) and different sample ratios (1 left out because of 'O.O.M' for LA setting) are also provided for FORF

	FORF-LA	FORF-RA	FORF-SA	FORF-NA	avg
1	O.O.M.	0.829 (0.011)	0.849 (0.003)	0.768 (0.003)	0.815
0.75	0.859 (0.003)	0.840 (0.002)	0.849 (0.004)	0.769 (0.002)	0.819
0.50	0.856 (0.006)	0.839 (0.004)	0.850 (0.001)	0.766 (0.004)	0.818
0.25	0.856 (0.004)	0.849 (0.007)	0.847 (0.004)	0.763 (0.004)	0.820
0.10	0.853 (0.004)	0.823 (0.012)	0.842 (0.002)	0.739 (0.007)	0.801
sqrt	0.844 (0.005)	0.806 (0.004)	0.824 (0.006)	0.716 (0.004)	0.782
avg	0.854	0.832	0.843	0.751	
	TILDE-LA	TILDE-RA	TILDE-SA	TILDE-NA	
1	O.O.M.	0.827 (0.006)	0.834 (0.004)	0.723 (0.008)	

Satoms, *Datoms*, *Tatoms*, *Qatoms*) relation counts the atoms that have multiplicity s, d, t, or q respectively. Additional unary predicates describe to which of the 23 classes a compound belongs. Several learning settings are defined on this data set: using *prop* only, using *red* only, and using both *prop* and *red*. In our experiments with FORF, only the *red*-relation was used, since we expect that by allowing aggregate functions FORF should be able to construct the *prop*-relation by itself if necessary. The aggregate functions used are *min*, *max* and *avg* for the frequencies, *mode* for the multiplicities, and *count* for the different values of multiplicities.

Table 9 gives predictive accuracies of experiments using 33 trees. These results are used to evaluate questions 1, 2 and 4 from Section 5.1. For efficiency reasons, we changed the minimal number of examples a leaf has to cover from 2 to 20, both for FORF and for TILDE. As a result, the numbers in the table do not compare favourably with earlier published results (Džeroski et al., 1998), but the experimental setting is also different; especially the minimal leaf size of 20 seems to have a detrimental effect. From this table we can see that using simple aggregates leads to a large performance improvement. Refining aggregates or using lookahead does not improve accuracy further though. Taking both accuracy and efficiency into account, again sampling at 25% or even 10% of the number of features seems to be advisable.

Table 10 shows the average complexity (number of nodes) of one tree in the random forests. As can be seen from this table, the jump in performance from FORF-NA to FORF-SA (visible in Table 9) can also be found in the complexity results: there is a significant drop in complexity when using simple aggregates. This suggests that aggregates are necessary to learn the concept in this data set. At the bottom it shows the average time that is used to refine one node in a tree. Again sampling is more beneficial when the aggregation level is higher. The table also shows that, at least for the NA-setting and a little less for the SA-setting, the time to refine one node increases when going from no sampling to sampling 75% of the features. As was explained in Section 4.3, the extra work for counting the number of candidates will not pay off if both the sample ratio is too large and the number of features is too small.

To allow a good comparison with previously published results (question 5 from Section 5.1), we performed a single experiment where, just like in Džeroski et al. (1998),

Table 10 Complexity and timing results on the Diterpenes data set. The top table shows the average complexity (number of nodes) of one tree in a forest. The bottom table shows the average time needed to refine a single node in a tree of the forest. 'X' indicates that these timing results are not comparable to the others due to memory swapping

	Complexity			
	FORF-LA	FORF-RA	FORF-SA	FORF-NA
1.0	X	10.2	10.8	16.4
0.75	X	11.2	12.4	18.0
0.50	X	11.8	13.0	17.2
0.25	10.8	13.8	13.6	18.6
0.10	9.2	12.8	14.4	15.0
sqrt	13.0	12.4	16.8	15.4
	Time per node			
1.0	X	424.353	190.593	56.024
0.75	X	347.357	197.016	109.111
0.50	X	214.881	95.185	74.651
0.25	44302.000	116.899	49.015	33.538
0.10	18070.609	52.750	26.236	19.120
sqrt	479.785	29.355	12.726	9.234

Table 11 Accuracy results on the Diterpenes data set compared to other systems. The results for FOIL, RIBL, ICL, and TILDE are obtained from Džeroski et al. (1998) All results were obtained by averaging over five tenfold cross-validations (no standard deviations were available for these results)

	FORF-SA	FOIL	RIBL	ICL	TILDE
red	0.928 (0.006)	0.465	0.865	0.816	0.653
red+prop		0.783	0.912	0.860	0.904

results of five tenfold cross-validations are averaged. The minimal leaf size of trees was reset to 2; and we used the FORF-SA setting with 33 trees and a sampling ratio of 25%, which, judging from Table 9, are good parameter values. The result of this experiment is compared with published results for other first order systems in Table 11. RIBL (Emde & Wettschereck, 1995) is a relational instance based learning algorithm. The ICL system (De Raedt & Van Laer, 1995) uses exactly the same representation as TILDE, but induces rule sets instead of trees.

We only used the *red*-relation for FORF, since it should be able to construct the *prop* relation by itself, but we compared it to the other systems both using only *red* and using *red* and *prop*. It was already found that combining propositional (aggregate) features with relational information yielded the best results (Džeroski et al., 1998). Comparing with those best results, we see that FORF is at least competitive with the best of the other approaches and can construct the aggregates that other systems need to be given.

5.2.3. *Financial*

Our last real world experiment deals with the Financial data set, originating from the discovery challenge that was organised at PKDD'99 and PKDD'00 (Berka, 2000). This data set involves learning to classify expired bank loans into good and bad ones. Since 86% of the examples is positive, the data distribution is quite skewed. The data set consists of 8 relations. For each of the 234 loans, customer information and account information is provided. The account

Table 12 Accuracy results on the Financial data set. The rows indicate the sample ratio at each node. The columns compare predictive accuracies for the different aggregation levels (LA, RA, SA, and NA) for FORF with 33 trees and TILDE. The standard deviation is indicated between parentheses. Results for FORF are obtained by averaging over 5 runs of out-of-bag estimation, while results for TILDE are computed by averaging over 5 full threefold cross-validations. Averages over the different aggregate settings (LA left out, because of 'O.O.M' for sampling 1 and 0.75) and different sample ratios (1 and 0.75 left out because of 'O.O.M' for LA setting) are also provided for FORF.

	FORF-LA	FORF-RA	FORF-SA	FORF-NA	avg
1	O.O.M.	0.992 (0.002)	0.995 (0.004)	0.850 (0.004)	0.946
0.75	O.O.M.	0.994 (0.005)	0.995 (0.004)	0.847 (0.007)	0.945
0.50	0.997 (0.005)	0.996 (0.006)	0.998 (0.002)	0.843 (0.008)	0.946
0.25	0.998 (0.004)	0.997 (0.004)	0.993 (0.002)	0.852 (0.002)	0.947
0.10	0.997 (0.004)	0.995 (0.006)	0.995 (0.004)	0.855 (0.007)	0.948
sqrt	0.983 (0.009)	0.989 (0.007)	0.990 (0.008)	0.857 (0.006)	0.945
avg	0.994	0.994	0.994	0.852	
	TILDE-LA	TILDE-RA	TILDE-SA	TILDE-NA	
1	O.O.M.	0.962 (0.010)	0.985 (0.009)	0.847 (0.009)	

information includes permanent orders and several hundreds of transactions per account. This problem is thus a typical business data set which is highly non-determinate. The aggregate functions used apply to the orders and transactions and include all the functions mentioned in Table 2.

Predictive accuracies are shown in Table 12. Again we see that it is very beneficial to add simple aggregates to our language. We find an average improvement of 14%. Refinement of aggregates or lookahead within the aggregates does not give any further gain.

Table 13 shows the average complexity (number of nodes) of one tree in the random forests. Again we see a clear drop in complexity corresponding to the increase in performance when adding simple aggregates to the language. At the bottom the table shows the average time that is used to refine one node in a tree. For the NA-setting sampling does not seem to improve efficiency. The reason is that the number of non-aggregate features for this data set is quite small, which causes the overhead of determining the number of features in the sampling procedure (see Section 4.3) to be higher than the resulting efficiency gain for the NA-setting. Once aggregates are introduced this is no longer the case. For the LA-setting, for instance, we again find a gain of two orders of magnitude.

Figure 4 shows the ROC analysis on this data set. As can be seen from the right figure, for the SA setting the ROC-curve for FORF coincides with the Y-axis since its AUC is 1. This means that this forest is able to rank all examples correctly, but since its accuracy is lower than 1, the threshold for classification is too low. If we would increase the usual threshold of 0.5 for classification to the optimal one, which corresponds to moving from the point with coordinates (1,0.16) in ROC-space to the point (1,0), the forest would correctly classify all examples.

Table 14 shows predictive accuracies compared to other systems (Krogel & Wrobel, 2001). DINUS-C (Lavrač & Džeroski, 1994) is a propositionalisation technique using only determinate features and using C4.5 rules as propositional learner. RELAGGS (Krogel & Wrobel, 2001) was discussed in Section 2. For the random forest, we used the SA setting, since

Table 13 Complexity and timing results on the Financial data set. The top table shows the average complexity (number of nodes) of one tree in a forest. The bottom table shows the average time needed to refine a single node in a tree of the forest

	Complexity			
	FORF-LA	FORF-RA	FORF-SA	FORF-NA
1.0	6.2	6.6	6.2	18.6
0.75	7.4	3.8	6.8	21.0
0.50	6.6	7.6	6.0	20.6
0.25	3.8	5.2	3.8	22.6
0.10	5.2	5.6	5.6	12.2
sqrt	7.6	7.2	6.8	14.8
	Time per node			
1.0	65430.323	870.758	867.097	1.656
0.75	14709.946	1171.789	746.441	2.381
0.50	11280.939	392.368	530.267	2.087
0.25	10542.789	292.231	392.632	1.584
0.10	2096.731	156.107	125.429	1.738
sqrt	152.316	42.000	48.000	1.743

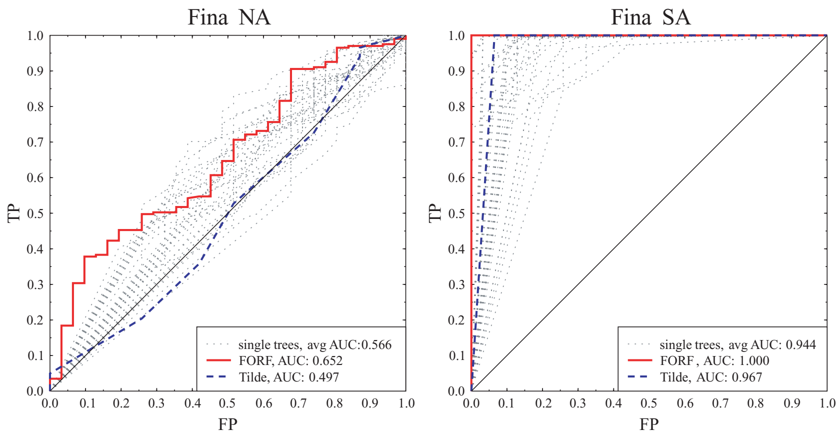

Fig. 4 ROC-curves for the Financial data set. The left figure shows results without using aggregates, the right figure with using simple aggregates. FORF is built using 33 trees and a sample ratio of 25%

neither LA nor RA yielded significantly better results for this data set. The forest contained 33 trees and a sampling ratio of 25%.

5.3. Artificial data

As can be seen from Tables 6, 9 and 12, the use of simple aggregates yields quite a large performance improvement on all data sets. Refinement of the aggregate conditions, on the other hand, increased accuracy only in a few cases, and in general the use of lookahead within the aggregate query also added only a slight performance improvement. The reason for these small improvements could be that the target concept simply does not require a combination of selection and aggregation. To test this conjecture and to know whether complex aggregates can add any improvements when the target concept does contain

Table 14 Accuracy results on the Financial data set compared to other systems. The results for DINUS-C, RELAGGS, and PROGOL are obtained from (Krogel & Wrobel, 2001). FORF-SA is built using 33 trees and a sample ratio of 25%. All results were obtained by averaging over five tenfold cross-validations. The standard deviation is indicated between parentheses

FORF-SA	DINUS-C	RELAGGS	PROGOL
0.993 (0.005)	0.851 (0.103)	0.880 (0.065)	0.863 (0.071)

complex aggregates, we conducted experiments on artificially generated data where the target concept was defined to involve these complex aggregate functions. We used the Random Train Generator from Muggleton[6] for generating random "Michalski-style" train examples (Michalski, 1980) according to a specified concept. We used the following concept:

$$\begin{aligned}
&eastbound(T) \leftarrow sum(W, (car(T, C), wheels(C, W)), SumW), SumW > 7,\\
&\qquad count(C, (car(T, C), roof(C, none),\\
&\qquad\qquad load(C, rectangle, N)), NbC),\\
&\qquad NbC > 1, !.\\
&westbound(T) \leftarrow sum(W, (car(T, C), wheels(C, W)), SumW), SumW > 7, !.\\
&westbound(T) \leftarrow sum(L, (car(T, C), load(C, circle, L)), SumL), SumL > 1, !.\\
&eastbound(T).
\end{aligned}$$

This concept states that if a train has more than 7 wheels and there is more than one car without a roof and with a rectangular load, the train goes east, else if only the first condition holds or if the train has at least 2 circular loads, it goes west. In all other cases it goes east.

Table 15 gives an overview of predictive accuracies for different settings of the sample size and different aggregate levels. The aggregate functions used include counting the number of cars, and taking the sum over the number of wheels and the number of loads. In this table we report an average over the results of training on 5 randomly generated training sets of 500 examples and testing on test sets of 500 examples, half of the examples being positive and half negative. Note that for this data set each level of aggregation brings about a considerable accuracy increase. This suggests that when the concept indeed involves complex aggregates, it is very useful to use higher levels of aggregation.

Table 16 shows the average complexity (number of nodes) of one tree in the random forests. Where for the Diterpenes and Financial data sets the complexity dropped only when adding simple aggregates, we now see a further clear drop when refining the aggregates, especially when using lookahead within the aggregates. This decrease of complexity when adding higher levels of aggregation again corresponds to an increase in performance as could be seen from Table 15. The lower half of Table 16 shows the average time that is used to refine one node in a tree. As for the Financial data set, sampling does not seem to improve efficiency for the NA-setting. For the other levels of aggregation the efficiency gain is also lower than for the real world data sets because the number of features is considerably smaller. For instance, the maximum size of the number of generated features (LA-setting) is 123005 for the Financial data set while for the Trains data set only 2594.

[6] The train generator is available at http://www-users-csyork. ac.uk/∼stephen/progol.html.

Table 15 Accuracy results on the Trains data set. The rows indicate the sample ratio at each node. The columns compare predictive accuracies for the different aggregation levels (LA, RA, SA, and NA) for FORF with 33 trees and TILDE. The standard deviation is indicated between parentheses. Averages over the different aggregate settings (LA left out, because of 'O.O.M' for sampling 1) and different sample ratios (1 left out because of 'O.O.M' for LA setting) are also provided for FORF

	FORF-LA	FORF-RA	FORF-SA	FORF-NA	avg
1.0	O.O.M.	0.911 (0.010)	0.915 (0.008)	0.749 (0.010)	0.858
0.75	0.964 (0.007)	0.946 (0.003)	0.915 (0.006)	0.746 (0.009)	0.869
0.50	0.959 (0.011)	0.940 (0.006)	0.908 (0.005)	0.742 (0.006)	0.863
0.25	0.962 (0.009)	0.952 (0.010)	0.904 (0.011)	0.739 (0.012)	0.865
0.10	0.965 (0.006)	0.925 (0.012)	0.849 (0.003)	0.719 (0.010)	0.831
sqrt	0.954 (0.009)	0.909 (0.014)	0.853 (0.009)	0.720 (0.012)	0.827
avg	0.961	0.934	0.886	0.733	
	TILDE-LA	TILDE-RA	TILDE-SA	TILDE-NA	
1	O.O.M.	0.908 (0.023)	0.889 (0.024)	0.707 (0.032)	

Table 16 Complexity and timing results on the Trains data set. The top table shows the average complexity (number of nodes) of one tree in a forest. The bottom table shows the average time needed to refine a single node in a tree of the forest

	Complexity			
	FORF-LA	FORF-RA	FORF-SA	FORF-NA
1.0	O.O.M.	25.1	26.3	53.1
0.75	9.7	24.2	29.3	53.5
0.50	10.4	26.0	31.3	54.2
0.25	11.2	27.0	32.6	51.9
0.10	13.7	28.6	25.0	37.9
sqrt	21.9	28.2	25.0	41.8
	Time per node			
1.0	O.O.M.	7.420	4.735	2.831
0.75	564.305	9.530	5.611	3.913
0.50	407.321	7.183	4.402	3.202
0.25	206.340	5.251	3.258	2.435
0.10	72.702	4.469	2.847	2.033
sqrt	16.670	4.121	2.881	2.141

Figure 5 shows the ROC analysis for FORF, all its base classifiers and TILDE. Here we do show the ROC-curves for all different aggregate levels since on this data set FORF-LA gives a large improvement over FORF-SA, as can be seen from the figure.

5.4. Discussion of the results

In this section we summarise the conclusions drawn from our experiments, and relate them to the five questions formulated in Section 5.1.

5.4.1. No aggregates vs. aggregates

The results on the real world data sets (see Tables 6, 9 and 12) make clear that the use of aggregates is really beneficial for the performance of both TILDE and FORF. Especially using

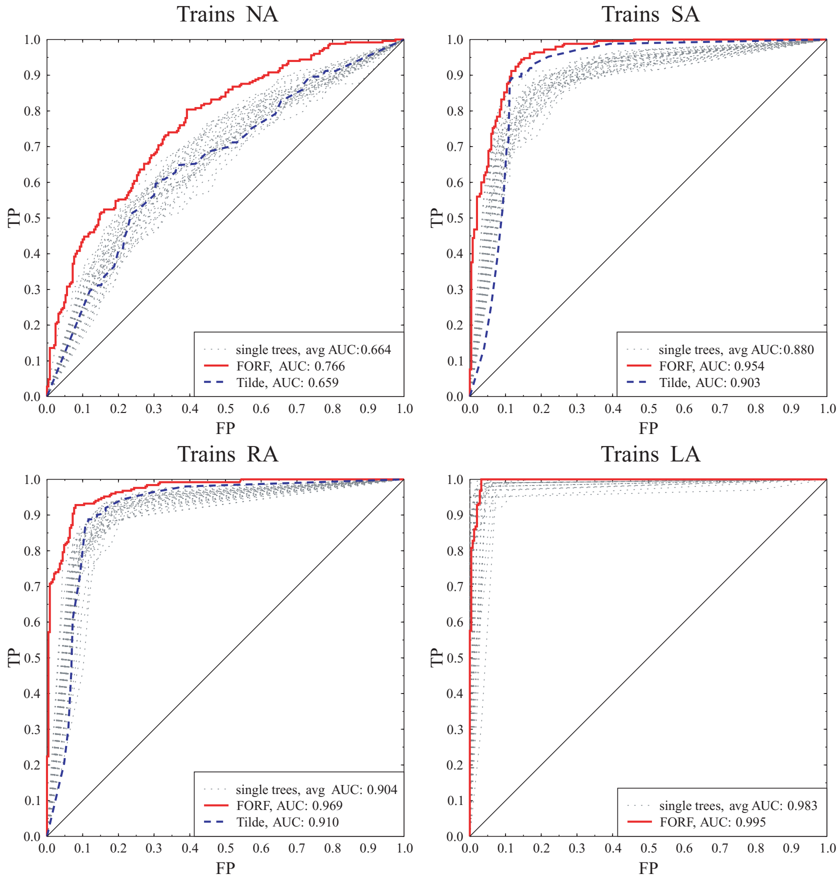

Fig. 5 ROC-curves for the Trains data set. From left to right and top to bottom: the results without aggregates, using simple aggregates, using refined aggregates and aggregates with lookahead. For this last setting no results were available for TILDE due to memory restrictions. FORF is built using 33 trees and a sample ratio of 25%

simple aggregates yields a large performance boost on all data sets. Refining those aggregates or using lookahead to construct aggregate conditions only gives a slight improvement on the real world data sets considered. Hence, the experiments suggest that in these data sets, complex aggregates are often not necessary to describe the target concept. Experiments on artificially generated data sets however, show large improvements when using these settings on target concepts with complex aggregates (see Table 15 and Fig. 5). Especially using lookahead within the aggregate query is very rewarding with respect to predictive performance in that context. We also found that the use of aggregates often decreases the complexity of the trees (see Tables 10, 13 and 16) .

5.4.2. Trees vs. random forests

Throughout all the experiments first order random forests in general outperformed TILDE with respect to predictive accuracy. Although ensemble methods are quite computationally

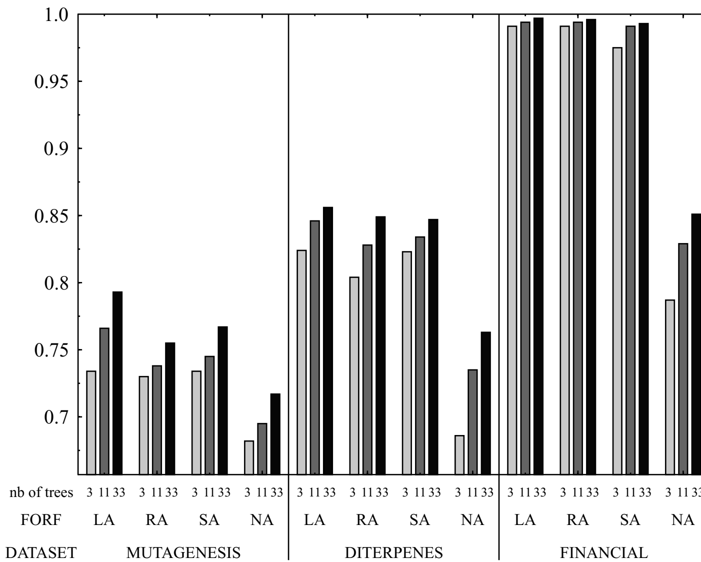

Fig. 6 Accuracy for random forests using 25% of the features. Results are shown for FORF-LA, FORF-RA, FORF-SA and FORF-NA for different tree sizes on the three different real world data sets

expensive, since different models are built and combined, FORF is able to considerably reduce this cost by sampling the feature space, as such also decreasing the memory cost. Especially when the feature space becomes very large, and traditional ILP systems run into problems, this seems to be very beneficial. More concretely, in the case of Mutagenesis for instance, sampling up to the square root of the number of features gives an efficiency gain of one order of magnitude for the NA setting up to two orders of magnitude for the LA setting.

We have also investigated the ROC behaviour of FORF with respect to its base classifiers and to TILDE (see Figs. 3, 4 and 5). We found again that FORF outperformed TILDE regarding their area under the ROC-curve (AUC). Even when the accuracy of TILDE is not significantly different from the accuracy of FORF (e.g., for SA in the Financial data set), we can still conclude that FORF is doing a better job than TILDE in ranking the examples from positive to negative, since its AUC is higher. A drawback of FORF compared to TILDE is of course the model complexity. Since it consists of several trees, a random forest is very hard to interpret.

5.4.3. Number of trees

As is to be expected and can be seen from Fig. 6, adding more trees to the forest clearly increases the performance in all experiments. Of course, more trees means longer runtimes, so there is still a trade-off between efficiency and accuracy.

5.4.4. Sample size

We observe that for all experiments, using only a random part of the features (certainly down to 25%) to select the best test seems to be advisable, since there is no significant difference in accuracy and we profit from the efficiency gain by testing fewer features in each node. Breiman (2001) on the other hand, obtained improvements with even much smaller

proportions of the feature set in the propositional case, in fact choosing a random sample of one single feature seemed to work well. This setting was also tested in FORF, but did not yield good results (these results are not reported here). This difference is probably due to the fact that in our approach a random subset of tests is taken, while Breiman takes a random subset of the attributes, and selects the best test using these attributes.

When looking at the increase in size of the feature space resulting from using aggregation, random forests indeed seem to profit from a large feature space, which is certainly present when using lookahead in the aggregate queries. To illustrate, the maximum size of the feature space that was observed for the Financial data set was 110 for the NA setting, versus 123005 for the LA setting. While for FORF-NA, FORF-SA and FORF-RA accuracy tends to slightly decrease when using sample ratios smaller than 25%, this is less the case for FORF-LA. Hence, for experiments where the feature space becomes very large (e.g., for FORF-LA), we can even sample smaller sets of features (e.g., taking the square root of the number of tests) without sacrificing performance, and by doing so gain efficiency.

As could be seen from Tables 7, 10, 13 and 16 the efficiency gain, obtained by sampling the feature space, also increases when using higher levels of aggregation.

5.4.5. Comparison to other systems

We compared the results of FORF on the real world data sets to available results of a number of other relational systems. FORF clearly outperformed these systems on the data sets considered (see Tables 8, 11 and 14).

6. Related work

In this section we discuss some related work. Since our contributions are situated in two different topics, this is reflected in our discussion. Section 6.1 covers the combination of selection and aggregation. Related work on random forests is presented in Section 6.2.

6.1. Aggregates and selection

Whereas traditional propositionalisation approaches to ILP such as Linus or Dinus (Lavrač & Džeroski, 1994) handle only constrained or determinate clauses, Krogel & Wrobel (2001) present a system called RELAGGS where simple aggregation is used to represent non-determinate relationships in summary features. Simultaneously, Knobbe, de Haas & Siebes (2001) propose the system ROLLUP, another system that uses simple aggregates to propositionalise a multi-relational database. The feature sets of ROLLUP and RELAGGS overlap but do not coincide. Krogel and Wrobel (2003) present a comparative evaluation of approaches to propositionalisation, where they compare RELAGGS to logic-oriented transformation approaches.

Other systems employ simple aggregate functions directly in their model representations. For example, Neville et al. (2003) use aggregate functions to construct splits in relational probability trees. Probabilistic relational models (Koller, 1999) use aggregates to specify non-deterministic relations in a dependency structure and in conditional probability tables.

Several authors argue in favour of including selection conditions into the aggregate functions. Perlich and Provost (2003) provide a detailed examination of aggregation for relational learning. They define various classes of relational learning problems with respect to aggregation. On their domain of interest the results demonstrate that aggregation operators of higher

complexity can significantly improve generalisation performance. Blockeel & Bruynooghe (2003) discuss the bias that is imposed on relational learners that either use aggregates or use selections of specific elements and provide some ideas to remove it. One of these ideas is to use relational neural networks. Also, Krogel et al. (2003), when comparing logic-oriented and database-oriented methods for propositionalisation, conclude that a combination of the features produced by both groups of methods seems a valuable venture.

A few combinations of aggregation and selection have been proposed in the literature. Krogel and Wrobel (2003) (when presenting an extended version of the system RELAGGS) introduce in their propositionalised table aggregate functions that apply not only to single attributes, but also to pairs of attributes, one of which has to be nominal and serves as a group by condition. Hence, they include aggregates over selections, but the selection conditions are of limited complexity and are not refined during the search. Knobbe, Siebes and Marseille (2002) introduce aggregate functions into the selection graph pattern language and allow them to be refined. However, in order to obtain a valid refinements, only monotone aggregate conditions are considered to be refined. A non-symbolic approach towards combining aggregates and selections was proposed by Uwents and Blockeel (2005). They describe so called relational neural networks. Their approach is not constrained to using predefined aggregate functions and does not make a distinction between searching for aggregate functions and searching for complex conditions.

6.2. Random forests

Although ensemble methods have proven to be very useful in propositional learning, in relational learning, and in ILP in particular, not much attention has been paid to it. Some exceptions are the initial work on relational boosting of Quinlan (1996) and Hoche and Wrobel (2001). Quinlan (1996) used the ideas of boosting from the propositional context for the first time in ILP, by boosting FFOIL (a first order rule induction system). There are two drawbacks of this approach. On the one hand, understandability of the learned hypothesis drops significantly, since the result is a large set of rules with weighted votes connected to each rule. On the other hand, boosting standard ILP systems is very time consuming due to the high effort already expended by a typical ILP system. Hoche and Wrobel (2001) address these problems by using a method called constrained confidence-rated boosting (based on the work by Schapire and Singer (1999)) on a fast but weak ILP learner. This method improves the understandability of the boosted learning results by restricting the kinds of rule sets allowed.

Apart from this work about boosting, de Castro Dutra et al. (2002) performed an empirical evaluation of bagging in the context of ILP. This work confirms the advantages of using ensemble methods in ILP, achieving an interesting improvement in performance. On the other hand, resulting theories are more complex and thus harder to understand.

7. Conclusions and future work

Our paper makes two main contributions to the field of relational learning, more specifically to the induction of first-order logical decision trees. First, it introduces the capability to use aggregate-based tests in the internal nodes of the trees. Second, it uses the random forest approach to generate ensembles of first-order logical decision trees. Furthermore, the synergy between these two dimensions is exploited. The utility of the proposed developments is evaluated on three real-world and a synthetic dataset.

The capability of using aggregates brings clear performance improvements. The largest improvements are due to the use of simple aggregates and can be observed for both the real-world and the artificial datasets. The use of complex aggregates (aggregates with selection conditions) is only useful if the target concept is complex and clearly involves complex aggregates, as was the case for the artificial datasets.

First order random forests perform better than single trees, as could be expected. Moreover, as random forests only consider a randomly chosen fraction of the possible tests that can go into an internal node, they can even address datasets that are just out of reach of single tree induction. These are datasets where possible tests abound, as is definitely the case when we allow the use of aggregates (especially complex ones). In this context, an important contribution of our paper is the approach where only a random fraction of the tests is generated, rather than being selected from the completely generated set of possible tests (in both cases, the best of the tests is selected afterwards).

aggregates was shown to yield advantages when the target concept involves complex aggregates, it is not known to what extent this kind of concepts occur in real life datasets. Therefore, additional ILP datasets should be investigated.

In this article, the use of complex aggregates was shown to be beneficial in the relational decision tree setting. We believe that in general any relational approach would benefit from the use of aggregates. As Perlich and Provost (2003) show, the choice of aggregation operator can have a much stronger impact on the resulting model's generalisation performance than the choice of the model induction method. Therefore, we believe it is worth investigating the use of complex aggregates in, e.g., systems as ICL (De Raedt & Van Laer, 1995) or ALEPH (Srinivasan, 2003). In the context of such rule learners however, attention should be paid to the refinement of aggregates which will not be automatically valid anymore, indeed, a local bottom-up search could interleave the top-down search strategy.

For first order learning, more than for propositional learning, one of the most important benefits of the use of random forests is the efficiency gain resulting from the sampling that is performed at each node of the trees. This allows to search through a larger hypothesis space. Nevertheless, the optimal boundary of the sample ratio needs to be explored further. Since the number of possible tests increases with the depth of the tree, one might want to vary the sample function throughout the tree building process, such that the number of tests in the sample is not increasing linearly with the number of tests in the nodes. Next to the square root, other sample functions such as logarithmic or even constant functions could be explored.

Another interesting direction for further work is to investigate the analogue of random forests for rule sets, which has to our knowledge not been studied extensively.

Acknowledgments Anneleen Van Assche is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (I.W.T.-Vlaanderen). Celine Vens is supported by the FWO-project on Probabilistic-Logical Learning. Hendrik Blockeel is Postdoctoral Fellow of the Fund for Scientific Research—Flanders (Belgium) (F.W.O.-Vlaanderen). Sašo Džeroski is supported by the Slovenian Research Agency through the research programme P2-103 "Knowledge Technologies".

We would like to thank Maurice Bruynooghe, for some useful discussions that have led to a number of important improvements to the text. We also owe our thanks to the reviewers for their valuable comments and suggestions.

References

Berka, P. (2000). Guide to the financial data set. In: A. Siebes & P. Berka (Eds.), *The ECML/PKDD 2000 Discovery Challenge*.

Blockeel, H., & Bruynooghe, M. (2003). Aggregation versus selection bias, and relational neural networks. In: *IJCAI-2003 Workshop on Learning Statistical Models from Relational Data, SRL-2003,* Acapulco, Mexico.

Blockeel, H., & De Raedt, L. (1997). Lookahead and discretization in ILP. In: *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, vol. 1297 of *Lecture Notes in Artificial Intelligence* (pp. 77–85), Springer-Verlag.

Blockeel, H., & De Raedt, L. (1998). Top-down induction of first order logical decision trees. *Artificial Intelligence*, *101*(1–2), 285–297.

Blockeel, H., Dehaspe, L., Demoen, B., Janssens, G., Ramon, J., & Vandecasteele, H. (2002) . Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research*, *16*, 135–166.

Breiman, L. (1996a). Bagging predictors. *Machine Learning*, *24*(2), 123–140.

Breiman, L. (1996b). Out-of-bag estimation. ftp.stat.berkeley.edu/pub/users/breiman/OOBestimation.ps.

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32.

de Castro Dutra, I., Page, D., Costa, V., & Shavlik, J. (2002). An empirical evalutation of bagging in inductive logic programming. In: *Proceedings of the 12th International Conference on Inductive Logic Programming*, vol. 2583 of *Lecture Notes in Computer Science* (pp. 48–65).

De Raedt, L., & Van Laer, W. (1995). Inductive constraint logic. In: K. P. Jantke, T. Shinohara, & T. Zeugmann (Eds.), *Proceedings of the Sixth International Workshop on Algorithmic Learning Theory*, vol. 997 of *Lecture Notes in Artificial Intelligence* (pp. 80–94), Springer-Verlag.

Dietterich, T. (2000). Ensemble methods in machine learning. In: *Proceedings of the 1th International Workshop on Multiple Classifier Systems*, vol. 1857 of *Lecture Notes in Computer Science* (pp. 1–15).

Džeroski, S., Schulze-Kremer, S., Heidtke, K. R., Siems, K., Wettschereck, D., & Blockeel, H. (1998). Diterpene structure elucidation from ^{13}C NMR spectra with inductive logic programming. *Applied Artificial Intelligence*, *12*(5), 363–384.

Emde, W., & Wettschereck, D. (1995). Relational instance based learning. In: *Proceedings of the 1995 Workshop of the GI Special Interest Group on Machine Learning*.

Freund, Y., & Schapire, R. E. (1996). Experiments with a new boosting algorithm. In: L. Saitta (Ed.), *Proceedings of the Thirteenth International Conference on Machine Learning* (pp. 148–156), Morgan Kaufmann.

Hansen, L., & Salamon, P. (1990). Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *12*, 993–1001.

Hoche, S., & Wrobel, S. (2001). Relational learning using constrained confidence-rated boosting. In: C. Rouveirol, & M. Sebag (Eds.), *Proceedings of the Eleventh International Conference on Inductive Logic Programming*, vol. 2157 of *Lecture Notes in Artificial Intelligence* (pp. 51–64), Springer-Verlag.

Knobbe, A., de Haas, M., & Siebes, A. (2001). Propositionalisation and aggregates. In: L. De Raedt, & A. Siebes (Eds.), *Proceedings of the 5th European Conference on Principles of Data Mining and Knowledge Discovery*, vol. 2168 of *Lecture Notes in Artificial Intelligence* (pp. 277–288), Springer.

Knobbe, A., Siebes, A., & Marseille, B. (2002). Involving aggregate functions in multi-relational search. In: *Principles of Data Mining and Knowledge Discovery, Proceedings of the 6th European Conference* (pp. 287–298), Springer-Verlag.

Koller, D. (1999). Probabilistic relational models. In: *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, vol. 1634 of *Lecture Notes in Artificial Intelligence* (pp. 3–13), Springer-Verlag.

Krogel, M.-A., Rawles, S., Železný, F., Flach, P., Lavrač, N., & Wrobel, S. (2003). Comparative evaluation of approaches to propositionalization. In: *Proceedings of the 13th International Conference on Inductive Logic Programming*, vol. 2835 of *Lecture Notes in Artificial Intelligence* (pp. 194–217), Springer-Verlag.

Krogel, M.-A., & Wrobel, S. (2001). Transformation-based learning using multi-relational aggregation. In: *Proceedings of the Eleventh International Conference on Inductive Logic Programming* (pp. 142–155).

Krogel, M.-A., & Wrobel, S. (2003). Facets of aggregation approaches to propositionalization. In: T. Horváth, & A. Yamamoto (Eds.), *Proceedings of the Work-in-Progress Track at the 13th International Conference on Inductive Logic Programming* (pp. 30–39).

Lavrač, N., & Džeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood.

Michalski, R. (1980). Pattern recognition as rule-guided inductive inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *2*, 349–361.

Muggleton, S. (Ed.) (1992). *Inductive Logic Programming*. Academic Press.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, *13*(3–4), 245–286.

Neville, J., Jensen, D., Friedland, L., & Hay, M. (2003). Learning relational probability trees. In: *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Perlich, C., & Provost, F. (2003). Aggregation-based feature invention and relational concept classes. In: *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 167–176), ACM Press.

Plotkin, G. (1969). A note on inductive generalization. *Machine Intelligence*, *5*, 153–163.

Provost, F. J., & Fawcett, T. (2001). Robust classification for imprecise environments. *Machine Learning*, *42*(3), 203–231.

Quinlan, J. (1990). Learning logical definitions from relations. *Machine Learning*, *5*, 239–266.

Quinlan, J. (1996). Boosting first-order learning. In: *Algorithmic Learning Theory, 7th International Workshop (ALT '96)*.

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann series in Machine Learning. Morgan Kaufmann.

Schapire, R. E., & Singer, Y. (1999). Improved boosting algorithms using confidence-rated predictions. *Machine Learning*, *37*(3), 297–336.

Srinivasan, A. (2003). The aleph manual. http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.

Srinivasan, A., King, R., & Bristol, D. (1999). An assessment of ilp-assisted models for toxicology and the PTE-3 experiment. In: *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, vol. 1634 of *Lecture Notes in Artificial Intelligence* (pp. 291–302), Springer-Verlag.

Srinivasan, A., Muggleton, S., & King, R. (1995). Comparing the use of background knowledge by Inductive Logic Programming systems. In: L. De Raedt (Ed.), *Proceedings of the Fifth International Workshop on Inductive Logic Programming* (pp. 199–230). Department of Computer Science, Katholieke Universiteit Leuven.

Uwents, W., & Blockeel, H. (2005). Classifying relational data with neural networks. In: *Proceedings of 15th International Conference on Inductive Logic Programming, Bonn, Germany*, vol. 3625 of *Lecture Notes in Artificial Intelligence* (pp. 384–396), Springer.