First Order Random Forests with Complex Aggregates

Celine Vens[1], Anneleen Van Assche[1], Hendrik Blockeel[1], and Sašo Džeroski[2]

[1] Department of Computer Science, Katholieke Universiteit Leuven,
Celestijnenlaan 200A, 3001 Leuven, Belgium
{celine,anneleen,hendrik}@cs.kuleuven.ac.be

[2] Department of Knowledge Technologies, Jozef Stefan Institute,
Jamova 39, 1000 Ljubljana, Slovenia
Saso.Dzeroski@ijs.si

Abstract. Random forest induction is a bagging method that randomly samples the feature set at each node in a decision tree. In propositional learning, the method has been shown to work well when lots of features are available. This certainly is the case in first order learning, especially when aggregate functions, combined with selection conditions on the set to be aggregated, are included in the feature space. In this paper, we introduce a random forest based approach to learning first order theories with aggregates. We experimentally validate and compare several variants: first order random forests without aggregates, with simple aggregates, and with complex aggregates in the feature set.

Keywords
Random forests, Aggregation, Decision Tree Learning

1 Introduction

Given the widespread use of multi-relational databases, relational learners are very important. Among the many approaches to relational learning that currently exist, an important distinction can be made with respect to how they handle one-to-many and many-to-many relationships. Whereas propositionalization approaches usually handle sets by aggregating over them, inductive logic programming (ILP) [20] techniques select specific elements. This imposes an undesirable bias on both kinds of learners [2].

In this paper we try to overcome this bias by introducing aggregate functions, possibly combined with selection conditions on specific elements, into an ILP setting. So far, this combination has been considered a difficult task, because the feature set grows quickly, and because the search space is less well-behaved due to non-monotonicity [16]. Our approach is based on random forests [8]. Random forest induction is a bagging method that builds decision trees using only a random subset of the feature set at each node.

The motivation for using random forests is based on two observations. First, because of the reduction of the feature space at each node and because of the search strategy used by decision trees, random forests seem to be very suitable to tackle the problems mentioned above. Second, in propositional learning, random forest induction has been shown to work well when many features are available [8]. Because ILP typically deals with large feature sets, it seems worthwhile to investigate whether random forests perform well in this context.

The paper is organized as follows. In Sect. 2, random forests are discussed. Section 3 illustrates the problem of combining aggregates with selection conditions in ILP. Our method, which is a first order random forest with complex aggregates, is presented in Sect. 4. In Sect. 5, we experimentally evaluate our method both in structurally complex domains, and in a (highly non-determinate) business domain. Finally, we formulate conclusions and some ideas for future research in Sect. 6.

2 Random Forests

Random forest induction [8] is an ensemble method. An ensemble learning algorithm constructs a set of classifiers, and then classifies new data points by taking a vote of the predictions of each classifier. A necessary and sufficient condition for an ensemble of classifiers to be more accurate than any of its individual members, is that the classifiers are accurate and diverse [14]. 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 for constructing ensembles: bagging [6] and boosting [13] 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 [10].

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*
- make predictions according to the majority vote of the set of k trees

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 $f(x) = \sqrt{x}$, ...), 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.

An advantage of using bagging is that out-of-bag estimates [7] can be used to estimate the generalization errors. This removes the need for a set aside test set or cross-validation. Out-of-bag estimation proceeds as follows: each tree is learned on a training set D_i, drawn with replacement from the original training set D. The trees vote over new examples to form the bagged classifier. For each example d in the original training set, the votes 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 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 [8]. 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 parallellized.

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, such data sets abound. Moreover, using random forests allows us to enlarge the feature set by including aggregate functions, possibly refined with selection conditions. We discuss this in the next section.

3 Aggregates and Selection in ILP

When considering multi-relational learning, sets (or more generally, bags) need to be handled. They are represented by one-to-many or many-to-many relationships in or between relations. Blockeel and Bruynooghe [2] categorize current approaches to relational learning with respect to how they handle sets. Whereas ILP [20] is biased towards selection of specific elements, other approaches, such as PRM's [17], or certain propositionalization approaches (e.g. the one by Krogel and Wrobel [18]) use aggregate functions, which summarize the set. The latter methods are optimized for highly non-determinate (business) domains, whereas ILP is geared more towards structurally complex domains, e.g. molecular biology, language learning,...

Perlich and Provost [23] present a hierarchy of relational concepts of increasing complexity, where the complexity depends on that of any aggregate functions used. They argue that ILP currently is the only approach that is able to explore concepts up to level four (out of five) in the hierarchy. However, until now, ILP-like feature construction and the use of aggregation features have mostly been combined in relatively restricted ways, e.g. without allowing complex conditions within the aggregation operator. The combination of both approaches is not a trivial task: the number of features grows quickly, and some issues with respect to monotonicity of aggregates arise. In the following, we present some related work on ILP systems that incorporate aggregate functions.

Krogel and Wrobel [18] present a system called RELAGGS (RELational AGGregationS), which builds on transformation-based approaches to ILP. RELAGGS computes several joins of the input tables according to their foreign key relationships. These joins are compressed using aggregate functions, specific to the data types of the table columns, such that there remains a single row for each example. The result is an appropriate input for conventional data mining algorithms, such as decision tree induction or support vector machines. However, the aggregation operators do not allow for complex selection conditions.

Relational Probability Trees (RPTs) [22] extend standard probability estimation trees to a relational setting. The algorithm for learning an RPT uses aggregation functions to dynamically propositionalize the data. However, as RPTs are not able to refer to a particular object throughout a series of conjunctions, again, no selection conditions can occur within the aggregate conditions.

We provide some definitions that are useful for our discussion [16]. An *aggregate condition* is a triple (f, o, v), where f is an aggregate function, o is a comparison operator, and v is a value of the domain of f. An aggregate condition is *monotone* if, given sets of records S and S', $S' \subseteq S, f(S')\ o\ v \Rightarrow f(S)\ o\ v$. In this paper, we denote the aggregate function f by a predicate with three arguments: the first one is the set to aggregate, the second one is the *aggregate query* that generates the set, and the third argument is the result of f. We illustrate the concepts with an example. Suppose we have the following query, which represents an aggregate condition on the *child* relation of a person X:

```
person(X), count(Y, child(X,Y), C), C>=2.
```

Adding a literal to the aggregate query gives

```
person(X), count(Y, (child(X,Y), female(Y)), C), C>=2,
```

which must have at most the same coverage (people with at least two daughters must be a subset of people with at least two children), so $(count, \geq, v)$ is a monotone aggregate condition. However, if we consider the following query

```
person(X), count(Y, child(X,Y), C), C<2,
```

and its refinement

```
person(X), count(Y, (child(X,Y), female(Y)), C), C<2,
```

then the examples covered by the refinement are a superset of the original set (people with less than two children certainly have less than two daughters, but people with less than two daughters may have more than two children), so $(count, <, v)$ is anti-monotone. The following aggregate conditions are monotone: $(count, \geq, v)$, $(max, \geq, v)$, $(min, \leq, v)$. By a *non-monotone* aggregate condition, we mean an aggregate condition that is not monotone, either because it is anti-monotone, or because it is neither monotone nor anti-monotone (such as $(mode, =, v)$, $(avg, \geq, v)$ or $(avg, \leq, v)$).

Knobbe et al. [16] present an approach to combining aggregates with reasonably complex selection conditions by generalizing their selection graph pattern language. Selection graphs are a graphical description of sets of objects in a multi-relational database. In order to have only refinements that reduce the coverage, they restrict refinement to monotone aggregate conditions.

In Sect. 4 we will discuss an approach that does not prohibit the refinement of non-monotone aggregate conditions.

4 First Order Random Forests with Complex Aggregates

This section discusses our approach, which is a first order random forest with (complex) aggregates in the feature space. The section is started by discussing first order decision trees (Sect. 4.1). We continue by explaining how (complex) aggregate conditions are added to the feature space (Sect. 4.2). Finally, we show how the first order decision tree was upgraded to a random forest (Sect. 4.3).

4.1 First Order Decision Trees

The base decision tree induction algorithm in our random forest is Tilde [4], which is included in the ACE data mining system [5]. Tilde is a relational top-down induction of decision trees (TDIDT) instantiation, and outputs a first order decision tree.

A first order decision tree [4] 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 [25]. 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 Fig. 1.

The OPTIMAL_SPLIT procedure returns a conjunction Q_b, which is selected from a set of candidates generated by the refinement operator ρ, using a heuristic such as information gain. The refinement operator typically 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. 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.

```
procedure GROW_TREE ($E$: set of examples, $Q$: query):
    $candidates := \rho(\leftarrow Q)$
    $\leftarrow Q_b :=$ OPTIMAL_SPLIT($candidates, E$)
    if STOP_CRIT ($\leftarrow Q_b, E$)
    then
        $K :=$ MOST_FREQUENT_CLASS($E$)
        return leaf($K$)
    else
        $conj := Q_b - Q$
        $E_1 := \{e \in E | \leftarrow Q_b$ succeeds in $e \wedge B\}$
        $E_2 := \{e \in E | \leftarrow Q_b$ fails in $e \wedge B\}$
        $left :=$ GROW_TREE ($E_1, Q_b$)
        $right :=$ GROW_TREE ($E_2, Q$)
        return node($conj, left, right$)
```

Fig. 1. Algorithm for first order logical decision tree induction [4]

4.2 First Order Decision Trees with Complex Aggregates

Tilde was modified to include (complex) aggregate conditions. We first discuss how these aggregates were added. Next, we focus on some issues regarding non-monotone aggregate conditions.

(Complex) Aggregate Conditions in Decision Trees. The feature set considered at each node in the tree was expanded to consist of the regular features, augmented with aggregate conditions (both simple and complex ones). By *complex* aggregate conditions, we mean aggregate conditions with selection conditions on the set to be aggregated. This results in aggregate queries having more than one literal. Of course, complex aggregate conditions could have been included by just declaring them as intentional background knowledge, but this presumes a large expertise in the problem domain. The main difficulty is that the aggregate query is itself the result of a search through some hypothesis space.

Our method works as follows. To include aggregates, the user needs to specify the basic ingredients in the language bias: the aggregate functions to be used as well as some sets to be aggregated, together with a query to generate them. The system then constructs simple aggregate conditions, using the components provided by the user, and using for example discretization [3] to obtain a number of values to compare the result with. The refinement operator ρ includes the aggregate conditions in the set of candidate queries it generates.

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, using only the regular features. Each candidate generated by ρ_{inn} is included in an aggregate condition, which is then considered as a candidate of ρ. 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(X), count(Y, child(X,Y), C), C>4,
```

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

```
person(X), count(Y, child(X,Y), C), C>4,
        count(Y', (child(X,Y'),female(Y')), C'), C'>4.
```

If the query above is chosen by the OPTIMAL_SPLIT procedure, then

```
count(Y', (child(X,Y'),female(Y')), C'), C'>4
```

is the conjunction added in the left childnode of n.

The second way to build complex aggregates is based on lookahead [3], 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 $k+1$ 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 conjunction

```
count(Y', (child(X,Y'),female(Y')), C'), C'>4
```

could immediately be inserted, without having the conjunction

```
count(Y, child(X,Y), C), C>4
```

in one of its ancestor nodes. Obviously, this technique is computationally expensive, but it will turn out to yield significant improvements.

Non-monotone Aggregate Conditions in Decision Trees. Knobbe et al. [16] leave non-monotone aggregate conditions untouched, in order to have only refinements that reduce the coverage. In decision trees however, a refinement of a node that does not result in a specialization will never be chosen, since such a test will hold for all examples in the node and hence will not reduce entropy. Therefore, there is no need to explicitly restrict ourselves to monotone aggregate conditions. Moreover, allowing non-monotone aggregate conditions may lead to a number of meaningful refinements when adding selection conditions. We discuss three examples.

First, remark that in the propositional case, the set of objects to aggregate can only become smaller when a new conjunct is added. In relational learning however, this is not the case when the new conjunct is introducing a new relation in the aggregate query. In that case, the result is a bag instead of a set. In fact, the bag of objects is then generated by the cartesian product of the different relations in the aggregate query. This may change the outcome of certain aggregate functions, such as count and sum. Hence, the refinement of e.g. the anti-monotone $(count, <, v)$ may yield a specialization. For example, consider a node n with the following query:

```
person(X), count(Y, child(X,Y), C), C<5.
```

A possible refinement of this query would be obtained by introducing a new relation (e.g. *horse*) within the aggregate query. This could lead to the following conjunction being entered at the left child node of n:

```
count(Y', (child(X,Y'),horse(X,Z)), C'), C'<5.
```

The refinement results in the cartesian product of the relations *child*(X,Y) and *horse*(X,Z). The aggregate condition evaluates to true for the following examples: persons with three or four children and maximum one horse, persons with two children and maximum two horses, persons with one child and maximum four horses, and persons with zero children and any number of horses. Thus, the introduction of a new relation into the aggregate condition $(count, <, v)$ yields a refinement that reduces the coverage.

Second, depending on the multiplicity of a relation, the set to be aggregated may be empty, i.e. the aggregate query generates no instantiations of the objects to be aggregated. This is naturally treated by aggregate functions such as count; however, other aggregate functions such as min, max and average, fail when dealing with an empty set. So in that case, examples with multiplicity 0 for that relation always end up in the right ("no") branch of the tree. This means that for these aggregate functions, although the aggregate condition is non-monotone, refinements can be found that reduce the coverage. For example, consider the following query in a certain node in the tree

```
person(X), max(A, (child(X,Y),age(Y,A)), M), M<12.
```

This node can then be refined by adding

```
max(A', (child(X,Y'),female(Y'),age(Y',A')), M'), M'<12
```

as a left-child, which is equivalent to adding a selection condition outside the aggregate condition:

```
child(X,Z), female(Z),
```

since both alternatives fail for all persons without daughters. Note that this kind of refinement can also occur with monotone aggregate conditions.

Third, the use of lookahead, in combination with anti-monotone aggregate conditions, allows to simulate a bottom-up search strategy included in our top-down decision tree induction method, which may lead to useful refinements. We illustrate this with an example. Say the maximum depth for lookahead is two. Assume that the following aggregate condition, which counts the number of blonde daughters of a person, occurs in a node of the tree:

```
person(X), count(Y, (child(X,Y),female(Y),blonde(Y)), C), C<4.
```

Since at each node all aggregate conditions with up to three conjuncts are in the feature set, the following conditions will be generated:

```
count(Y', child(X,Y'), C'), C'<4,
count(Y', (child(X,Y'),female(Y')), C'), C'<4,
count(Y', (child(X,Y'),blonde(Y')), C'), C'<4.
```

These conditions indeed reduce the coverage and hence, lead to valid refinements.

4.3 First Order Random Forests with Complex Aggregates

In order to upgrade Tilde with complex aggregates to a first order random forest, 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[3]. As a result, constructing a new node proceeds as follows: first all possible candidates $\rho(\leftarrow Q)$ are generated, then a random subset of approximate size $f(|candidates|)$ (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 need to be executed on the examples to calculate the heuristics, which obviously results in an efficiency gain.

Finally, we added another selection technique that could further increase diversity among the trees, and might improve the strength of the random forest. Instead of always selecting the optimal query at each node in a tree (out of the queries in the random sample), queries can be chosen according to a certain distribution. This way, even less good queries have a chance to be chosen; this chance being proportional to their quality.

To summarize, we provide an overview of the resulting algorithm in Fig. 2. The procedure GROW_FOREST takes the number of trees to grow as one of its input parameters (N). 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 Fig. 1). The refinement operator ρ includes (complex) aggregate conditions, as discussed in Sect. 4.2. The SUBSET procedure generates a random subset of the candidate set. Each candidate has probability P to be chosen. P can take any number between zero and one, it can also be $1/sqrt(|candidates|)$. The GET_SPLIT procedure either returns the optimal split, or returns a split with probability proportional to its quality, as explained above.

5 Experimental Results

In this section, we 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. We start by describing our experimental setup (Sect. 5.1). Next, we present results on three well-known data sets: Mutagenesis [26] (Sect. 5.2), Diterpenes [11] (Sect. 5.3), and Financial [1] (Sect. 5.4). The first two data sets contain complex structures and have been widely used as ILP benchmarks. The latter is a business domain data set with high degree of non-determinacy. Finally, we formulate some experimental conclusions (Sect. 5.5).

[3] This actually differs from the definition in [8] where a random subset of the attributes, instead of the tests, is chosen. Note that one attribute may yield different tests.

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

procedure GROW_TREE_2 (E: set of examples, Q: query, P: probability):
    $candidates := \rho(\leftarrow Q)$
    $candidates\_subset$ := SUBSET($candidates, P$)
    $\leftarrow Q_b$ := GET_SPLIT($candidates\_subset, E$)
    if STOP_CRIT ($\leftarrow Q_b, E$)
    then
        $K$ := MOST_FREQUENT_CLASS($E$)
        return leaf($K$)
    else
        $conj := Q_b - Q$
        $E_1 := \{e \in E | \leftarrow Q_b$ succeeds in $e \wedge B\}$
        $E_2 := \{e \in E | \leftarrow Q_b$ fails in $e \wedge B\}$
        $left$ := GROW_TREE_2 ($E_1, Q_b, P$)
        $right$ := GROW_TREE_2 ($E_2, Q, P$)
        return node($conj, left, right$)
```

Fig. 2. Algorithm for first order random forest induction

5.1 Experimental Setup

All experiments with random forests were performed using out-of-bag estimation (unless otherwise stated) and were carried out five times, in order to obtain a more reliable estimate of the performance. The predictive performance is compared to that of Tilde, obtained by averaging five runs of tenfold cross-validation. We used the basic form of Tilde, i.e. without aggregate conditions.

Different parameters needed to be set. First of all, the number of random features in each node: we chose to consider random subsets of 100%, 75%, 50%, 25%, 10%, and the square root of the number of tests at each node in the trees. We have also examined the influence of the number of trees in the random forests, experimenting with 3, 11, and 33 trees.

To investigate the performance of first order random forests in the context of aggregation, we have performed experiments using 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). The aggregate functions used were *min*, *max* and *avg* for continuous attributes, *mode* for discrete attributes, and *count*.

If, for a certain data set, results are available for other systems, we also compared the performance of FORF to that of the other systems.

5.2 Mutagenesis

For our first experiment, we used the Mutagenesis data set. This ILP benchmark data set, introduced to the ILP community by Srinivasan et al. [26], consists of structural descriptions of 230 molecules that are to be classified as mutagenic (60%) or not. The description consists of the atoms and the bonds that make up the molecule. Since the data is not very non-determinate and does not contain many numerical attributes, we expected only a slight gain using aggregates.

Predictive accuracies are shown in Table 1. 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(BId,(bond(BId,Mol,At1,At2,Tp)),C), C>28.
```

with the following conjunction in its left child

```
count(BId,(bond(BId,Mol,At1,At2,Tp),atom(Mol,At1,carbon)),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 et al. [16]. The refinement of the aggregate describes all molecules that have at least 28 bonds connected to an atom of type carbon. Unfortunately, we can not compare our results to those given by Knobbe et al. [16], since they only report predictive accuracies for one best rule, covering only a part of the positive examples.

Table 1. Accuracy results on the Mutagenesis data set. The rows indicate the sample ratio at each node. The columns compare predictive accuracies for FORF-LA, FORF-RA, FORF-SA, FORF-NA, and Tilde. The standard deviation is indicated between parentheses.

	FORF (33 trees)				Tilde
	LA	RA	SA	NA	NA
1	0.779 (0.008)	0.774 (0.014)	0.777 (0.011)	0.731 (0.010)	0.720 (0.007)
0.75	0.774 (0.013)	0.775 (0.010)	0.764 (0.017)	0.720 (0.015)	
0.50	0.777 (0.012)	0.781 (0.012)	0.789 (0.018)	0.747 (0.014)	
0.25	0.770 (0.013)	0.772 (0.016)	0.758 (0.011)	0.736 (0.014)	
0.10	0.790 (0.014)	0.765 (0.017)	0.761 (0.013)	0.653 (0.015)	
sqrt	0.785 (0.016)	0.752 (0.010)	0.743 (0.016)	0.690 (0.027)	

5.3 Diterpenes

In our second experiment, we used the Diterpenes data set [11]. The data contains information on 1503 diterpenes with known structure, stored in three relations. The *atom* relation specifies to which element an atom in a given compound belongs. The *bond* relation specifies which atoms are bound in a given compound. The *red* relation stores the measured NMR-Spectra. For each of the 20 carbon atoms in the diterpene skeleton, it contains the multiplicity and frequency. Additional unary predicates describe to which of the 25 classes a compound belongs.

Table 2 gives predictive accuracies of experiments using 33 trees. For efficiency reasons, we changed the minimal cases a leaf has to cover from 2 to 20.

Table 2. Accuracy results on the Diterpenes data set. The rows indicate the sample ratio at each node. The columns compare predictive accuracies for FORF-LA, FORF-RA, FORF-SA, FORF-NA, and Tilde. The standard deviation is indicated between parentheses.

	FORF (33 trees)				Tilde
	LA	RA	SA	NA	NA
1	0.859 (0.004)	0.829 (0.011)	0.849 (0.003)	0.768 (0.003)	0.737 (0.011)
0.75	0.859 (0.003)	0.840 (0.002)	0.849 (0.004)	0.769 (0.002)	
0.50	0.856 (0.006)	0.839 (0.004)	0.850 (0.001)	0.766 (0.004)	
0.25	0.856 (0.004)	0.849 (0.007)	0.847 (0.004)	0.763 (0.004)	
0.10	0.853 (0.004)	0.823 (0.012)	0.842 (0.002)	0.739 (0.007)	
sqrt	0.844 (0.005)	0.806 (0.004)	0.824 (0.006)	0.716 (0.004)	

The numbers in Table 2 do not compare favorably with earlier published results [11], but the experimental setting is also different; especially the minimal leaf size of 20 seems to have a detrimental effect. To allow a better comparison with published results, we performed a single experiment where, just like in [11], results of five runs of a tenfold cross-validation 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 size of 25%, which, judging from our earlier results, are good parameter values. The result of this experiment is compared with published results for other systems in Table 3. FOIL [24] induces concept definitions represented as function-free Horn clauses, from relational data. RIBL [12] is a relational instance based learning algorithm. It generalizes the nearest neighbor method to a relational representation. The ICL system [9] uses exactly the same representation of training examples as Tilde, but induces rule sets instead of trees. It was already found that combining propositional (aggregate) features with relational information yielded the best results [11]. Comparing with those best results, we see that FORF is at least competitive with the best of the other approaches. (Unavailability of standard deviations for the published results makes it difficult to test the significance of the improvement of FORF-SA over RIBL.)

Table 3. Accuracy results on the Diterpenes data set compared to other systems. The results for FOIL, RIBL, and ICL are obtained from [11]. No standard deviations were given.

FORF-SA	FOIL	RIBL	ICL
0.928 (0.006)	0.783	0.912	0.860

5.4 Financial

Our last experiment deals with the Financial data set, originating from the discovery challenge organized at PKDD '99 and PKDD '00 [1]. This data set involves learning to classify 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 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.

Predictive accuracies are shown in Table 4. In this case the square root of the number of tests is on average larger than 10%, so we switched these two rows.

Table 4. Accuracy results on the Financial data set. The rows indicate the sample ratio at each node. The columns compare predictive accuracies for FORF-LA, FORF-RA, FORF-SA, FORF-NA, and Tilde. The standard deviation is indicated between parentheses.

	FORF (33 trees)				Tilde
	LA	RA	SA	NA	NA
1	0.986 (0.009)	0.989 (0.005)	0.990 (0.005)	0.845 (0.005)	0.790 (0.027)
0.75	0.991 (0.005)	0.991 (0.005)	0.990 (0.002)	0.834 (0.010)	
0.50	0.991 (0.004)	0.992 (0.004)	0.992 (0.004)	0.846 (0.012)	
0.25	0.991 (0.006)	0.995 (0.006)	0.991 (0.002)	0.853 (0.011)	
sqrt	0.984 (0.010)	0.988 (0.004)	0.993 (0.009)	0.854 (0.004)	
0.10	0.975 (0.011)	0.972 (0.012)	0.964 (0.020)	0.864 (0.004)	

Table 5 shows predictive accuracies compared to other systems [18]. DINUS-C [19] is a propositionalisation technique using only determinate features and using C4.5 rules as propositional learner. RELAGGS [18] was discussed in Sect. 3. PROGOL [21] is an ILP learner capable of learning in structurally very complex domains. For our random forest, we used the SA setting, since neither LA nor RA yielded significantly better results for this data set. The forest contained 33 trees and a sampling size of 25%. For better comparisions, we averaged five runs of tenfold cross-validation instead of out-of-bag estimations.

Table 5. Accuracy results on the Financial data set compared to other systems. The results for DINUS-C, RELAGGS, and PROGOL are obtained from [18]. 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)

5.5 Experimental Conclusions

As can be seen from Tables 1, 2 and 4, the use of simple aggregates yields quite a large improvement on all data sets. As was expected, the gain is largest on the Financial data set (about 15%). Refinement of the aggregate conditions increased accuracy in some cases, while the use of lookahead within the aggregate query in general added another slight performance improvement.

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 drop in accuracy (no significant gain either) and we profit from the efficiency gain by testing fewer features in each node. Breiman [8] on the other hand, obtained higher improvements with even much smaller proportions of features in the propositional case. This difference might be 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. However, this remains to be investigated.

Random forests indeed seem to profit from a large feature space, which is certainly present when using lookahead in the aggregate queries. While for FORF-NA, FORF-SA and FORF-RA accuracy slightly tends to decrease when using smaller samples of features (an exception to that is the Financial data set, where for FORF-NA the accuracy tends to increase by decreasing the number of features due to the skewness of the data), this is less the case for FORF-LA.

As can be seen from Fig. 3, adding more trees to the forest clearly increased accuracy in all experiments. Of course more trees mean longer runtimes, so there is still a trade-off between efficiency and accuracy.

Concerning the complexity of the trees in the random forests, we found that both adding aggregate conditions and further refining them, strongly decreased the size of the trees for all experiments. For example, for the Financial data set FORF-NA produced trees with on average 17.65 nodes, while FORF-SA yielded trees with on average 4.72 nodes. Applying lookahead to the aggregate queries slightly increased the number of nodes in each tree, while still being much smaller than the number of nodes in each tree of FORF-NA.

While in general performing at least as well as bagging, random forests are computationally more efficient. If we compare random forests, consisting of 33 trees and using 25% of the features, with bagging using 33 trees, we found an efficiency gain of factor 1.3 to 2.6 over the different data sets. Again, this factor is smaller than in the propositional case, where there is no need to generate tests at each node, since the set of tests remains the same. On the Mutagenesis data and the Diterpenes data, computation times of FORF-NA, FORF-SA and FORF-RA are quite in the same range (i.e. a few minutes), on the Financial data on the other hand, computation times of FORF-NA and FORF-RA might differ up to a factor 5 (i.e. from seconds to minutes). As was expected, applying lookahead resulted in quite an explosion of features which couldn't be offset by the speed-up due to feature sampling. Comparing runtimes of FORF-RA and FORF-LA gives differences of factor 10 to 70 (i.e. from minutes to a few hours).

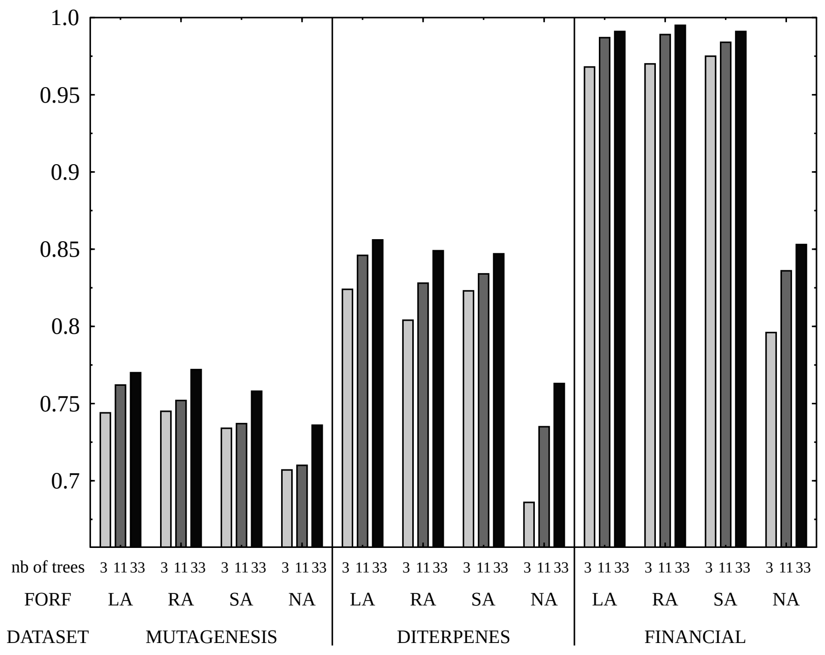

Fig. 3. Accuracy for random forests using 25% of the features. Results are shown for FORF-LA, FORF-RA, FORF-SA and FORF-NA on the three different data sets.

In order to compensate for these huge runtimes, we certainly need to reduce the time for generating the queries.

Another observation concerns the effect of proportional selection of a test in the GET_SPLIT procedure. We repeated our experiments using this mechanism. The difference in predictive accuracy between selecting the optimal test and proportionally selecting a test did not turn out to be statistically significant. Remark that when using aggregate functions, features may have several logical equivalent notations. Thus, when using a large set of features at each node (e.g. with a sample ratio of 1.0 or 0.75) it is likely that several equivalent versions of the best test are in the feature set. Consequently, even using proportional selection, this feature has a high chance to be chosen. When using only a small random sample of tests at each node, proportional selection did not seem to add any improvement over the randomization.

To conclude this discussion, we compared the results of FORF-SA to available results of a number of other systems that either include aggregates themselves or use a language bias with user predefined aggregate functions. FORF-SA clearly outperformed these systems on our chosen data sets (see Tables 3 and 5).

6 Conclusions and Future Work

In this paper we have presented a first order random forest induction algorithm. Random forest induction [8] is a bagging method that randomly samples the feature set at each node in a decision tree. In the propositional case, random forests have been shown to work well when lots of features are available. As this is often the case in relational learning, it seems worthwhile to investigate the strength of random forests in this context.

Moreover, using random forests, the feature space can be further enlarged by including aggregate conditions, possibly combined with selection conditions on the set to be aggregated. This combination can occur when refining an aggregate condition, or when applying a lookahead mechanism within the aggregate query. The resulting aggregate conditions are called complex ones.

The combination of aggregation and selection is not a trivial task, because the feature space grows quickly and because the search space is less well-behaved due to non-monotonicity of some aggregate conditions. However, the use of random forests overcomes these problems. First, the feature set considered at each node in a tree is reduced by the sampling procedure. Second, the divide and conquer approach used in decision tree induction algorithms allows to include non-monotone aggregate conditions, and as we showed, these can lead to some useful refinements. Moreover, non-monotone aggregate conditions can, in combination with lookahead, simulate a bottom-up search strategy included in the top down decision tree induction approach.

To implement the above ideas, we used Tilde [4] as a base first order decision tree induction algorithm. Tilde was upgraded to a first order random forest with complex aggregates in the feature space.

The strength of our first order random forest approach and the use of (complex) aggregates were experimentally validated. Our results show that, if we randomly decrease the feature set at each node in a tree down to a certain level, the classification performance is at least as good as bagging, thus we profit from the efficiency gain. In our data sets the optimal level turned out to be 25% of the features; below this threshold classification performance decreased. The benefit of including simple aggregate functions was clearly shown. Using complex aggregate functions yielded another small performance improvement. Furthermore, our approach clearly outperformed existing systems on our chosen data sets.

Now we focus on two possible directions for future work. The first and most important direction is to make the forest induction algorithm more efficient, in order to make it more generally applicable. Efficiency gain can be obtained by generating only a sample of all queries at each node. This could be done by using a probabilistic grammar to represent the language of possible tests. At each node in a tree, a random sample of the features from the probabilistic grammar could be taken. This way, features would be generated probabilistically, and runtimes would be further reduced. By doing this, we would move closer towards Breiman's approach [8], where attributes are selected instead of tests.

A second direction is to investigate the effect of degree disparity [15] in the context of random forests. Degree disparity occurs when the frequency of a relation is correlated with the values of the target variable. This can cause learning algorithms to be strongly biased toward certain features, when using aggregate functions. Consequently, this may influence the extent to which different trees in the forest are independent.

Acknowledgements

Celine Vens is supported by the GOA/2003/08(B0516) on Inductive Knowledge Bases. Anneleen Van Assche is supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (I.W.T.-Vlaanderen). Hendrik Blockeel is Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.-Vlaanderen).
We also would like to thank Maurice Bruynooghe, for some useful discussions that have led to a number of important improvements to the text.

References

1. P. Berka. Guide to the financial data set. In A. Siebes and P. Berka, editors, *The ECML/PKDD 2000 Discovery Challenge*, 2000.
2. H. Blockeel and M. Bruynooghe. Aggregation versus selection bias, and relational neural networks. In *IJCAI-2003 Workshop on Learning Statistical Models from Relational Data, SRL-2003, Acapulco, Mexico, August 11, 2003*, 2003.
3. H. Blockeel and L. De Raedt. Lookahead and discretization in ILP. In *Proceedings of the Seventh International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Artificial Intelligence*, pages 77–85. Springer-Verlag, 1997.
4. H. Blockeel and L. De Raedt. Top-down induction of first order logical decision trees. *Artificial Intelligence*, 101(1-2):285–297, June 1998.
5. H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Improving the efficiency of inductive logic programming through the use of query packs. *Journal of Artificial Intelligence Research*, 16:135–166, 2002.
6. L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.
7. L. Breiman. Out-of-bag estimation. ftp.stat.berkeley.edu/pub/users/breiman/OOBestimation.ps, 1996.
8. L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.
9. L. De Raedt and W. Van Laer. Inductive constraint logic. In K. P. Jantke, T. Shinohara, and T. Zeugmann, editors, *Proceedings of the Sixth International Workshop on Algorithmic Learning Theory*, volume 997 of *Lecture Notes in Artificial Intelligence*, pages 80–94. Springer-Verlag, 1995.
10. T. Dietterich. Ensemble methods in machine learning. In *Proceedings of the 1th International Workshop on Multiple Classifier Systems*, volume 1857 of *Lecture Notes in Computer Science*, pages 1–15, 2000.
11. S. Džeroski, S. Schulze-Kremer, K. R. Heidtke, K. Siems, D. Wettschereck, and H. Blockeel. Diterpene structure elucidation from ^{13}C NMR spectra with inductive logic programming. *Applied Artificial Intelligence*, 12(5):363–384, July-August 1998.
12. W. Emde and D. Wettschereck. Relational instance based learning. In *Proceedings of the 1995 Workshop of the GI Special Interest Group on Machine Learning*, 1995.

13. Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In L. Saitta, editor, *Proceedings of the Thirteenth International Conference on Machine Learning*, pages 148–156. Morgan Kaufmann, 1996.
14. L. Hansen and P. Salamon. Neural network ensembles. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12:993–1001, 1990.
15. D. Jensen, J. Neville, and M. Hay. Avoiding bias when aggregating relational data with degree disparity. In *Proceedings of the 20th International Conference on Machine Learning*, 2003.
16. A. Knobbe, A. Siebes, and B. Marseille. Involving aggregate functions in multi-relational search. In *Principles of Data Mining and Knowledge Discovery, Proceedings of the 6th European Conference*, pages 287–298. Springer-Verlag, August 2002.
17. D. Koller. Probabilistic relational models. In *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 3–13. Springer-Verlag, 1999.
18. M.-A. Krogel and S. Wrobel. Transformation-based learning using multi-relational aggregation. In *Proceedings of the Eleventh International Conference on Inductive Logic Programming*, pages 142–155, 2001.
19. N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications.* Ellis Horwood, 1994.
20. S. Muggleton, editor. *Inductive Logic Programming.* Academic Press, 1992.
21. S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
22. J. Neville, D. Jensen, L. Friedland, and M. Hay. Learning relational probability trees. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003.
23. C. Perlich and F. Provost. Aggregation-based feature invention and relational concept classes. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 167–176. ACM Press, 2003.
24. J. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
25. J. R. Quinlan. *C4.5: Programs for Machine Learning.* Morgan Kaufmann series in Machine Learning. Morgan Kaufmann, 1993.
26. A. Srinivasan, R. King, and D. Bristol. An assessment of ILP-assisted models for toxicology and the PTE-3 experiment. In *Proceedings of the Ninth International Workshop on Inductive Logic Programming*, volume 1634 of *Lecture Notes in Artificial Intelligence*, pages 291–302. Springer-Verlag, 1999.