CARAF: Complex Aggregates within Random Forests

Clément Charnay[1], Nicolas Lachiche[1], and Agnès Braud[1]

ICube, Université de Strasbourg, CNRS
300 Bd Sébastien Brant - CS 10413, F-67412 Illkirch Cedex
{charnay,nicolas.lachiche,agnes.braud}@unistra.fr

Abstract. This paper presents an approach integrating complex aggregate features into a relational random forest learner to address relational data mining tasks. CARAF, for Complex Aggregates within RAndom Forests, has two goals. Firstly, it aims at avoiding exhaustive exploration of the large feature space induced by the use of complex aggregates. Its second purpose is to reduce the overfitting introduced by the expressivity of complex aggregates in the context of a single decision tree. CARAF compares well on real-world datasets to both random forests based on the propositionalization method RELAGGS, and the relational random forest learner FORF.

1 Introduction and Context

Relational data mining, as opposed to attribute-value learning, refers to learning from data represented across several tables. These tables represent different objects, linked by relationships. Many datasets from many domains fall into the relational paradigm, leading to a much richer representation. The applications go from the molecular domain, to geographical data, and any kind of spatio-temporal data such as speech recognition.

The difference to attribute-value learning is the one-to-many relationship. In particular, we focus on a two-table setting: one table, the main table, represents the objects we want to learn on. The second table, referred to as the secondary table, contains objects related to the main ones in a one-to-many relationship, which means several secondary objects are linked to one main object. In practice, many datasets are represented in this two-table setting: sequential data is represented as a main table containing information on the sequence, while the secondary table contains the elements of the sequence. The multi-dimensional setting is another use case, where one is often interested in learning on one dimension based on the contents of the table of facts, which are linked through a one-to-many relationship.

As an example, the relational schema for the Auslan dataset, an Australian sign language recognition task, is given in Figure 1. The main table, associated to records, contains only the attribute to learn, i.e. the language sign associated to the record, while the secondary table contains the records for all 22 channels monitored, and the timestamp attribute.

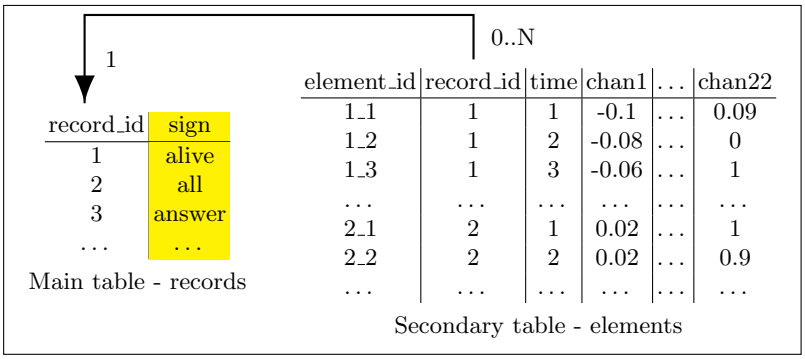

Fig. 1. Schema of the real-world Auslan dataset

Most relational data mining algorithms are based on inductive logic programming concepts, and handle the relationships through the use of the existential quantifier: it introduces a secondary object B linked to the main object A, B usually meets a certain condition and its existence is relevant to classify A. For instance, on the Auslan dataset, to discriminate between signs, a feature like the fact that an element of the record has a value higher than 0.9 for channel 13 could be useful. TILDE [2] is a relational extension of Quinlan's C4.5 [10] decision tree learner based on this idea. Other approaches use aggregates: they take all B objects linked to A, and aggregate the set to one value, for instance by computing the average of a numerical property of the B objects. For instance, the average value of channel 9 over the whole record may help discriminate between signs. The propositionalization approach RELAGGS [8] introduces such aggregates.

One approach combines both, by filtering the B objects on a condition before aggregating them. This approach is known as complex aggregation. As opposed to simple aggregation, it consists in aggregating a subset of the B objects linked to A, the subset being defined by a conjunction of conditions over the attributes of the secondary table. For instance, a feature that may be useful to classify signs could be the average value of channel 9 over record elements between timestamps 15 and 22. TILDE has been extended to handle complex aggregates [12], although it is not able to introduce more than one condition on the secondary objects to be aggregated. The RRHCCA algorithm [4] handles complex aggregates by a stochastic, random-restart based, hill-climbing search in the feature space.

However, the complex aggregates introduce two specific challenges: firstly, the introduction of a condition prior to the aggregation increases exponentially the size of the search space, which makes an exhaustive exploration intractable. Secondly, the complex aggregates, being a very rich representation, are also very specific and strict, which implies they are prone to overfitting. Especially, complex aggregate-based algorithms consider also the simple aggregates that RELAGGS builds. Therefore, if the performance of a decision tree built on top of simple-aggregate features is better than the performance of a decision tree

based on complex aggregates, it means the complex aggregates have been found to be better than simple aggregates on the training set, but this is not confirmed on test data. In other words, if complex aggregates do not perform better than RELAGGS, it means they overfit.

In this paper, we propose extending the decision tree learner based on RRHCCA to a random forest learner, introducing two faster hill-climbing algorithms. The implementation has a SQL-based version, which constitutes a first step towards handling bigger datasets.

The rest of the paper is organized as follows: in Section 2 we briefly define the concept of complex aggregates. In Section 3, we review the use of random forests in the relational setting. In Section 4, we introduce CARAF (Complex Aggregates with RAndom Forests), a new relational random forest learner implementing our contributions. In Section 5, we present experimental results obtained with CARAF. Finally, in Section 6, we conclude and give some future work perspectives.

2 Complex Aggregates

In this section, we briefly define the concept of complex aggregates, which has been thoroughly explained in [4].

In a setting with two tables linked through a one-to-many relationship, let us denote the main table by M and the secondary table by S. We define a complex aggregate feature of table M as a triple *(Selection, Feature, Function)* where:

- Selection selects the objects to aggregate. It is a conjunction of s conditions, i.e. $Selection = \bigwedge_{1 \leq i \leq s} c_i$, where c_i is a condition on a descriptive attribute of the secondary table. Formally, let $S.A$ be the set of descriptive attributes of table S, and $Attr \in S.A$, then c_i is:
  - either $Attr \in vals$ with $vals$ a subset of the possible values of $Attr$ if $Attr$ is a categorical feature,
  - or $Attr \in [val_1; val_2[$ if $Attr$ is a numerical feature.

  In other words, for a given object of the main table, the objects of the secondary table that meet the conditions in *Selection* are selected for aggregation.
- *Feature* can be:
  - nothing,
  - a descriptive attribute of the secondary table, i.e. $Feature \in S.A$.

  Thus, *Feature* is the attribute of the selected objects that will be aggregated. It can be nothing since the selected objects can simply be counted, in which case a feature to aggregate is not needed.
- *Function* is the aggregation function to apply to the bag of feature values for the selected objects. In this work we will consider six aggregation functions: *count, min, max, sum, mean* and *standard deviation*.

In the rest of the paper, we will denote a complex aggregate by *Function(Feature, Selection)*. We will refer to the set of possible $(Function, Feature)$ pairs as the aggregation processes, i.e. the different possibilities to aggregate a set of secondary objects.

The introduction of a condition on the objects to aggregate makes the feature space impossible to explore exhaustively. Heuristics have been proposed to explore this space in a smart way. The refinement cube [12] is based on the idea of the monotonicity of the dimensions of the cube. Indeed, the aggregation condition, aggregation function and threshold can be explored in a general-to-specific way, using monotone paths: when a complex aggregate (a point in the refinement cube) is too specific (*i.e.* it fails for every training example), the search does not restart from this point.

The RRHCCA algorithm [4] has been proposed to explore a larger search space with a random-restart hill-climbing approach to find the appropriate condition with respect to the aggregation process, still in the context of a decision tree learner. However, the decision tree model with complex aggregates often fails to outperform RELAGGS, which shows overfitting. As a solution, we propose its extension to a Random Forest model.

3 Random Forests

Random Forest [3] is an ensemble classification technique which builds a set of diverse decision trees and combines their predictions into a single output. Diversity between the trees is achieved by two means:

- Bagging: each tree is built on a different training set using sampling with replacement from the original training set.
- To build each node of each tree, a subset of features is used. If there are $numFeatures$ available, $\sqrt{numFeatures}$ has been found a good size for feature subsampling.

The use of Random Forests for relational data mining purposes is not new: TILDE decision trees have been used as a basis for FORF (First-Order Relational Random Forests) [11], which can, as TILDE, be used with complex aggregates. However, the implementation suffers memory limitations, e.g. allocation failures when the feature space induced by the language bias is too wide. Also, the logic programming formalism makes the case of empty sets ambiguous. Indeed, the failure of a comparison test on an aggregate can have two reasons: the comparison can actually fail or the aggregate predicate can fail because it cannot compute a result, generally because the set to aggregate is empty. In the implementation of CARAF, we overcome this limitation by considering aggregation failure as a third outcome of a test.

Another relational Random Forest algorithm is described in [1]. It uses random rules based on the existential quantifier. However, it does not consider aggregates and the current implementation is limited to binary classification problems, which is inappropriate for most of the datasets we consider.

4 CARAF: Complex Aggregates with RAndom Forests

In this section, we describe the main contributions brought by CARAF (Complex Aggregates with RAndom Forests).

First is the use of random forests. The instance bagging part is performed the same way as Breiman does, by sampling with replacement from the training set. The feature sampling is different, based on the complex aggregates space structure. Let us denote by $AggProc = |(Function, Feature)|$ the number of aggregation processes, N_s the number of secondary objects, and A the number of attributes in the secondary table. The number of conjunctions of conditions, i.e. the number of possible $Selection$ grows like N_s^A for numeric attributes. A good estimation for the number of complex aggregates is then $ComplAgg = AggProc \cdot N_s^A$. As a subsampling method, we want to keep a search space of size $\sqrt{ComplAgg}$. We then keep $\sqrt{AggProc}$ aggregation processes and, in each process, $A/2$ attributes to put conditions on. This gives us the desired feature subsampling.

The RRHCCA algorithm aims at exploring the complex aggregates search space in a stochastic way. It uses random restart hill-climbing to find the best conjunction of conditions $Selection$ for given aggregation process $(Function, Feature)$. The hill-climbing process used to search this space can be RRHCCA, but we chose to simplify it to make it less time-consuming. We propose two approaches to achieve that. The aim of the first approach is the same as in RRHCCA, i.e. finding a suitable conjunction of conditions for a given aggregation process. We only use one process of hill-climbing, starting from an empty condition, which chooses one random move at each step from the same set of local moves as RRHCCA: by adding a random condition, removing one, or slightly modifying one. This approach is denoted by "Random". The difference is that RRHCCA tries every local move, while "Random" tries only one at random. We do that for each aggregation process and keep the best aggregate found over all aggregation processes. Pseudo-code for RRHCCA is recalled in Algorithm 1 and Algorithm 2. New pseudo-code for the "Random" approach is given in Algorithm 3 and Algorithm 4.

The second hill-climbing approach is based on loop inversion: instead of looking for the best condition for each aggregation process, we look for the best global condition. This is done by starting from an empty condition, and testing every aggregation process with this condition. The condition is then locally modified, still in the same way as in "Random", and all aggregation processes are tested with it, until no improvement is found. In this case, we keep the condition and the aggregation process that led to the best score with it. This approach is denoted by "Global" and pseudo-code is given in Algorithm 5. Algorithm 6 shows a useful function for enumerating local neighborhood of a conjunction of conditions. This hill-climbing algorithm has been designed for its efficiency in a SQL context. Indeed, we implemented CARAF to be compatible with SQL databases, with which it is more efficient to compute all aggregates for a given condition than all aggregates with different conditions on each one. This use of SQL tech-

nology allows CARAF to consider bigger datasets since it removes the need to store all data in memory.

Algorithm 1 Random Restart Hill-Climbing Algorithm (RRHCCA)

```
Input: functions: list of aggregation functions, features: list of attributes of the secondary table, train: labelled training set
Output: split: best complex aggregate found through hill-climbing
wheel ← InitializeBranches(functions, features)
bestSplits ← []
bestScore ← WORST_SCORE_FOR_METRIC
for i = 1 to MAX_ITERATIONS and wheel contains at least one branch do
    branch ← ChooseBranchToGrow(wheel)
    hasImproved ← branch.Grow(train)
    if not hasImproved then
        if branch.split.score ≥ bestScore then
            if branch.split.score > bestScore then
                bestScore ← branch.split.score
                bestSplits ← []
            end if
            bestSplits.Add(branch.split)
        end if
        branch.Reinitialize();
    end if
end for
split ← PickOneAtRandom(bestSplits)
return split
```

An additional feature is the use of ternary decision trees instead of binary decision trees. Each internal node of the tree has three sub-branches: one for success of the test, one for actual failure, and one for the unapplicability of the test, e.g. if the value of the feature involved in the test cannot be computed for the instance at hand. This is a way of dealing with empty sets in the context of complex aggregates. Indeed, imposing conditions on the secondary objects to aggregate can result in the absence of objects to be aggregated, i.e. aggregating an empty set. This is a problem for most aggregation functions, e.g. the average. We chose to tackle this issue by considering this as a third possible outcome of the test.

5 Experimental Results

In this section, we compare CARAF using the 3 different hill-climbing approaches to RELAGGS used in combination with Random Forest in Weka [7], and to FORF. All random forests were run to build 33 trees. We consider seven real-world real datasets.

Algorithm 2 Branch.Grow: Hill-Climbing Algorithm for One Branch

Input: *train*: labelled training set
Output: *hasImproved*: boolean indicating if the step of the hill-climbing has improved the best split found in the current hill-climbing of the branch

```
hasImproved ← false
allNeighbors ← EnumerateAggregateNeighbors(this.aggregate)
for all neighbor ∈ allNeighbors do
    aggregateToTry ← CreateAggregate(this.aggregate.function, this.aggregate.feature, neighbor)
    spl ← EvaluateAggregate(aggregateToTry, train)
    hasImproved ← hasImproved or UpdateBestSplit(spl)
end for
return hasImproved
```

Algorithm 3 Random Hill-Climbing Algorithm

Input: *functions*: list of aggregation functions, *features*: list of attributes of the secondary table, *train*: labelled training set
Output: *split*: best complex aggregate found through hill-climbing

```
wheel ← InitializeBranches(functions, features)
bestSplits ← []
bestScore ← WORST_SCORE_FOR_METRIC
for all aggProc ∈ wheel do
    iterWithoutImprovement ← 0
    for i = 1 to MAX_ITERATIONS and iterWithoutImprovement < 0.2*MAX_ITERATIONS do
        hasImproved ← branch.GrowRandom(train)
        if not hasImproved then
            iterWithoutImprovement++
            if branch.split.score ≥ bestScore then
                if branch.split.score > bestScore then
                    bestScore ← branch.split.score
                    bestSplits ← []
                end if
                bestSplits.Add(branch.split)
            end if
        else
            iterWithoutImprovement ← 0
        end if
    end for
end for
split ← bestSplits.oneRandomElement()
return split
```

Algorithm 4 Branch.GrowRandom: Hill-Climbing Algorithm for One Branch

Input: *train*: labelled training set
Output: *hasImproved*: boolean indicating if the step of the hill-climbing has improved the best split found in the current hill-climbing of the branch

---

allNeighbors ← EnumerateNeighbors(this.aggregate.condition)
neighbor ← allNeighbors.oneRandomElement()
aggregateToTry ← CreateAggregate(this.aggregate.function, this.aggregate.feature, neighbor)
spl ← EvaluateAggregate(aggregateToTry, train)
hasImproved ← UpdateBestSplit(spl)
return hasImproved

Algorithm 5 Global Hill-Climbing Algorithm

Input: *functions*: list of aggregation functions, *features*: list of attributes of the secondary table, *train*: labelled training set
Output: *split*: best complex aggregate found through hill-climbing

---

```
aggregationProcesses ← InitializeProcesses(functions, features)
bestSplits ← []
bestScore ← WORST_SCORE_FOR_METRIC
conjunction ← InitEmptyConjunction()
iterWithoutImprovement ← 0
for i = 1 to MAX_ITERATIONS and iterWithoutImprovement < 0.2*MAX_ITERATIONS do
    allNeighbors ← EnumerateNeighbors(conjunction)
    neighbor ← allNeighbors.oneRandomElement()
    hasImproved ← false
    for all aggProc ∈ aggregationProcesses do
        aggregateToTry ← CreateAggregate(aggProc.function, aggProc.feature, neighbor)
        spl ← EvaluateAggregate(aggregateToTry, train)
        if spl.score ≥ bestScore then
            if spl.score > bestScore then
                bestScore ← spl.score
                bestSplits ← []
                hasImproved ← true
            end if
            bestSplits.Add(spl)
        end if
    end for
    if hasImproved then
        iterWithoutImprovement ← 0
    else
        iterWithoutImprovement++
    end if
end for
split ← bestSplits.oneRandomElement()
return split
```

Algorithm 6 EnumerateNeighbors

```
1: Input: conjunction: aggregation conjunction of conditions
2: Output: allNeighbors: array of aggregation conjunctions, neighbors of conjunction
---
3: allNeighbors ← []
4: for all attr ∈ secondary attributes not present in conjunction do
5:    nextConjunction ← conjunction obtained by adding one randomly initialized
      condition on attr to conjunction
6:    allNeighbors.Add(nextConjunction)
7: end for
8: for all attr ∈ secondary attributes already present in conjunction do
9:    nextConjunction ← condition obtained by removing the condition on attr present
      in conjunction
10:   allNeighbors.Add(nextConjunction)
11: end for
12: for all attr ∈ secondary attributes already present in conjunction do
13:   for all move ∈ possible moves on the condition on attr present in conjunction
      do
14:      nextConjunction ← aggregate obtained by applying move to conjunction
15:      allNeighbors.Add(nextConjunction)
16:   end for
17: end for
18: return allNeighbors
```

- Auslan is a task of recognition of the Australian language sign.
- Diterpenes [6]
- Japanese vowels is related to recognition of Japanese vowels utterances from cepstrum analysis.
- Musk1 and Musk2 [5] are molecule classification tasks.
- Opt-digits deals with optical recognition of handwritten digits.
- Urban blocks [9] is a geographical classification task. This dataset is a clean version of the one used in [4] in the sense that duplicate urban blocks were removed.

A description of the datasets is given in Table 1.

The accuracy results are reported in Table 2. It is test set accuracy when a test set is available for the dataset or out-of-bag accuracy on the training set when there is no test set. The figures in bold indicate that the difference with RELAGGS is statistically significant with 95% confidence, while the underlined figures indicate a significant difference with FORF. The run of FORF on the Auslan dataset resulted in an unknown error and cannot be reported.

We observe that CARAF with the original RRHCCA hill-climbing algorithm is always performing better than both RELAGGS and FORF, the difference being significant in 3 cases out of 7 over RELAGGS, and 4 out of 6 over FORF. The Random and Global hill-climbing approaches also perform better than RELAGGS and FORF in a majority of cases, some cases also being statistically significant. These two approaches, considering less complex aggregates, also have

Table 1. Characteristics of the datasets used in the experimental comparison.

Dataset	Instances	Classes
Auslan	2565	96
Diterpenes	1503	23
Japanese vowels	270+370	9
Musk1	92	2
Musk2	102	2
Opt-digits	3823+1797	10
Urban blocks	591	6

Table 2. Results of CARAF with different hill-climbing heuristics on different datasets (out-of-bag accuracy or test set accuracy)

Dataset	RELAGGS	FORF	RRHCCA	Random	Global
Auslan	94.19%	ERR	**96.53%**	**95.91%**	94.66%
Diterpenes	89.09%	90.49%	**92.95%**	85.06%	**93.35%**
Japanese vowels	93.78%	94.86%	95.41%	**97.30%**	**97.03%**
Musk1	80.43%	78.26%	89.13%	84.78%	80.43%
Musk2	76.47%	75.49%	81.37%	85.29%	82.35%
Opt-digits	22.37%	76.57%	**95.94%**	**94.60%**	**92.77%**
Urban blocks	83.42%	75.81%	84.94%	83.76%	84.60%
			7 **(3)** - 6 (4)	6 **(3)** - 5 (2)	6.5 **(3)** - 6 (3)

the advantage of speed over RRHCCA. As shown in Table 3, the runtimes of both Random and Global are lower by a factor at least 4 than the runtimes of RRHCCA, Global being faster than Random. The loss in accuracy performance is tiny: RRHCCA outperforms Random 5 times, the difference being statistically significant only once. RRHCCA outperforms Global 4 times, significantly twice. The Random and Global approaches are then good performers too. Therefore, our recommendation is, if runtime is not a problem for the dataset at hand, to use RRHCCA. If time is critical, then Random is the best option, followed by Global.

Table 3. Runtime of the algorithms (in minutes)

Dataset	RRHCCA	Random	Global
Auslan	921	250	146
Diterpenes	4	1	1
Japanese vowels	13	1	1
Musk1	98	8	5
Musk2	733	71	55
Opt-digits	35	9	5
Urban blocks	4	1	1

6 Conclusion and Future Work

In this paper, we presented CARAF, a relational random forest learner based on complex aggregates. The hill-climbing algorithms to explore the search space perform better than RELAGGS with Random Forests and FORF on most datasets. The basic random hill-climbing algorithms to explore the complex aggregates search space yield a considerable speed up while not suffering performance loss.

Future work will consist in exploring database technologies that are suitable for learning from relational data. Indeed, most relational algorithms have not been designed to handle big data, and there is an increasing trend towards relevant representation of relational data and the technologies, potentially NoSQL-based, fitted for relational data mining.

References

1. Anderson, G., Pfahringer, B.: Relational random forests based on random relational rules. In: Boutilier, C. (ed.) IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009. pp. 986–991 (2009), http://ijcai.org/papers09/Papers/IJCAI09-167.pdf
2. Blockeel, H., Raedt, L.D.: Top-down induction of first-order logical decision trees. Artif. Intell. 101(1-2), 285–297 (1998)
3. Breiman, L.: Random forests. Machine Learning 45(1), 5–32 (2001), http://dx.doi.org/10.1023/A:1010933404324
4. Charnay, C., Lachiche, N., Braud, A.: Construction of complex aggregates with random restart hill-climbing. In: 24th International Conference on Inductive Logic Programming (ILP'14) (2014)
5. Dietterich, T.G., Lathrop, R.H., Lozano-Pérez, T.: Solving the multiple instance problem with axis-parallel rectangles. Artif. Intell. 89(1-2), 31–71 (1997), http://dx.doi.org/10.1016/S0004-3702(96)00034-3
6. Dzeroski, S., Schulze-Kremer, S., Heidtke, K.R., Siems, K., Wettschereck, D., Blockeel, H.: Diterpene structure elucidation from 13cnmr spectra with inductive logic programming. Applied Artificial Intelligence 12(5), 363–383 (1998), http://dx.doi.org/10.1080/088395198117686
7. Hall, M.A., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. SIGKDD Explorations 11(1), 10–18 (2009), http://doi.acm.org/10.1145/1656274.1656278
8. Krogel, M.A., Wrobel, S.: Facets of aggregation approaches to propositionalization. In: Horvath, T., Yamamoto, A. (eds.) Work-in-Progress Track at the Thirteenth International Conference on Inductive Logic Programming (ILP) (2003)
9. Puissant, A., Lachiche, N., Skupinski, G., Braud, A., Perret, J., Mas, A.: Classification et évolution des tissus urbains à partir de données vectorielles. Revue Internationale de Géomatique 21(4), 513–532 (2011)
10. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann (1993)
11. Van Assche, A., Vens, C., Blockeel, H., Dzeroski, S.: First order random forests: Learning relational classifiers with complex aggregates. Machine Learning 64(1-3), 149–182 (2006)
12. Vens, C., Ramon, J., Blockeel, H.: Refining aggregate conditions in relational learning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) PKDD. Lecture Notes in Computer Science, vol. 4213, pp. 383–394. Springer (2006)