

CARAF: Complex Aggregates within Random Forests

Clément Charnay^(✉), Nicolas Lachiche, and Agnès Braud

ICube, Université de Strasbourg, CNRS, 300 Bd Sébastien Brant - CS 10413,
67412 Illkirch Cedex, France
{[@unistra.fr](mailto:charnay,nicolas.lachiche,agnes.braud)}

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. CARAF allows to perform complex aggregate feature selection.

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 perform prediction on. This prediction, supervised learning, task is either a classification task if the attribute to predict is categorical, i.e. if it takes a finite number of values, or a regression task, if the attribute to predict is numeric. 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 Fig. 1. It is a classification task, where the aim

is to predict the sign associated to a record of hand motion. 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 samples of the records, with a timestamp attribute and 22 attributes representing values from the channels that monitor the hand motion through a glove.

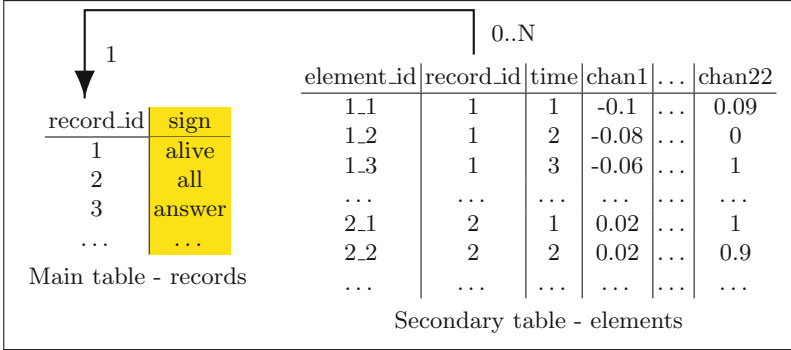


Fig. 1. Schema of the real-world Auslan dataset.

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 [11] 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 numeric 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 [9] 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.

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, when a model introduces complex aggregates, it means they have been found to increase the performance on the training set over simple aggregates. If on test data, the complex aggregate-based model performs worse than a simple aggregate-based one, there is overfitting.

In this paper, we propose the extension of the decision tree learner based on RRHCCA to a random forest learner, introducing two faster hill-climbing algorithms. The method to perform complex aggregate feature selection in order to identify relevant families of aggregates is also presented.

The rest of the paper is organized as follows: in Sect. 2 we briefly define the concept of complex aggregates. In Sect. 3, we review the use of random forests in the relational setting. In Sect. 4, we introduce CARAF (Complex Aggregates with RRandom Forests), a new relational random forest learner implementing our contributions. In Sect. 5, we present experimental results obtained with CARAF. In Sect. 6, we present how to perform complex aggregate feature selection with CARAF. Finally, in Sect. 7, 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 [5].

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 numeric 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. Aggregation functions we consider are *count* which does not need an attribute to aggregate, *min*, *max*, *sum*, *average*, *standard deviation*, *median*, *first quartile*, *third quartile*, *interquartile range*, *first decile* and *ninth decile* for numeric attributes, *proportion of secondary objects with*

attribute value for categorical attributes, the latter is defined as the ratio of secondary objects linked to a given main object with a given value for the attribute, over the count of all secondary objects linked this main object.

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 [14] 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 approach introduced in [3] builds complex aggregate features for use in a Bayesian classifier, guided by minimum description length and a heuristic sampling. The RRHCCA algorithm [5] 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 [4] is an ensemble classification technique which builds a set of diverse decision trees and combines their predictions into a single output. Considered individually, each decision tree is less accurate than a decision tree built in a classic way. However, the introduction of diversity through the forest improves the performance over a single decision tree, by solving the overfitting problem induced by the latter approach. Algorithm 1 shows the building process of a Random Forest. Diversity between the trees is achieved by two means:

- Bootstrapping: each tree is built on a different training set using sampling with replacement from the original training set. In other words, each decision tree is built using a training set with same size as the original one, but where repetitions may occur. This corresponds to lines 5 to 8 of Algorithm 1.
- Feature sampling: to build each node of each tree, a subset of features is used. If there are *numFeatures* available, $\sqrt{\text{numFeatures}}$ are considered for introduction in node split. This corresponds to lines 9 to 15 of Algorithm 1.

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) [13], which can, as TILDE, be used with complex aggregates. However, the implementation suffers memory limitations, e.g. allocation failures

Algorithm 1. BuildRandomForest

```

1: Input: train: set of training examples, feats: set of possible split features, target:
   the target attribute, n: number of trees in the forest.
2: Output: forest: a random forest.

```

```

3: forest  $\leftarrow$  InitEmptyForest()
4: for k = 1 to n do
5:   trainForTree  $\leftarrow$  InitEmptyInstances()
6:   for i = 1 to train.Size() do
7:     trainForTree.Add(train.OneRandomElement())
8:   end for
9:   featsCopy  $\leftarrow$  feats.Copy()
10:  featsForTree  $\leftarrow$  InitEmptyFeatures()
11:  for j = 1 to  $\sqrt{\text{feats.Size()}}$  do
12:    f  $\leftarrow$  featsCopy.OneRandomElement()
13:    featsCopy.Remove(f)
14:    featsForTree.Add(f)
15:  end for
16:  tree  $\leftarrow$  BuildDecisionTree(trainForTree, featsForTree, target)
17:  forest.Add(tree)
18: end for
19: return forest

```

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.

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 bootstrapping 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 $\text{AggProc} = |(Function, Feature)|$ 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.

For instance, let us consider again the Auslan dataset. For sake of simplicity, we consider *count*, *minimum*, *maximum* and *average* as the possible aggregation function, and attributes time and channels 1 to 4. Table 1 shows an example of complex aggregates subsampling on this dataset. Out of the 16 aggregation processes available, the square root will be considered at each node, i.e. 4, as shown in Table 1a. For each aggregation process, half of the 5 secondary attributes will be kept for use in the selection conjunction of conditions, i.e. 3 per aggregation process, as shown in Table 1b.

Table 1. Subsampling of complex aggregates.

(a) Subsampling of aggregation processes.

Function	Attribute	Chosen
Count		x
Minimum	Time	
Minimum	Chan1	
Minimum	Chan2	
Minimum	Chan3	
Minimum	Chan4	x
Maximum	Time	
Maximum	Chan1	
Maximum	Chan2	
Maximum	Chan3	
Maximum	Chan4	
Average	Time	x
Average	Chan1	
Average	Chan2	
Average	Chan3	
Average	Chan4	x

(b) Subsampling of secondary attributes.

Attribute	Chosen
Time	x
Chan1	x
Chan2	x
Chan3	
Chan4	

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 a 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.

We first introduce the “Random” hill-climbing algorithm, for which pseudo-code is given in Algorithm 2. Like RRHCCA, the aim is to look for an appropriate conjunction of basic conditions for a fixed aggregation process. But instead of considering all neighbors of an aggregate at each step of hill-climbing, the Random algorithm will consider only one, randomly chosen, neighbor, for split evaluation.

Algorithm 2. Random Hill-Climbing Algorithm

```

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

3: aggregationProcesses  $\leftarrow$  InitializeProcesses(functions, features)
4: bestSplits  $\leftarrow$  []
5: bestScore  $\leftarrow$  WORST_SCORE_FOR_METRIC
6: for all aggProc  $\in$  aggregationProcesses do
7:   iterWithoutImprovement  $\leftarrow$  0
8:   for i = 1 to MAX_ITERATIONS and iterWithoutImprovement <
       0.2*MAX_ITERATIONS do
9:     hasImproved  $\leftarrow$  aggProc.GrowRandom(train)
10:    if not hasImproved then
11:      iterWithoutImprovement++
12:      if aggProc.split.score  $\geq$  bestScore then
13:        if aggProc.split.score > bestScore then
14:          bestScore  $\leftarrow$  aggProc.split.score
15:          bestSplits  $\leftarrow$  []
16:        end if
17:        bestSplits.Add(aggProc.split)
18:      end if
19:    else
20:      iterWithoutImprovement  $\leftarrow$  0
21:    end if
22:  end for
23: end for
24: split  $\leftarrow$  bestSplits.OneRandomElement()
25: return split

```

Algorithm 3. AggregationProcess.GrowRandom: Perform One Step of Hill-Climbing for the Aggregation Process

```

1: Input: train: labelled training set.
2: Output: hasImproved: boolean indicating if the step of the hill-climbing has
   improved the best split found in the current hill-climbing of the aggregation process.

3: allNeighbors  $\leftarrow$  EnumerateNeighbors(this.aggregate.condition)
4: neighbor  $\leftarrow$  allNeighbors.OneRandomElement()
5: aggregateToTry  $\leftarrow$  CreateAggregate(this.aggregate.function,
   this.aggregate.feature, neighbor)
6: spl  $\leftarrow$  EvaluateAggregate(aggregateToTry, train)
7: hasImproved  $\leftarrow$  UpdateBestSplit(spl)
8: return hasImproved

```

Algorithm 4. Global Hill-Climbing Algorithm

```

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

```

```

3: aggregationProcesses  $\leftarrow$  InitializeProcesses(functions, features)
4: bestSplits  $\leftarrow$  []
5: bestScore  $\leftarrow$  WORST_SCORE_FOR_METRIC
6: conjunction  $\leftarrow$  InitEmptyConjunction()
7: iterWithoutImprovement  $\leftarrow$  0
8: for i = 1 to MAX_ITERATIONS and iterWithoutImprovement <
   0.2*MAX_ITERATIONS do
9:   allNeighbors  $\leftarrow$  EnumerateNeighbors(conjunction)
10:  neighbor  $\leftarrow$  allNeighbors.oneRandomElement()
11:  hasImproved  $\leftarrow$  false
12:  for all aggProc  $\in$  aggregationProcesses do
13:    aggregateToTry  $\leftarrow$  CreateAggregate(aggProc.function, aggProc.feature,
      neighbor)
14:    spl  $\leftarrow$  EvaluateAggregate(aggregateToTry, train)
15:    if spl.score  $\geq$  bestScore then
16:      if spl.score > bestScore then
17:        bestScore  $\leftarrow$  spl.score
18:        bestSplits  $\leftarrow$  []
19:        hasImproved  $\leftarrow$  true
20:      end if
21:      bestSplits.Add(spl)
22:    end if
23:  end for
24:  if hasImproved then
25:    iterWithoutImprovement  $\leftarrow$  0
26:  else
27:    iterWithoutImprovement++
28:  end if
29: end for
30: split  $\leftarrow$  bestSplits.OneRandomElement()
31: return split

```

This corresponds to the function `GrowRandom` shown in Algorithm 3. If the chosen neighbor improves over the original aggregate, the search resumes from this neighbor. The neighbors are defined as in Algorithm 5: from a current aggregate, they are obtained by adding a random basic condition to the conjunction, removing a condition from the conjunction, and modifying one.

The hill-climbing has two possible stopping criteria: when a maximum number of hill-climbing steps have been performed, or when a certain number of neighbors of a given aggregate have been considered without improvement, this number has been arbitrarily fixed to 20% of the maximum number of hill-climbing steps. In other words, if 20% of the maximum number of iterations

Algorithm 5. EnumerateNeighbors

```

1: Input: conjunction: aggregation conjunction of conditions.
2: Output: allNeighbors: array of aggregation conjunctions, neighbors of conjunction.


---


3: allNeighbors  $\leftarrow \emptyset$ 
4: for all attr  $\in$  secondary attributes not present in conjunction do
5:   nextConjunction  $\leftarrow$  conjunction obtained by adding one randomly initialized
     condition on attr to conjunction
6:   allNeighbors.Add(nextConjunction)
7: end for
8: for all attr  $\in$  secondary attributes already present in conjunction do
9:   nextConjunction  $\leftarrow$  condition obtained by removing the condition on attr present
     in conjunction
10:  allNeighbors.Add(nextConjunction)
11: end for
12: for all attr  $\in$  secondary attributes already present in conjunction do
13:   for all move  $\in$  possible moves on the condition on attr present in conjunction
     do
14:     nextConjunction  $\leftarrow$  aggregate obtained by applying move to conjunction
15:     allNeighbors.Add(nextConjunction)
16:   end for
17: end for
18: return allNeighbors

```

have passed with no improvement, the search stops. This aggregation process-wise hill-climbing loop corresponds to lines 8 through 22.

This hill-climbing search is then performed once for each aggregation process available, starting from an empty conjunction of conditions, without a restart. This corresponds to the loop from line 6 to line 23.

Following the idea of the Random hill-climbing algorithm, we propose to invert the loops of hill-climbing and aggregation process, materialized in the “Global” hill-climbing algorithm. In practice, only one hill-climbing search is performed, which aims at finding the best conjunction of conditions for all aggregation processes available. For a given conjunction of conditions, all aggregation processes are used to form aggregates and splitting conditions, and the conjunction is evaluated according to the best score achieved over all aggregation processes. The pseudo-code is given in Algorithm 4. This time, the aggregation process loop (from line 12 to line 23) is enclosed in the hill-climbing loop (from line 8 to line 29).

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 choose 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 [8], and to FORF. All random forests were run to build 33 trees. We consider seven real-world real datasets.

- Auslan is a task of recognition of the Australian language sign.
- Diterpenes [7] is a molecule classification task.
- Japanese vowels is related to recognition of Japanese vowels utterances from cepstrum analysis.
- Musk1 and Musk2 [6] are molecule classification tasks.
- Mutagenesis [12] is about predicting mutagenicity of a molecule with respect to the properties of its atoms. In our two-table setting, we use the so-called “regression-friendly” subset of the dataset, and consider a molecule as a bag of atoms, i.e. we do not consider the bond information between atoms.
- Opt-digits deals with optical recognition of handwritten digits.
- Urban blocks [10] is a geographical classification task. This dataset is a clean version of the one used in [5] in the sense that duplicate urban blocks were removed.

A description of the datasets is given in Table 2.

The accuracy results are reported in Table 3. 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. Out-of-bag error is defined as follows: as mentioned previously, each tree in a Random Forest is trained using a subsample of the original training set, i.e. for each tree, there is a fraction of the training set that has not been actually used to build the tree. The out-of-bag accuracy for the tree is the error made by the tree on this set of unseen examples, called the out-of-bag examples. Any error metric can be used. For classification tasks, error rate will be most likely used, while for regression tasks root mean squared error could be used. By extension, out-of-bag accuracy is defined as the complementary to 1 of the out-of-bag classification error rate. 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 8 over RELAGGS, and 4 out of 7 over FORF. The Random and Global hill-climbing approaches also perform better

Table 2. Characteristics of the datasets used in the experimental comparison.

Dataset	Instances	Classes	Secondary objects	Secondary attributes
Auslan	2 565	96	146 949	23
Diterpenes	1 503	23	30 060	2
Japanese vowels	270 + 370	9	9 961	12
Musk1	92	2	476	166
Musk2	102	2	6 598	166
Mutagenesis	188	2	4 893	3
Opt-digits	3 823 + 1 797	10	5 754 880	3
Urban blocks	591	6	7 692	3

Table 3. 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 %	<u>89.13 %</u>	84.78 %	80.43 %
Musk2	76.47 %	75.49 %	81.37 %	85.29 %	82.35 %
Mutagenesis	88.30 %	87.77 %	90.43 %	91.49 %	92.02 %
Opt-digits	22.37 %	76.57 %	95.94 %	94.60 %	92.77 %
Urban blocks	83.42 %	75.81 %	<u>84.94 %</u>	<u>83.76 %</u>	<u>84.60 %</u>
			8 (3) - 7 <u>(4)</u>	7 (3) - 6 <u>(2)</u>	7.5 (3) - 7 <u>(3)</u>

than RELAGGS and FORF in a majority of cases, some cases also being statistically significant. These two approaches, considering less complex aggregates, also have the advantage of speed over RRHCCA. As shown in Table 4, 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 4. 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
Mutagenesis	6	2	2
Opt-digits	35	9	5
Urban blocks	4	1	1

6 Aggregation Processes Selection with Random Forests

Random Forests can be used to perform feature selection, as introduced by Breiman in [4]. The aim is to first check which families of complex aggregates are the most promising, to learn a model afterwards using only these useful families.

Our goal is to perform feature selection, i.e. to assess the importance of an input feature for prediction of the output attribute. This achieved using permutation tests. For a given tree, we first measure the out-of-bag error. The second step is to permute among the out-of-bag examples the value for the input feature we want to measure the importance. This gives a new out-of-bag examples set, for which we compute an after-permutation out-of-bag error. The importance of the feature at the tree-level is the increase in error between the after-permutation out-of-bag set and the original out-of-bag set. The final feature importance is then obtained by averaging tree-level feature importances over the whole forest.

In a relational context where complex aggregates are being used, this method needs adaptation. Indeed, the size of the complex aggregates search space implies that a given complex aggregate is rarely used twice in the same model. However, the structure of the complex aggregates allows us to define families of complex aggregates, and to measure importance of the families rather than specific complex aggregates.

Families of complex aggregates can be defined according to two elements:

- Aggregation processes: Complex aggregates sharing a common aggregation process will belong to the same family.
- Attributes in selection conjunctions: Complex aggregates whose selection conjunctions of conditions have a condition on a common attribute will belong to the same family.

These two elements can be combined to define more specific attributes, e.g. complex aggregates with the same aggregation process whose conjunctions of conditions have a condition on the same given attribute.

As an example, we use the urban blocks dataset from Sect. 5. We consider *count*, *minimum*, *maximum* and *average* as the possible aggregation functions. Block-wise features are area, elongation, convexity and density, while building-wise features are area, elongation and convexity.

If we define families of complex aggregates at the aggregation process level, we obtain as many families as aggregation processes, 10 in this example. Thus, following aggregates will fall into the same family, since they are all based on the same aggregation process, the average area of buildings:

- *average*(*area*, *buildings*, *true*)
- *average*(*area*, *buildings*, *elongation* ≥ 0.7)
- *average*(*area*, *buildings*, *convexity* < 0.5)

If we define families based on one common attribute in the conjunction of conditions, we have as many families as attributes in the secondary table, 3 in this example. Thus, following aggregates will fall into the same family, since their conjunctions of conditions all have a condition on elongation of buildings:

- *average*(*area*, *buildings*, *elongation* ≥ 0.7)
- *maximum*(*convexity*, *buildings*, *elongation* < 0.6)
- *count*(*buildings*, *elongation* $< 0.8 \wedge$ *area* ≥ 100)

Both can be combined to create families based on the aggregation process and a common attribute in conjunction of conditions, 30 in this example. For instance, following aggregates will belong to the same family, sharing both the aggregation process of average area of buildings and a condition on elongation of buildings:

- *average*(*area*, *buildings*, *elongation* ≥ 0.7)
- *average*(*area*, *buildings*, *elongation* $< 0.9 \wedge$ *convexity* ≥ 0.7)
- *average*(*area*, *buildings*, *elongation* $\geq 0.5 \wedge$ *area* < 100)

The permutation of values of complex aggregates has then to be performed. Since we are not permuting the values of a single feature, but of a whole family, we have to keep some coherence: each training example has one value for each aggregate in the family, and they should not be separated by the permutation. An example that obtains the value of a second example for a first aggregate, should not obtain the value of a third example for a second aggregate, but rather the value of the second example. In other words, for a given family of aggregates, only one permutation of examples has to be found, since a set of aggregate values for a given example should be conserved through permutation. We achieve this by permuting groups of secondary objects, i.e. the set of secondary objects related to one example will be assigned to another example. By doing this, all aggregate values are transferred from one example to another.

The family importance is then computed as described above: for each tree we obtain the error gain between before and after the permutation, and the gain average over all trees gives the final importance.

Table 5. Importance of main features and aggregation processes in urban blocks.

Feature	Score
Area	0.039
Elongation	0.003
Convexity	0.005
Density	0.157
Count	0.027
Minimum Area	0.062
Minimum Elongation	0.034
Minimum Convexity	0.028
Maximum Area	0.111
Maximum Elongation	0.054
Maximum Convexity	0.038
Average Area	0.177
Average Elongation	0.061
Average Convexity	0.039

As an example, the importances of blocks main features and buildings aggregation processes are reported in Table 5. Importances were obtained using a forest of 100 trees built using the “Random” hill-climbing heuristic to find complex aggregates.

We observe that the 3 most important features for urban blocks classification are the average area of buildings, the density of blocks, and the maximum area of buildings.

7 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. Boullé, M.: Towards automatic feature construction for supervised classification. In: Calders, T., Esposito, F., Hüllermeier, E., Meo, R. (eds.) ECML PKDD 2014, Part I. LNCS, vol. 8724, pp. 181–196. Springer, Heidelberg (2014). http://dx.doi.org/10.1007/978-3-662-44848-9_12
4. Breiman, L.: Random forests. *Mach. Learn.* **45**(1), 5–32 (2001). <http://dx.doi.org/10.1023/A:1010933404324>
5. Charnay, C., Lachiche, N., Braud, A.: Construction of complex aggregates with random restart hill-climbing. In: Davis, J., et al. (eds.) ILP 2014. LNCS, vol. 9046, pp. 49–61. Springer, Heidelberg (2015). doi:[10.1007/978-3-319-23708-4_4](https://doi.org/10.1007/978-3-319-23708-4_4)
6. 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](http://dx.doi.org/10.1016/S0004-3702(96)00034-3)
7. Dzeroski, S., Schulze-Kremer, S., Heidtke, K.R., Siems, K., Wettschereck, D., Blockeel, H.: Diterpene structure elucidation from ¹³CNMR spectra with inductive logic programming. *Appl. Artif. Intell.* **12**(5), 363–383 (1998). <http://dx.doi.org/10.1080/088395198117686>
8. Hall, M.A., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. *SIGKDD Explor.* **11**(1), 10–18 (2009). <http://doi.acm.org/10.1145/1656274.1656278>
9. 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)
10. Puissant, A., Lachiche, N., Skupinski, G., Braud, A., Perret, J., Mas, A.: Classification et évolution des tissus urbains à partir de données vectorielles. *Rev. Int. de Géomatique* **21**(4), 513–532 (2011)
11. Quinlan, J.R.: C4.5: Programs for Machine Learning. Morgan Kaufmann, San Francisco (1993)
12. Srinivasan, A., Muggleton, S., Sternberg, M.J.E., King, R.D.: Theories for mutagenicity: A study in first-order and feature-based induction. *Artif. Intell.* **85**(1–2), 277–299 (1996). [http://dx.doi.org/10.1016/0004-3702\(95\)00122-0](http://dx.doi.org/10.1016/0004-3702(95)00122-0)
13. Van Assche, A., Vens, C., Blockeel, H., Dzeroski, S.: First order random forests: Learning relational classifiers with complex aggregates. *Mach. Learn.* **64**(1–3), 149–182 (2006)
14. Vens, C., Ramon, J., Blockeel, H.: Refining aggregate conditions in relational learning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) PKDD 2006. LNCS (LNAI), vol. 4213, pp. 383–394. Springer, Heidelberg (2006)

Inductive Logic Programming

25th International Conference, ILP 2015, Kyoto, Japan,

August 20-22, 2015, Revised Selected Papers

Inoue, K.; Ohwada, H.; Yamamoto, A. (Eds.)

2016, X, 215 p. 56 illus., Softcover

ISBN: 978-3-319-40565-0