CLUS: User's Manual

Jan Struyf, Bernard Ženko, Hendrik Blockeel, Celine Vens,
Matej Petković, Tomaž Stepišnik Perdih, Vanja Mileski,
Martin Breskvar, Jurica Levatić, Dragi Kocev, Sašo Džeroski

August 30, 2022

Contents

Chapter 1

Introduction

This text is a user's manual for the open source machine learning system Clus. Clus is a decision tree and rule learning system that works in the *predictive clustering* framework [3]. While most decision tree learners induce classification or regression trees, Clus generalizes this approach by learning trees that are interpreted as cluster hierarchies. We call such trees predictive clustering trees or PCTs. Depending on the learning task at hand, different goal criteria are to be optimized while creating the clusters, and different heuristics will be suitable to achieve this.

Classification and regression trees are special cases of PCTs, and by choosing the right parameter settings Clus can closely mimic the behavior of tree learners such as CART [5] or C4.5 [23]. However, its applicability goes well beyond classical classification or regression tasks: Clus has been successfully applied to many different tasks including multi-task learning (multi-target classification and regression), structured output learning, multi-label classification, hierarchical classification, hierarchical regression, and time series prediction. Next to these supervised learning tasks, PCTs are also applicable to semi-supervised learning, subgroup discovery, and clustering. In a similar way, predictive clustering rules (PCRs) generalize classification rule sets [8] and also apply to the aforementioned learning tasks.

A full description of how Clus works is beyond the scope of this text. In this User's Manual, we focus on how to use Clus: how to prepare its inputs, how to interpret the outputs, and how to change its behavior with the available parameters. This manual is a work in progress and all comments are welcome. For background information on the rationale behind the Clus system and its algorithms we refer the reader to the following papers:

- H. Blockeel, L. De Raedt, and J. Ramon. Top-down induction of clustering trees. In *Proceedings of the 15th International Conference on Machine Learning*, pages 55–63, 1998.
- H. Blockeel and J. Struyf. Efficient algorithms for decision tree cross-validation. *Journal of Machine Learning Research*, 3: 621–650, December 2002.
- H. Blockeel, S. Džeroski, and J. Grbović, Simultaneous prediction of multiple chemical parameters of river water quality with TILDE, Proceedings of the Third European Conference on Principles of Data Mining and Knowledge Discovery (J.M. Żytkow and J. Rauch, eds.), vol 1704, LNAI, pp. 32-40, 1999.
- T. Aho, B. Ženko, and S. Džeroski. Rule ensembles for multi-target regression. In *Proceedings of 9th IEEE International Conference on Data Mining (ICDM 2009)*, pages 21–30, 2009.
- E. Fromont, H. Blockeel, and J. Struyf. Integrating decision tree learning into inductive databases. *Lecture Notes in Computer Science*, 4747: 81–96, 2007.
- D. Kocev, C. Vens, J. Struyf, and S. Džeroski. Tree ensembles for predicting structured outputs. *Pattern Recognition*, 46 (3): 817–833, 2013.
- I. Slavkov, V. Gjorgjioski, J. Struyf, and S. Džeroski. Finding explained groups of time-course gene expression profiles with predictive clustering trees. *Molecular Biosystems*, 2009. To appear.
- J. Struyf and S. Džeroski. Clustering trees with instance level constraints. *Lecture Notes in Computer Science*, 4701: 359–370, 2007.
- J. Struyf and S. Džeroski. Constraint based induction of multi-objective regression trees. *Lecture Notes in Computer Science*, 3933: 110–121, 2005.

- C. Vens, J. Struyf, L. Schietgat, S. Džeroski, and H. Blockeel. Decision trees for hierarchical multi-label classification. *Machine Learning*, 73 (2): 185–214, 2008.
- B. Ženko and S. Džeroski. Learning classification rules for multiple target attributes. In *Advances in Knowledge Discovery and Data Mining*, pages 454–465, 2008.
- J. Levatić, D. Kocev, M. Ceci, and S. Džeroski. Semi-supervised trees for multi-target regression. In *Information Sciences*, 450:109-127, 2018.
- M. Breskvar, D. Kocev, and S. Džeroski. Ensembles for multi-target regression with random output selections. In *Machine Learning*, 107 (11): 1673–1709, 2018.
- M. Petković, D. Kocev, and S. Džeroski. Feature ranking for multi-target regression. In *Machine Learning*, 109 (6): 1179–1204, 2020.
- M. Petković, D. Kocev, and S. Džeroski. Multi-label feature ranking with ensemble methods. In *Machine Learning*, 109 (6): 2141–2159, 2020.

Chapter 2

Getting Started

2.1 Installing and Running Clus

Clus is written in the Java programming language, which is available from `http://www.oracle.com`. You will need Java version 1.8.x or newer and Apache Maven 3.3.9 or newer. To run Clus, it suffices to install the Java Runtime Environment (JRE). If you want to make changes to Clus and compile its source code, then you will need to install the Java Development Kit (JDK) instead of the JRE.

The Clus software is released under the GNU General Public License version 3 or later and is available for download at `http://source.ijs.si/ktclus/clusproject.git`. After downloading Clus, unpack it into a directory of your choice. Clus is a command line application and should be started from the command prompt (Windows) or a terminal window (Unix). To build Clus, enter the following commands:

```
git clone http://source.ijs.si/ktclus/clusproject.git
cd ClusProject
mvn clean package
```

Clus executable is located in the `ClusProject/target` folder. In order to verify that your copy of Clus is working properly, you can run it without parameters like so:

```
java -jar Clus-<version>.jar
```

You should receive an output similar to the one below:

```
Clus <VERSION> - Software for Predictive Clustering
Copyright (C) 2007 - 2022
Katholieke Universiteit Leuven, Leuven, Belgium
Jozef Stefan Institute, Ljubljana, Slovenia

This program is free software and comes with ABSOLUTELY NO
WARRANTY. You are welcome to redistribute it under certain
conditions. Type 'clus -copying' for distribution details.

Usage: clus appname
Database: appname.arff
Settings: appname.s
Output:   appname.out
Expected main argument
```

2.2 Input and Output Files for Clus

Clus uses (at least) two input files and these are named *filename*`.s` and *filename*`.arff`, with *filename* a name chosen by the user. The file *filename*`.s` contains the parameter settings for Clus. The file *filename*`.arff` contains the training data to be read. The format of the data file is Weka's ARFF format[1].

[1] `http://weka.wikispaces.com/ARFF`

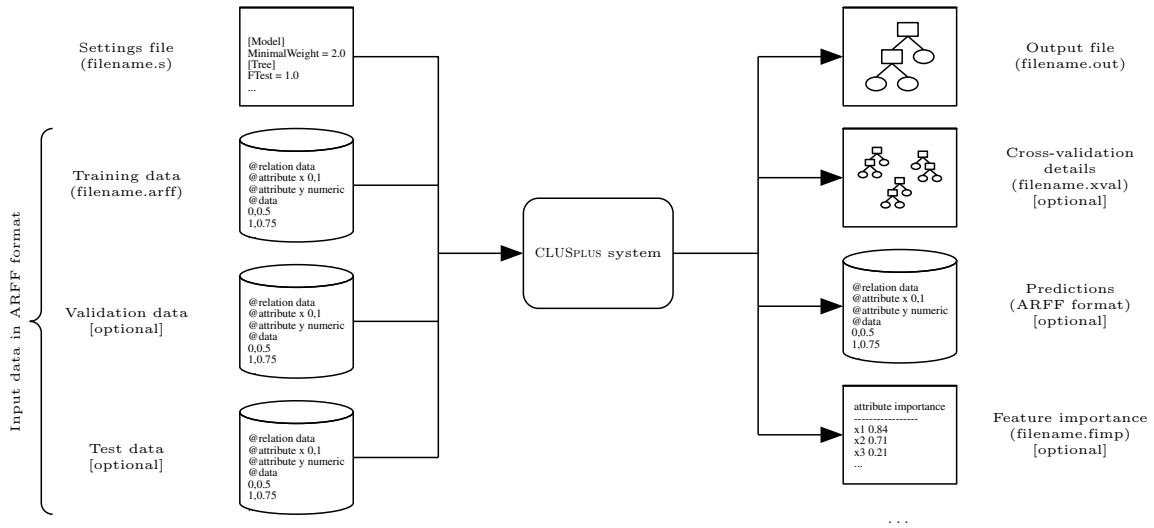

Figure 2.1: Input and output files of CLUS.

```
[Attributes]
Descriptive = 1-2
Target = 3-4
Clustering = 3-4

[Tree]
Heuristic = VarianceReduction
```

Figure 2.2: The settings file (`weather.s`) for the *Weather* example.

```
@RELATION "weather"

@ATTRIBUTE outlook     {sunny,rainy,overcast}
@ATTRIBUTE windy       {yes,no}
@ATTRIBUTE temperature numeric
@ATTRIBUTE humidity    numeric

@DATA
sunny,    no,  34, 50
sunny,    no,  30, 55
overcast, no,  20, 70
overcast, yes, 11, 75
rainy,    no,  20, 88
rainy,    no,  18, 95
rainy,    yes, 10, 95
rainy,    yes, 8,  90
```

Figure 2.3: The training data (`weather.arff`) for the *Weather* example (in Weka's ARFF format).

The results of a CLUS run are put in an output file *filename*`.out`. Figure 2.1 gives an overview of the input and output files supported by CLUS. The format of the data files is described in detail in Chapter 3, the format of the settings file is discussed in Chapter 4, and the output files are covered in Chapter 6. Optionally, CLUS can also generate a detailed output of the cross-validation (`weather.xval`) and model predictions in ARFF format.

2.3 A Step-by-step Example

The CLUS distribution includes a number of example datasets. In this section we briefly take a look at the *Weather* dataset, and how it can be processed by CLUS. We use Unix notation for paths to filenames; in Windows notation the slashes become backslashes (see also previous section).

1. Move to the directory `Clus/data/weather`, which contains the *Weather* dataset:

   ```
   cd Clus/data/weather
   ```

2. First inspect the file `weather.arff`. Its contents is also shown in Figure 2.3. This file contains the input data that CLUS will learn from. It is in the ARFF format: first, the name of the table is given; then, the attributes and their domains are listed; finally, the table itself is listed.

3. Next, inspect the file `weather.s`. This file is also shown in Figure 2.2. It is the *settings* file, the file where CLUS will find information about the task it should perform, values for its parameters, and other information that guides it behavior.

   The *Weather* example is a small multi-target or multi-task learning problem [6], in which the goal is to predict the target attributes *temperature* and *humidity* from the input attributes *outlook* and *windy*. This kind of information is what goes in the settings file. The parameters under the heading `[Attributes]` specify the role of the different attributes. In our learning problem, the first two attributes (attributes 1-2: *outlook* and *windy*) are *descriptive* attributes: they are to be used in the cluster descriptions, that is, in the tests that appear in the predictive clustering tree's nodes (or, in rule learning, the conditions that appear in predictive clustering rules). The last two attributes (attributes 3-4) are so-called *target* attributes: these are to be predicted from the descriptive attributes. The setting `Clustering = 3-4` indicates that the clustering heuristic, which is used to construct the tree, should be computed based on the target attributes only. (That is, CLUS should try to produce clusters that are coherent with respect to the target attributes, not necessarily with respect to all attributes.) Finally, in the `Tree` section of the settings file, which contains parameters specific to tree learning, `Heuristic = VarianceReduction` specifies that, among different clustering heuristics that are available, the heuristic that should be used for this run is variance reduction.

   These are only a few possible settings. Chapter 4 provides a detailed description of each setting supported by CLUS.

4. Now that we have some idea of what the settings file and data file look like, let's run CLUS on these data and see what the result is. From the Unix command line, type, in the directory where the weather files are:

   ```
   java -jar ../../Clus.jar weather.s
   ```

5. CLUS now reads the data and settings files, performs it computations, and writes the resulting predictive clustering tree, together with a number of statistics such as the training set error and the test set error (if a test set has been provided), to an output file, `weather.out`. Open that file and inspect its contents; it should look like the file shown in Figure 2.4. The file contains information about the CLUS run, including some statistics, and of course also the final result: the predictive clustering tree that we wanted to learn. By default, CLUS shows both an "original model" (the tree before pruning it) and a "pruned model", which is a simplified version of the original one.

   In this example, the resulting tree is a multi-target tree: each leaf predicts a vector of which the first component is the predicted *temperature* (attribute 3) and the second component the predicted *humidity* (attribute 4). A feature that distinguishes CLUS from other decision tree learners is exactly the fact that CLUS can produce this kind of trees. Constructing a multi-target tree has several advantages over constructing a separate regression tree for each target variable. The most obvious one is the

number of models: the user only has to interpret one tree instead of one tree for each target. A second advantage is that the tree makes features that are relevant to all target variables explicit. For example, the first leaf of the tree in Figure 2.4 shows that `outlook = sunny` implies both a high temperature and a low humidity. Finally, due to so-called inductive transfer, multi-target PCTs may also be more accurate than regression trees. More information about multi-target trees can be found in the following publications: [3, 2, 25, 20].

```
Clus run "weather"
******************
Date: 1/10/10 12:23 PM
File: weather.out
Attributes: 4 (input: 2, output: 2)

[Data]
File = weather.arff

[Attributes]
Target = 3-4
Clustering = 3-4
Descriptive = 1-2

[Tree]
Heuristic = VarianceReduction
PruningMethod = M5

Statistics
----------
Induction Time: 0.017 sec
Pruning Time: 0.001 sec
Model information
     Original: Nodes = 7 (Leaves: 4)
     Pruned: Nodes = 3 (Leaves: 2)

Training error
--------------
Number of examples: 8
Mean absolute error (MAE)
   Default        : [7.125,14.75]: 10.9375
   Original       : [2.125,2.75]: 2.4375
   Pruned         : [4.125,7.125]: 5.625
Mean squared error (MSE)
   Default        : [76.8594,275.4375]: 176.1484
   Original       : [6.5625,7.75]: 7.1562
   Pruned         : [19.4375,71.25]: 45.3438

Original Model
**************
outlook = sunny
+--yes: [32,52.5]: 2
+--no:  outlook = rainy
        +--yes: windy = yes
        |       +--yes: [9,92.5]: 2
        |       +--no:  [19,91.5]: 2
        +--no:  [15.5,72.5]: 2

Pruned Model
************
outlook = sunny
+--yes: [32,52.5]: 2
+--no:  [14.5,85.5]: 6
```

Figure 2.4: The *Weather* example's output (`weather.out`). (Some parts have been omitted for brevity.)

Chapter 3

Input Format

Like many machine learning systems, CLUS learns from tabular data. These data are assumed to be in the ARFF format that is also used by the Weka data mining tool. Full details on ARFF can be found elsewhere[1]. We only give a minimal description here.

In the data table, each row represents an instance, and each column represents an attribute of the instances. Each attribute has a name and a domain (the domain is the set of values it can take). In the ARFF format, the names and domains of the attributes are declared up front, before the data are given. The syntax is not case sensitive. An ARFF file has the following format:

```
% all comment lines are optional, start with %, and can occur
% anywhere in the file

@RELATION name

@ATTRIBUTE name domain
@ATTRIBUTE name domain
...

@DATA
value1, value2, ... , valuen
value1, value2, ... , valuen
```

The domain of an attribute can be one of:

- `numeric`
- { nomvalue$_1$, nomvalue$_2$, ... , nomvalue$_n$ }
- `string`
- `hierarchical` hvalue$_1$, hvalue$_2$, ... , hvalue$_n$
- `timeseries`

The first option, `numeric` (`real` and `integer` are also legal and are treated in the same way), indicates that the domain is the set of real numbers. The second type of domain is called a discrete domain. Discrete domains are defined by enumerating the values they contain. These values are nominal. The third domain type is `string` and can be used for attributes containing arbitrary textual values.

The fourth type of domain is called `hierarchical` (multi-label). It implies two things: first, the attribute can take as a value a *set of values* from the domain, rather than just a single value; second, the domain has a hierarchical structure. The elements of the domain are typically denoted $v_1/v_2/.../v_i$, with $i \leq d$, where d is the depth of the hierarchy. A set of such elements is denoted by just listing them, separated by `@`. This type of domain is useful in the context of hierarchical multi-label classification and is not part of the standard ARFF syntax.

[1] `http://weka.wikispaces.com/ARFF`

```
@RELATION HMCNewsGroups

@ATTRIBUTE word1   {1,0}
...
@ATTRIBUTE word100 {1,0}
@ATTRIBUTE class hierarchical rec/sport/swim,rec/sport/run,rec/auto,alt/atheism,...

@DATA
1,...,1,rec/sport/swim
1,...,1,rec/sport/run
1,...,1,rec/sport/run@rec/sport/swim
1,...,0,rec/sport
1,...,0,rec/auto
0,...,0,alt/atheism
...
```

Figure 3.1: An ARFF file that includes a hierarchical multi-label attribute.

```
@RELATION GeneExpressionTimeSeries

@ATTRIBUTE geneid string
@ATTRIBUTE GO0000003 {1,0}
@ATTRIBUTE GO0000004 {1,0}
...
@ATTRIBUTE GO0051704 {1,0}
@ATTRIBUTE GO0051726 {1,0}
@ATTRIBUTE target timeseries

@DATA
YAL001C,0,0,0,0,0,0,0,0,0,0,0,0,0,...,0,0,0,0,[0.07, 0.15, 0.14, 0.15,-0.11, 0.07,-0.41]
YAL002W,0,0,0,0,0,0,0,0,0,0,1,0,0,...,1,1,0,0,[0.14, 0.14, 0.18, 0.14, 0.17, 0.13, 0.07]
YAL003W,0,0,0,0,0,0,0,0,0,0,0,0,0,...,0,0,0,0,[0.46, 0.33, 0.04,-0.60,-0.64,-0.51,-0.36]
YAL005C,0,0,0,0,0,0,0,0,0,0,0,0,0,...,1,1,0,0,[0.86, 1.19, 1.58, 0.93, 1,    0.85, 1.24]
YAL007C,0,0,0,0,0,0,0,0,0,0,0,0,0,...,1,1,0,0,[0.12, 0.49, 0.62, 0.49, 0.84, 0.89, 1.08]
YAL008W,0,1,0,0,0,0,0,0,0,0,0,0,0,...,0,0,0,0,[0.49, 1.01, 1.33, 1.23, 1.32, 1.03, 1.14]
...
```

Figure 3.2: An ARFF file that includes a time series attribute.

The last type of domain is `timeseries`. A time series is a fixed length series of numeric data where individual numbers are written in brackets and separated with commas. All time series of a given attribute must be of the same length. This domain type, too, is not part of the standard ARFF syntax.

The values in a row occur in the same order as the attributes: the i'th value is assigned to the i'th attribute. The values must, obviously, be elements of the specified domain.

Clus also supports the sparse ARFF format, where only non-zero data values are stored for the numeric attributes. The header of a sparse ARFF file is the same, but each data instance is written in curly braces and each attribute value is written as a pair of the attribute number (starting from one) and its value separated by a space; values of different attributes are separated by commas.

Figure 2.3 shows an example of an ARFF file. An example of a table containing hierarchical multi-label attributes is shown in Figure 3.1, an example ARFF file with a time series attribute is shown in Figure 3.2, and an example sparse ARFF file is shown in Figure 3.3.

```
@RELATION SparseData

@ATTRIBUTE a1    numeric
@ATTRIBUTE a2    numeric
...
@ATTRIBUTE a10   numeric
@ATTRIBUTE a11   numeric
@ATTRIBUTE class {pos,neg}

@DATA
{1 3.1, 8 2.5, 12 pos}
{7 2.3, 12 neg}
{2 8.5, 3 1.3, 12 neg}
{1 3.2, 12 pos}
{1 3.3, 8 2.7, 12 pos}
...
```

Figure 3.3: An ARFF file in sparse format.

Chapter 4

Settings File

The algorithms included in the CLUS system have a number of parameters that influence their behavior. Most parameters have a default setting; the specification of a value for such parameters is optional. For parameters that do not have a default setting or which should get another value than the default, a value must be specified in the settings file, *filename*.s.

The settings file is structured into sections. Each parameter belongs to a particular section. Including the section headers (section names written in brackets) is necessary, since some names of the settings are not unique, e.g., `Iterations` appear in the sections `[Ensemble]` and `SemiSupervised`. These headers also help users structure the settings.

We here explain the most common settings. Some settings that are connected to experimental or not yet fully implemented features of CLUS are either marked as such or not presented at all. Figure 4.1 shows an example of a settings file. All the settings (including the default ones) that were used in a specific CLUS run are printed at the beginning of the output file (*filename*.out).

4.1 General

- `Verbose`:
  - *possible values*: a nonnegative integer
  - *default*: `1`
  - *description* : specifies the verbosity of the information, printed to standard output.
- `RandomSeed`:
  - *possible values*: a nonnegative integer
  - *default*: `0`
  - *description* : Used to initialize the object that takes care of random number generation. Some procedures used by CLUS (e.g., creation of cross-validation folds) are randomized, and as a result, different runs of CLUS on identical data may still yield different outputs. When CLUS is run on identical input data with the same `RandomSeed` setting, it is guaranteed to yield the same results.

4.2 Data

All paths (see, for example `File`), can be given as absolute paths or relative paths (relative to the CLUS executable).

- `File`:
  - *possible values*: a string (path to the file)
  - *default*: `None`
  - *description* : The name of the file that contains the training set. The default value is internally changed to *filename*.arff. CLUS can read compressed (.arff.zip) or uncompressed (.arff) data files.

```
[General]
RandomSeed = 0              % seed of random generator

[Data]
File = weather.arff         % training data
TestSet = None              % data used for evaluation (file name / proportion)
PruneSet = None             % data used for tree pruning (file name / proportion)
XVal = 10                   % number of folds in cross-validation (clus -xval ...)

[Attributes]
Target = 5                  % index of target attributes
Disable = 4                 % Disables some attributes (e.g., "5,7-8")
Key = None                  % Sets the index of the key attribute
Weights = Normalize         % Normalize numeric attributes

[Model]
MinimalWeight = 2.0         % at least 2 examples in each subtree

[Tree]
FTest = 1.0                 % f-test stopping criterion for regression
ConvertToRules = No         % Convert the tree to a set of rules

[Constraints]
Syntactic = None            % file with syntactic constraints (a partial tree)
MaxSize = Infinity          % maximum size for Garofalakis pruning
MaxError = Infinity         % maximum error for Garofalakis pruning
MaxDepth = Infinity         % Stop building the tree at the given depth

[Output]
AllFoldModels = Yes         % Output model in each cross-validation fold
AllFoldErrors = No          % Output error measures for each fold
TrainErrors = Yes           % Output training error measures
UnknownFrequency = No       % proportion of missing values for each test
BranchFrequency = No        % proportion of instances for which test succeeds
WritePredictions = {Train,Test}    % write test set predictions to file

[Beam]
SizePenalty = 0.1           % size penalty parameter used in the beam heuristic
BeamWidth = 10              % beam width
MaxSize = Infinity          % Sets the maximum size constraint
```

Figure 4.1: An example settings file

- `TestSet`:
  - *possible values*: a string s (path to the file) or a real number p from the interval $[0, 1]$
  - *default*: `None`
  - *description* : If the default value `None` is used, no test set is used. If s is a valid file name containing a test set in ARFF format, CLUS will evaluate the learned model on this test set. If this settings is specified as a real number p, CLUS will use a proportion p of the data file as a separate test set (used for evaluating the model but not for training).
- `PruneSet`:
  - *possible values*: same as for `TestSet`
  - *default*: `None`

  - *description* : defines whether and how to use a pruning set; the meaning of this setting is analogous to `TestSet` setting.

- `XVal`:
  - *possible values*: an integer greater or equal to 2
  - *default*: `10`
  - *description* : the number of folds to be used in a cross-validation. To perform cross-validation, CLUS needs to be run with the `-xval` command line parameter.

4.3 Attributes

Note: all indices in this section are 1-based.

- `Target`:
  - *possible values*: an index or an interval or a comma-separated string of the first two options, e.g. `21` or `1-10` or `1-10,21,314`
  - *default*: `Default`
  - *description* : specifies the indices of the target attributes, i.e., the attributes which the predictions are made for. If this setting is not specified, then it is equal to the index of the last attribute in the training dataset, i.e., the last attribute is the target by default. This setting overrides the `Disable` setting. This is convenient if one needs to build models that predict only a subset S of all available target attributes T (and other target attributes should not be used as descriptive attributes). Because `Target` overrides `Disable`, this can be achieved by setting `Disable` to T `Target` to S.

- `Clustering`:
  - *possible values*: an index or an interval or a comma-separated string of the first two options, e.g. `21` or `1-10` or `1-10,21,314`
  - *default*: `Default`
  - *description* : The predictive clustering heuristic that is used to guide the model construction is computed with regard to these atrributes. If this setting is not specified, then the clustering attributes are by default equal to the target attributes.

- `Descriptive`:
  - *possible values*: an index or an interval or a comma-separated string of the first two options, e.g. `21` or `1-10` or `1-10,21,314`
  - *default*: `Default`
  - *description* : sets the range of attributes that can be used in the descriptive part of the models. For a PCT, these attributes will be used to construct the tests in the internal nodes of the tree. For a set of PCRs, these attributes will appear in the rule conditions. If this setting is not specified, then the descriptive attributes are all attributes that are not `Target`, `Key`, or `Disable`.

- `Key`:
  - *possible values*: an index or an interval or a comma-separated string of the first two options, e.g. `21` or `1-10` or `1-10,21,314`
  - *default*: `None`
  - *description* : sets the range of key attributes. A key attribute or a set of key attributes can be used as an example identifier. For example, if each instance represents a person, then the key attribute could store the person's name. Key attributes are not actually used by the induction algorithm, but they are written to output files, for example, to ARFF files with predictions. See `[Output]/WritePredictions` for an example.

- `Disable`:

- *possible values*: an index or an interval or a comma-separated string of the first two options, e.g. `21` or `1-10` or `1-10,21,314`
- *default*: `None`
- *description* : sets the range of attributes that are to be ignored by CLUS. These attributes are not read into memory.

- `Weights`:
  - *possible values*: a nonnegative positive real number or a list of such numbers or `Normalize`
  - *default*: `Normalize`
  - *description* : sets the relative weights w_i of the attributes x_i in the clustering heuristic. If given as a list, the weight w_i equals the i-th element of the list. If given as a single number, e.g., 1.0 all the weights of all clustering attributes are set to this weight. To use weights $w_i = 1/\mathrm{Var}(x_i)$, with $\mathrm{Var}(x_i)$ the variance of attribute x_i in the input data, use (the default option) `Normalize`.

- `GIS`:
  - *possible values*: an index or an interval or a comma-separated string of the first two options, e.g. `21` or `1-10` or `1-10,21,314`
  - *default*: `None`
  - *description* : specifies the indices of GIS attributes

4.4 Model

- `MinimalWeight`:
  - *possible values*: a positive real number
  - *default*: `2.0`
  - *description* : CLUS only generates clusters with at least r instances in each subset (tree nodes or rules). This is a standard setting used for pre-pruning of trees and rules.

- `NominalSubsetTests`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `Yes`
  - *description* : If set to `No`, only equality tests for nominal attributes are allowed. Otherwise, tests such as `attr value is an element of {green, blue, red}` are allowed also.

4.5 Tree

- `Heuristic`:
  - *possible values*: {`Default`, `ReducedError`, `Gain`, `GainRatio`, `VarianceReduction`, `MEstimate`, `Morishita`, `DispersionAdt`, `DispersionMlt`, `RDispersionAdt`, `RDispersionMlt`, `VarianceReductionGIS`}
  - *default*: `Default`
  - *description* : Sets the heuristic function that is used for evaluating the clusters (splits) when generating trees or rules. Please note that this setting is used for trees as well as rules.
    - `Default`: default heuristic. If learning trees this is equal to `VarianceReduction`, if learning rules this setting is equal to `RDispersionMlt`.
    - `ReducedError`: reduced error heuristic, can be used for trees.
    - `Gain`: information gain heuristic, can be used for classification trees.
    - `GainRatio`: information gain ratio heuristic [21], can be used for classification trees.
    - `VarianceReduction`: variance reduction heuristic, can be used for trees.
    - `MEstimate`: m-estimate heuristic [7], can be used for classification trees.
    - `Morishita`: Morishita heuristic [24], can be used for trees.

    * `DispersionAdt`: additive dispersion heuristic [28] pages 37–38, can be used for rules.
    * `DispersionMlt`: multiplicative dispersion heuristic [28] pages 37–38, can be used for rules.
    * `RDispersionAdt`: additive relative dispersion heuristic [28] pages 37–38, can be used for rules.
    * `RDispersionMlt`: multiplicative relative dispersion heuristic [28] pages 37–38, can be used for rules, the default heuristic for learning predictive clustering rules.
    * `VarianceReductionGIS`: used when GIS-attributes are present in the data

- `PruningMethod`:
  - *possible values*: an element of {`Default`, `None`, `C4.5`, `M5`, `M5Multi`, `ReducedErrorVSB`, `Garofalakis`, `GarofalakisVSB`, `CartVSB`, `CartMaxSize`}
  - *default*: `Default`
  - *description* : Sets the post-pruning method for trees.
    * `Default`: default pruning method for trees, if learning classification trees this is equal to`C4.5`, if learning regression trees this is equal to`M5`.
    * `None`: no post-pruning of learned trees is performed.
    * `C4.5`: pruning as in C4.5 [23], can be used for classification trees,
    * `M5`: pruning as in M5 [22], can be used for regression trees,
    * `M5Multi`: experimental modification to M5 [22] pruning for multi-target regression trees.
    * `ReducedErrorVSB`: reduced error pruning where the error is estimated on a separate validation data set (VSB = validation set based pruning).
    * `Garofalakis`: pruning method proposed by Garofalakis et al. [10] used for constraint induction of trees.
    * `GarofalakisVSB`: same as`Garofalakis`, but the error is estimated on a separate validation data set.
    * `CartVSB`: pruning method that is implemented in CART [5], and uses a separate validation set. It seems to work better than M5 on the multi-target regression data sets.
    * `CartMaxSize`: pruning method that is also implemented in CART [5], but uses cross-validation to tune the pruning parameter to achieve the desired tree size.

- `FTest`:
  - *possible values*: a real number r from the interval $[0, 1]$ or a list of such numbers, e.g., `0.001` or `[0.001,0.005,0.01,0.05,0.1,0.125]`
  - *default*: `1.0`
  - *description* : sets the f-test stopping criterion for regression; a node will only be split if a statistical F-test indicates a significant (at level r) reduction of variance inside the subsets. The f-test level can also be optimized by providing a vector of levels. In that case, the (smallest) f-test level will be chosen that minimizes the RMSE measure on the validation set provided (using the `PruneSet` setting).

- `BinarySplit`:
  - *possible values*: {`Yes`, `No`}
  - *default*: `Yes`
  - *description* : specifies whether only binary splits are used in the tree induction

- `ConvertToRules`:
  - *possible values*: an element of {`No`, `Leaves`, `AllNodes`}
  - *default*: `No`
  - *description* : Clus can convert a tree (or ensemble of trees) into a set of rules. This setting can be used for learning rule ensembles [1].
    * `No`: the tree is not converted to rules
    * `Leaves`: only tree leaves are converted to rules

* `AllNodes`: the internal nodes of tree(s) are converted also

- `SplitSampling`:
  - *possible values*: a nonnegative integer n
  - *default*: `0`
  - *description* : the split heuristic can be calculated on a sample of the training set. For $n > 0$, the training set is, for each split, sampled with replacement to form a sample of size n. Otherwise, the training set is used as is.
- `MissingClusteringAttrHandling`:
  - *possible values*: an element of {`Ignore`, `EstimateFromTrainingSet`, `EstimateFromParentNode`}
  - *default*: `EstimateFromParentNode`
  - *description* : determines how we handle the case where when searching evaluating candidate split all examples have only missing values for a clustering attribute, in one of the branches.
- `MissingTargetAttrHandling`:
  - *possible values*: an element of {`Zero`, `DefaultModel`, `ParentNode`}
  - *default*: `ParentNode`
  - *description* : determines how we calculate prototype (i.e., prediction) if all the tuple in leaf node have only missing values for target attriute
- `InductionOrder`:
  - *possible values*: {`DepthFirst`, `BestFirst`}
  - *default*: `DepthFirst`
  - *description* : specifies the tree induction order (depth first search or best first search)

4.6 Rules

- `CoveringMethod`:
  - *possible values*: an element of {`Standard`, `WeightedError`, `RandomRuleSet`, `HeurOnly`, `RulesFromTree`}.
  - *default*: `Standard`
  - *description* : Defines how the rules are generated:
    * `Standard`: standard covering algorithm [17], all examples covered by the new rule are removed from the current learning set, can be used for learning ordered rules.
    * `WeightedError`: error weighted covering algorithm [28] (Section 4.5), examples covered by the new rule are not removed from the current learning set, but their weight is decreased inversely proportional to the error the new rule makes when predicting their target values, can be used for learning unordered rules.
    * `RandomRuleSet`: rules are generated randomly, (experimental feature).
    * `HeurOnly`: no covering is used, the heuristic function takes into account the already learned rules and the examples they cover to focus on yet uncovered examples, (experimental feature).
    * `RulesFromTree`: rules are not learned with the covering approach, but a tree is learned first and then transcribed into a rule set. After this e.g. rule weight optimization methods can be used.
- `RuleAddingMethod`:
  - *possible values*: an element of {`Always`, `IfBetter`, `IfBetterBeam`}
  - *default*: `Always`
  - *description* : Defines how rules are added to the rule set. For regression rules setting this option to `IfBetter` is recommended.

    - `Always`: each rule when constructed is always added to the rule set,
    - `IfBetter`: rule is only added to the rule set if the performance of the rule set with the new rule is better than without it,
    - `IfBetterBeam`: similar to `IfBetter`, but if the rule does not improve the performance of the rule set, other rules from the beam are also evaluated and possibly added to the rule set.

- `CoveringWeight`:
  - *possible values*: a real number from the interval $[0, 1]$
  - *default*: `0.1`
  - *description* : weight controlling the amount by which weights of covered examples are reduced within the error weighted covering algorithm – ζ in [28] (Section 4.5, Equations 4.6 and 4.8), can be used for unordered rules with error weighted covering method.

- `InstCoveringWeightThreshold`:
  - *possible values*: a real number from the interval $[0, 1]$
  - *default*: `0.1`
  - *description* : instance weight threshold used in error weighted covering algorithm for learning unordered rules. When an instance's weight falls below this threshold, it is removed from the current learning set. ϵ in [28] (Section 4.5)

- `MaxRulesNb`:
  - *possible values*: a positive integer
  - *default*: `1000`
  - *description* : defines a maximum number of rules in a rule set.

- `ComputeDispersion`:
  - *possible values*: {`Yes`, `No`}
  - *default*: `No`
  - *description* : If set to `Yes`, CLUS will print some additional dispersion estimation for each rule and entire rule set.

- `PrintRuleWiseErrors`:
  - *possible values*: {`Yes`, `No`}
  - *default*: `No`
  - *description* : If `Yes`, CLUS will print error estimation for each rule separately.

- `OptAddLinearTerms`:
  - *possible values*: an element of {`No`, `Yes`, `YesSaveMemory`}
  - *default*: `No`
  - *description* : Defines whether to add descriptive attributes as linear terms to the rule set. Usually this increases the accuracy. Especially for multi-target data sets it also slows the algorithm down. For these, use value `YesSaveMemory`, otherwise it can take a lot of memory. For single target data sets `Yes` is faster. Used for learning rule ensembles [1].

- `OptNormalizeLinearTerms`:
  - *possible values*: an element of {`No`, `Yes`, `YesAndConvert`}
  - *default*: `Yes`
  - *description* : Defines whether the linear terms are scaled so that each descriptive attribute has a similar effect. The default setting `Yes` and it should always be used. However, if you want to transform the rule ensemble so that linear terms are of "standard type", you may use `YesAndConvert` setting. This moves the effect of normalizations to weights and default prediction after optimization. Used for learning rule ensembles [1].

- `OptLinearTermsTruncate`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `Yes`
  - *description* : Used in conjunction with the `OptAddLinearTerms` setting. If `Yes`, the linear terms are truncated so that they do not predict values greater or smaller than found in the training set. This adds more robustness against outliers. Used for learning rule ensembles [1].
- `OptGDMaxIter`:
  - *possible values*: a positive integer
  - *default*: `1000`
  - *description* : defines a number of iterations that a gradient descent algorithm for optimizing rule weights makes, used for learning rule ensembles [1]. The default value is 1000.
- `OptGDGradTreshold`:
  - *possible values*: a real value from the interval $[0, 1]$
  - *default*: `1.0`
  - *description* : the τ treshold value for the gradient descent (GD) algorithm used for learning rule ensembles [1]. τ defines the limit by which gradients are changed during every iteration of the GD algorithm. If $\tau = 1$ effect is similar to L1 regularization (Lasso) and $\tau = 0$ the effect is similar to L2. If `OptGDMaxNbWeights` is low (less than 40), setting $\tau = 1$ is usually enough (it is the fastest).
- `OptGDStepSize`:
  - *possible values*: a positive real number
  - *default*: `0.1`
  - *description* : If `OptGDIsDynStepsize` is set to `No`, the initial gradient descent step size factor.
- `OptGDIsDynStepsize`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `Yes`
  - *description* : Do we use as the step size factor a lower limit of optimal one? The value is computed based on the rule prediction values. Usually faster (lower step sizes are not tried at all) and often also more accurate than a given value.
- `OptGDMaxNbWeights`:
  - *possible values*: 0 or a positive integer
  - *default*: `0`
  - *description* : defines a maximum number of of allowed nonzero weights for rules/linear terms, used for learning rule ensembles [1]. If we have enough modified weights, only the nonzero ones are altered for the rest of the optimization. With this we can limit the size of the rule set. The default value of `0` means no rule set size limitation.
- `OptGDNbOfTParameterTry`:
  - *possible values*: a positive integer
  - *default*: `1`
  - *description* : Defines how many different τ values are checked between 1 and `OptGDGradTreshold`. We use a validation set to compute, which τ value gives the best accuracy. If `OptGDMaxNbWeights` is low, usually only a single value τ=1 is enough (fastest).
- `OptGDEarlyTTryStop`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `Yes`
  - *description* : specifies whether do we stop if validation error starts to increase too much, when trying different τ values starting from 1. Usually a lot faster, but may decrease the accuracy.

4.7 Ensemble

- `Iterations`:
  - *possible values*: positive integer n, $n \geq 2$, or a list of such integers
  - *default*: `100`
  - *description* : defines the number of base-level models (trees) in the ensemble. If this setting is given as a list of numbers $[n_1, n_2, \dots]$, such that $n_1 < n_2 < \cdots$, an ensemble $\mathcal{E}_i$ for each element of the list is computed in an efficient way, so that $\mathcal{E}_1$ consist of n_1 trees and is contained in $\mathcal{E}_2$ etc.
- `EnsembleMethod`:
  - *possible values*: an element from {`Bagging`, `RForest`, `RSubspaces`, `BagSubspace`, `Boosting`, `RFeatSelection`, `Pert`, `ExtraTrees`}
  - *default*: `Bagging`
  - *description* : defines the ensemble method.
- `VotingType`:
  - *possible values*: an element of {`Majority`, `ProbabilityDistribution`}
  - *default*: `ProbabilityDistribution`
  - *description* :
    - `Majority`: each base-level model casts one vote, for regression this is equal to averaging.
    - `ProbabilityDistribution`: each base-level model casts probability distributions for each target attribute, does not work for regression.
- `SelectRandomSubspaces`:
  - *possible values*: an integer k from the interval $[1, \text{number of attributes}]$ or a real number q from the interval $[0, 1]$, or an element of {`LOG`, `SQRT`}
  - *default*: `0`
  - *description* :
    - `SQRT` or `0`: the feature subset size is set to $\lceil\sqrt{\text{number of descriptive attributes}}\rceil$.
    - `LOG`: the feature subset size is set to $\lceil\log_2(\text{number of descriptive attributes})\rceil$ (recommended by Breiman [4] but his datasets are rather low-dimensional).
    - k: an integer number that specifies the feature subset size.
    - q: the fraction of the descriptive attributes used as feature subset, i.e., the feature subset value is set to $[q(\text{number of descriptive attributes})]$.
- `PrintAllModels`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : specifies whether the base models are included in the `.out` file.
- `PrintAllModelFiles`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : If `Yes`, CLUS will save all base-level models of an ensemble in the model file. The default setting prevents creating very large model files.
- `PrintAllModelInfo`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : If `Yes`, CLUS will include additional model information in the `.out` file.

- `PrintPaths`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* :If `Yes`, for each tree, the path that is followed by each instance in that tree will be printed to a file named `tree_i.path`. The default setting is `No`. Currently, the setting is only implemented for `Bagging` and `RForest`. This setting was used in random forest based feature induction, see [26] for details
- `Optimize`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : If set to `Yes`, CLUS will optimize memory usage during learning. In that case, the tree induction/predictive error computation procedure is slightly modified. For example, the predictions of the ensemble are updated every time a new tree is grown. After that, the tree is removed from the memory. Due to floating point arithmetic, this could result in slightly different predictions.
- `OOBestimate`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : If `Yes`, out-of-bag estimate of the performance of the ensemble will be done.
- `FeatureRanking`:
  - *possible values*: an element of {`None`, `RForest`, `GENIE3`, `SYMBOLIC`}
  - *default*: `None`
  - *description* : If set to non-default value, the specified ranking will be computed in addition to the ensemble predictive model.
    - `RForest`: random forest ranking as described in [19], does not work with extra trees (no out-of-bag examples)
    - `GENIE3`: Genie3 ranking as given in [19]
    - `SYMBOLIC`: Symbolic ranking, as described in [19]
- `FeatureRankingPerTarget`:
  - *possible values*:an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : If the `FeatureRanking`, the *number of targets* additional feature rankings are computed, each one only with respect to one target. This option is ignored when the additional rankings are the same as the original one, e.g., when there is only one target or `FeatureRanking` is set to `SYMBOLIC`.
- `SymbolicWeight`:
  - *possible values*: a real number w from the interval $(0, 1]$.
  - *default*: `1.0`
  - *description* : every appearance of an attribute at the depth d in a tree increases the importance score of `SYMBOLIC` by w^d. If `FeatureRanking` is not set to `SYMBOLIC`, this option has no influence.
- `SortRankingByRelevance`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `Yes`
  - *description* : If set to `Yes`, the attributes in the `.fimp` file are sorted decreasingly by relevance. Otherwise, they are in the same as in `.arff` file.

- `WriteEnsemblePredictions`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : If set to `Yes`, the predictions of the ensemble for the training set are written to *`filename`*`.ens.train.preds`. If the option `TestSet` is also specified, the predictions of the ensemble for the testing set are written to *`filename`*`.ens.test.preds`.
- `BagSize`:
  - *possible values*: an integer from the interval [0, number of instances]
  - *default*: `0`
  - *description* : Specifies the size of a bag. If set to 0, this will be converted to the *number of instances*.
- `NumberOfThreads`:
  - *possible values*: An integer, greater or equal to 1
  - *default*: `1`
  - *description* : Specifies the number of threads that run in parallel when inducing the ensemble from {`Bagging`, `RForest`, `ExtraTrees`}, or computing `Relief` feature ranking.

4.8 Constraints

- `Syntactic`:
  - *possible values*: a string
  - *default*: `None`
  - *description* : sets the file with syntactic constraints (e.g., a partial tree) [25].
- `MaxSize`:
  - *possible values*: a positive integer or `Infinity`
  - *default*: `Infinity`
  - *description* : sets the maximum size for Garofalakis pruning [10, 25].
- `MaxError`:
  - *possible values*: a positive real number of `Infinity`
  - *default*: `0.0`
  - *description* : sets the maximum error for Garofalakis pruning.
- `MaxDepth`:
  - *possible values*: a positive integer or `Infinity`
  - *default*: `-1`
  - *description* : Clus will build trees with the depth that does not exceed the specified value. In the context of rule ensemble learning [1], this sets the average maximum depth of trees that are then converted into rules and a value of 3 seems to work fine.

4.9 Output

- `ShowModels`:
  - *possible values*: a subset of {`Default`, `Original`, `Pruned`, `Others`}
  - *default*: {`Default, Pruned, Others`}
  - *description* : specifies which models are shown in the `.out` file
- `TrainErrors`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `Yes`
  - *description* : if set to `Yes`, training errors will be computed and included in the `.out` file
- `TestErrors`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `Yes`
  - *description* : if set to `Yes`, testing errors will be computed and included in the `.out` file
- `AllFoldModels`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `Yes`
  - *description* : if set to `Yes`, CLUS will output the model built in each fold of a cross-validation.
- `AllFoldErrors`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : if set to `Yes`, CLUS will output the test set error (and other evaluation measures) for each fold.
- `AllFoldDatasets`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : if set to `Yes`, CLUS will output the test set error (and other evaluation measures) for each fold.
- `UnknownFrequency`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : if set to `Yes`, CLUS will show in each node of the tree the proportion of instances that had a missing value for the test in that node.
- `BranchFrequency`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : if set to `Yes`, CLUS will show in each node of the tree, for each possible outcome of the test in that node, the proportion of instances that had that outcome.
- `ShowInfo`:
  - *possible values*: an element of {`Count`, `CountByTarget`, `Distribution`, `Index`, `NodePrototypes`, `Key`}
  - *default*: {`Count`}

- *description* : specifies the type of information shown in the models in `.out` file

- `PrintModelAndExamples`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : if set to `Yes`, the leaves of the models in the `.out` file will include some basic information about the distribution of values of the numeric descriptive attributes: minimal and maximal value, average and standard deviation.

- `WriteModelFile`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : if set to `Yes`, the `.model` file will be created

- `WritePredictions`:
  - *possible values*: a subset of {`Train`, `Test`} or `None`
  - *default*: {`None`}
  - *description* : If `Train` is included, then the prediction for each training instance will be written to an ARFF output file *`filename`*`.train.`*`i`*`.pred.arff` with i the iteration. In a single run, $i = 1$. In a 10-fold cross-validation, i will vary from 1 to 10. If `Test`, then the predictions for each test instance will be written to the file *`filename`*`.test.pred.arff`.

- `GzipOutput`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : if set to `Yes`, CLUS will compress output in gzip file format (approx. 10 times smaller file sizes).

4.10 Beam

- `SizePenalty`:
  - *possible values*: a positive real number
  - *default*: `0.1`
  - *description* : sets the size penalty parameter used in the beam heuristic [11].

- `BeamWidth`:
  - *possible values*: a positive integer
  - *default*: `10`
  - *description* : sets the width of the beam used in the beam search performed by CLUS [11].

- `MaxSize`:
  - *possible values*: a positive integer or `Infinity`
  - *default*: `-1`
  - *description* : sets the maximum size constraint [11].

4.11 HMLC

Settings for hierarchical multi-label classification. Figure 4.2 summarizes these settings briefly. If the hierarchy is `Tree`, possible values of a HML attribute are given as

```
@attribute myAttr hierarchical L1,L1<HSeparator>L1.1,L1<HSeparator>L1.1<HSeparator>L1.1.1, ...
```

i.e., as a comma separated list of all paths from the root label(s) that does not necessarily end in a leaf of hierarchy. If the hierarchy is `DAG`, the possible values of a HML attribute are given as

```
@attribute myAttr hierarchical L1,L1<HSeparator>L1.1,L1.1<HSeparator>L1.1.1, ...
```

i.e., as a comma separated list of all child-parent pairs. The option `HSeparator` is defined below.

- `Type`:
  - *possible values*: {`Tree`, `DAG`}
  - *default*: `Tree`
  - *description* : specifies the hierarchy type: tree or directed acyclic graph [27]
- `Distance`:
  - *possible values*: an element of {`WeightedEuclidean`}
  - *default*: `WeightedEuclidean`
  - *description* : internally, a list of relevant labels in the hierarchy (for a given example) is transformed into a vector of zeros and ones. The distance between two such vectors u and v is then computed as
    $$d(u,v) = \sqrt{\sum_i w_i (u[i] - v[i])^2},$$
    where the weight w_i on the depth d is defined by `WType`.
- `WType`:
  - *possible values*: an element of {`ExpSumParentWeight`, `ExpAvgParentWeight`, `ExpMinParentWeight`, `ExpMaxParentWeight`, `NoWeight`}
  - *default*: `ExpSumParentWeight`
  - *description* : if set to `NoWeight`, all weights equal 1. Otherwise, the weight w is computed as sum/average/minimum/maximum of parents' weights, multiplied by w_0. If the `Type` is set to `Tree`, the weight w on the depth d amounts to $w = w_0^d$. The value of w_0 is specified by `WParam`.
- `WParam`:
  - *possible values*: a real number from the interval $(0, 1]$
  - *default*: `0.75`
  - *description* : specifies, how fast the weights of the labels decrease with the depth in the hierarchy ([27], Section 4.1)
- `HSeparator`:
  - *possible values*: a character
  - *default*: `.`
  - *description* : is the separator used in the notation of values of the hierarchical domain (typically `/` or `.`)
- `EmptySetIndicator`:
  - *possible values*: a character
  - *default*: `n`
  - *description* : the symbol used to indicate the empty set

- `OptimizeErrorMeasure`:
  - *possible values*: an element of {`AverageAUROC`, `AverageAUPRC`, `WeightedAverageAUPRC`, `PooledAUPRC`}
  - *default*: `PooledAUPRC`
  - *description* : Clus can automatically optimize the `FTest` setting (see earlier). This parameter indicates what criterion should be maximized for this ([27], Section 5.2)
    - `AverageAUROC`: average of the areas under the class-wise ROC curves
    - `AverageAUPRC`: average of the areas under the class-wise precision-recall curves
    - `WeightedAverageAUPRC`: similar to `AverageAUPRC`, but each class's contribution is weighted by its relative frequency
    - `PooledAUPRC`: area under the average (or pooled) precision-recall curve
- `ClassificationThreshold`:
  - *possible values*: a real number from the interval $[0, 1]$ or a list of such values
  - *default*: `None`
  - *description* : The original tree constructed by Clus contains a vector of predicted probabilities (one for each class) in each leaf. Such a probabilistic prediction can be converted into a set of labels by applying a threshold t: all labels that are predicted with probability $\geq t$ are in the predicted set. Clus will output for each value in the set a tree in which the predicted label sets are constructed with this particular threshold. So, if `ClassificationThreshold` is set to `[0.5, 0.75, 0.80, 0.90, 0.95]`, the output file will contain 5 trees corresponding to the thresholds 0.5, 0.75, 0.80, 0.90 and 0.95.
- `RecallValues`:
  - *possible values*: a real number from the interval $[0, 1]$ or a list of such values
  - *default*: `None`
  - *description* : For each value, Clus will output the average of the precisions over all class-wise precision-recall curves that correspond to the particular recall value in the output file.
- `EvalClasses`:
  - *possible values*: a string (path to the file)
  - *default*: `None`
  - *description* : If this is set to `None`, Clus computes average error measures across all classes in the class hierarchy. Otherwise, then the error measures are only computed with regard to the classes given in the file, specified under this option. Each line of the file contains one label is of form `<root><HSeparator>...<HSeparator><our class>` if the hierarchy is `Tree`, and `<our class>` if the hierarchy is `DAG`.
- `MEstimate`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : if set to `Yes`, Clus will apply an m-estimate in the prediction vector of each leaf. For each leaf and each label, define T = total training examples and P = number of positive training examples. With the m-estimate, instead of predicting P/T for the given label, we predict $(P + pT')/(T + T')$, i.e. we act as if we have seen T' extra ("virtual") examples of which p are positive, where T' and p are parameters. In the Clus implementation, $T' = 1$ and p is the proportion of positive examples in the full training set. So the predictions in the leaf for a given label are interpreted as $(P + p)/(T + 1)$.

```
[Hierarchical]
Type = Tree                          % Tree or DAG hierarchy?
WType = ExpAvgParentWeight           % aggregation of class weights
WParam = 0.75                        % parameter w_0
HSeparator = /                       % separator used in class names
EmptySetIndicator = n                % symbol for empty set
OptimizeErrorMeasure = PooledAUPRC   % FTest optimization strategy
ClassificationThreshold = None       % threshold for "positive"
RecallValues = None                  % where to report precision
EvalClasses = None                   % classes to evaluate
MEstimate = No                       % whether to use m-estimate in the prediction vector
```

Figure 4.2: Settings specific for hierarchical multi-label classification

```
[HMTR]
Type = Tree                          % Tree or DAG hierarchy?
Distance = WeightedEuclidean         % The distance function
Aggregation = SUM                    % The aggregation function
Weight = 0.75                        % The weight parameter w_0
Hierarchy = ALL-ECOG,ALL-ADAS13...   % The HMTR hierarchy (see examples)
```

Figure 4.3: Settings specific for Hierarchical Multi-Target Regression (HMTR)

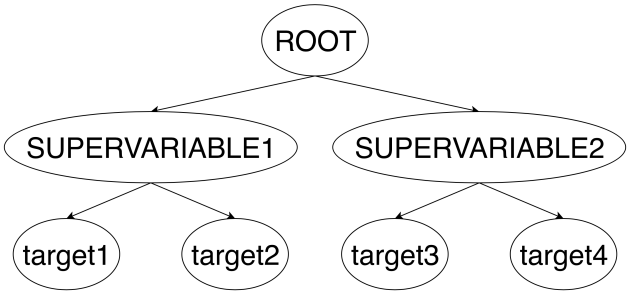

Figure 4.4: An example of a HMTR tree hierarchy.

4.12 Hierarchical Multi-Target Regression (HMTR)

These settings are relevant when using Clus for Hierarchical Multi-Target Regression (HMTR) [18]. Note that HMTR calculates the hierarchy supervariables by itself from the target attributes and it is not required to be done as a preprocessing step. A separate section "HMTR" is implemented for this. Figure 4.3 gives an example of these settings. They are listed below:

- `Type`:
  - *possible values*: an element of {`TREE`, `DAG`}
  - *description* : Indicates whether the class hierarchy is a tree or a directed acyclic graph [27].
- `Distance`:
  - *possible values*: an element of {`WeightedEuclidean`}
  - *description* : The distance function used. Currently, the `WeightedEuclidean` is implemented and recommended.
- `Aggregation`:
  - *possible values*: an element of {`SUM`, `AVG`, `MEDIAN`, `MIN`, `MAX`, `AND`, `OR`, `COUNT`, `VAR`, `STDEV`, `ZERO`, `ONE`}

  - *description* : The aggregation function used for the supervariables in the hierarchy. `SUM` stands for the sum of the values, `AVG` stands for the average value, `MEDIAN` is the median value from the (ordered by size) children, `MIN` and `MAX` are the minimum and maximum values amongst the children nodes, `AND` and `OR` can be used on values that are either 0 or 1 and are the logical *and* and *or* functions, `COUNT` returns the number of children, `VAR` and `STDEV` return the variance and standard deviation of the children node values respectively, and finally `ZERO` and `ONE` functions return a constant value of 0 or 1, respectively.

- `Hierarchy`:
  - *possible values*: text
  - *description* : The hierarchy of the dataset in the format of pair-wise comma-separated node names, starting from the root downwards. The target attribute names are case-sensitive and must be written exactly like in the dataset. If a node name is not found in the dataset, the algorithm assumes it is a supervariable and adds it as a supervariable target by calculating it using the aggregate function on its' children nodes. The first element is the parent node, and the second is the child node. All of the edges in the hierarchy must be represented. The input ignores white spaces, parentheses and the larger-than sign (" ", "(", ")", ">") therefore both of the following notations are acceptable for the hierarchy depicted in Figure 4.4:

    Example 1: (ROOT -> SUPERVARIABLE1), (ROOT -> SUPERVARIABLE2), (SUPERVARIABLE1 -> target1), (SUPERVARIABLE1 -> target2), (SUPERVARIABLE2 -> target3), (SUPERVARIABLE2 -> target4)

    Example 2: ROOT-SUPERVARIABLE1,ROOT-SUPERVARIABLE2,SUPERVARIABLE1-target1,SUPERVARIABLE1-target2,SUPERVARIABLE2-target3,SUPERVARIABLE2-target4

- `Weight`:
  - *possible values*: a value from the interval $[0, 1]$
  - *description* : The weight, used in the Weighted Euclidean distance. The weight is in the form of $w(n) = w_0^{depth(n)}$, i.e., it decreases exponentially with the depth of the node n in the hierarchy. A good starting point is 0.75 [18].

The output file of the HMTR task differs slightly from the standard output file. First, the hierarchy is drawn in the output file. This can be helpful in order to check if the hierarchy written in the settings file corresponds to the actual hierarchy. And the second change is on the location where the errors are reported. Here we now have two groups of errors: the first one is measured on all of the targets *and* supervariables. The second group of errors is the error on the targets only, excluding the error on the supervariables.

4.13 SemiSupervised

Settings for running Clus in semi-supervised learning mode, relevant when command line argument `-ssl` is used. These go in the separate section "SemiSupervised" in the settings file.

- `SemiSupervisedMethod`:
  - *possible values*: an element of {`PCT`, `SelfTraining`, `SelfTrainingFTF`}
  - *default*: `PCT`
  - *description* : specifies the semi-supervised method used:
    - `PCT`: Semi-supervised predictive clustering trees (SSL-PCTs) that use both descriptive and target attributes to evaluate the splits (if `-forest` is used, ensemble of SSL-PCTs will be built). For more details see [16, 15]
    - `SelfTraining`: The self-training method that "wraps" around ensembles of PCTs (`-forest` needs to be used). It iteratively uses its own most reliable predictions in the learning proces. Note, when this method is used, the Default model in .out is supervised random forest. For more details see [14].

    * `SelfTrainingFTF`: Self-training that operates without reliability score, iterates until distances between predictions of two consecutive iterations are smaller than predefined threshold. Note, when this method is used, the Default model in .out is supervised random forest. Implemented on the basis of [9].

- `StoppingCriteria`:
  - *possible values*: an element of {`NoneAdded`, `Iterations`, `Airbag`}
  - *default*: `NoneAdded`
  - *description* : Stopping criteria for the `SelfTraining` method:
    * `NoneAdded`: Stops if no example was added to the training set in the current iteration, i.e., if no unlabeled example met the criteria speficied with `UnlabeledCriteria`.
    * `Iterations`: Stops after the predefined number of iterations with the `Iterations` setting. Also applies for `SelfTrainingFTF`.
    * `Airbag`: "Smart" stopping criteria proposed in [13]. Monitors the out-of-bag error, and stops learning if performance degradation is detected.

- `UnlabeledCriteria`:
  - *possible values*: an element of {`Threshold`, `KMostConfident`, `KPercentageMostConfident`, `KPercentageMostAverage`, `AutomaticOOB`, `AutomaticOOBInitial`, `ExhaustiveSearch`}
  - *default*: `Threshold`
  - *description* : Criteria used by the `SelfTraining` method to select the predictions on unlabeled data that will be added to the training set.
    * `Threshold`: All of the unlabeled instances with confidence of prediction greater than `Threshold` will be added to the training set.
    * `KMostConfident`: `K` unlabeled instances with the most confident predictions will be added to the training set.
    * `KPercentageMostConfident`: `K` percentage of unlabeled instances with the most confident predictions will be added to the training set.
    * `KPercentageMostAverage`: The threshold is set to the average of the reliability scores of the `K` percentage of the most reliable examples (the threshold is set only once after the initial iteration).
    * `AutomaticOOB`: The optimal threshold will be automatically selected on the basis of reliability scores of out-of-bag labeled examples, for more details see [14].
    * `AutomaticOOBInitial`: Similar as `AutomaticOOB`, however, the threshold will be selected only after the initial iteration and used throughout next iterations.
    * `ExhaustiveSearch`: At each iteration, the optimal threshold will be selected from the list specified in `ExhaustiveSearchThresholds`, on the basis of out-of-bag error on labeled examples. Beware, this is computationally expensive.

- `ConfidenceThreshold`:
  - *possible values*: a real number from the interval $[0, 1]$
  - *default*: `0.8`
  - *description* : The threshold for reliability scores, applicable if `UnlabeledCriteria` is set to `Threshold` is used.

- `ConfidenceMeasure`:
  - *possible values*: an element of {`Variance`, `ClassesProbabilities`, `RForestProximities`, `RandomUniform`, `RandomGaussian`, `Oracle`}
  - *default*: `Variance` if the task is (multi-target) regression, `ClassesProbabilities` if the task is (hierarchical multi-label) classification.
  - *description* : Confidence (i.e., reliability) score used in `SelfTraining`.
    * `Variance`: Reliability is inversely proportional to the standard deviation of the votes of an ensemble, i.e., smaller deviation, greater reliability.

* `ClassesProbabilities`: Used for (hierarchical) multi-label classification. Reliability is proportional to empirical probabilites, i.e., proportion of trees in ensemble that voted for a given class.
* `RForestProximities`: Reliability scores are calculated by using estimation of an error of unlabeled examples via out-of-bag error of labeled examples in their random Forest proximity (see [14]).
* `RandomUniform`: Unlabeled examples to be added to the training set are randomly selected, where reliability scores are random numbers uniformly distributed in $[0, 1]$.
* `RandomGaussian`: Unlabeled examples to be added to the training set are randomly selected, where reliability scores are random numbers normally distributed in $[0, 1]$.
* `Oracle`: Actual errors on unlabeled examples are used to calculated to establish reliability scores. This is not attainable in practice (with real unlabeled data), but can be used to gain some insight into the algorithm. To use this score, unlabeled examples in `UnlabeledData` need to be provided with labels.

- `Iterations`:
  - *possible values*: a positive integer
  - *default*: `10`
  - *description* : The number of iterations for `SelfTraining` or `SelfTrainingFTF` methods.
- `K`:
  - *possible values*: a positive integer
  - *default*: `5`
  - *description* : K parameter for `KMostConfident`, `KPercentageMostConfident` and `KPercentageMostAverage`.
- `UnlabeledData`:
  - *possible values*: a string (path to the file)
  - *default*: the empty string
  - *description* : the name of the file that contains unlabeled data. This is optional, unlabeled data can be provided directly in the training set (with missing values for target attributes).
- `PercentageLabeled`:
  - *possible values*: an integer r from the interval $[0, 100]$
  - *default*: `5`
  - *description* : If unlabeled data is not given with `UnlabeledData` or given directly in the training set, then unlabeled data are randomly selected from the training set, where r is the ratio of labeled examples. Default if 5, which means that 5% of the data will be selected as labeled data, and the rest 95% will be used as unlabeled data.
- `UseWeights`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : if set to `Yes`, unlabeled examples that are added to the training set when `SelfTraining` is used will be given weights that correspond to their reliability scores.
- `AirbagTrials`:
  - *possible values*: a nonnegative integer
  - *default*: `0`
  - *description* : The `Airbag` stops `SelfTraining` when degradation of performance according to out-of-bag error is detected. This setting specifies the number of times it is allowed for the model's out-of-bag error to be worse than that of the model trained in previous iteration.

- `ExhaustiveSearchThresholds`:
  - *possible values*: a list of reals from the interval $[0, 1]$.
  - *default*: `[0.5,0.6,0.7,0.8,0.9,0.99]`
  - *description* : Candidate thresholds that are considered if `ExhaustiveSearch` is used.
- `OOBErrorCalculation`:
  - *possible values*: {`LabeledOnly`, `AllData`}
  - *default*: `LabeledOnly`
  - *description* : specifies the data which out of beg error is calculated for in `SelfTraining`.
- `Normalization`:
  - *possible values*: {`MinMaxNormalization`, `Ranking`, `Standardization`, `NoNormalization`}
  - *default*: `MinMaxNormalization`
  - *description* : Normalization of the per-target reliability scores, performed prior to aggregation. An element of . Default is `MinMaxNormalization`.
    - `MinMaxNormalization`: Per-target reliability scores are normalized to $[0, 1]$ interval, where the least reliable score gets 0, and the most reliable score gets 1.
    - `Ranking`: Ranks per-target scores according to their reliability, can be useful if per-target scores have very skewed distribution.
    - `Standardization`: Per-target scores are standardized to 0.5 mean and 0.125 standard deviation (ensures that 99.98% of the scores are in $[0, 1]$ interval).
    - `NoNormalization`: Normalization of per-target scores is not performed.
- `Aggregation`:
  - *possible values*: an element of {`Average`, `Minimum`, `Maximum`}
  - *default*: `Average`
  - *description* : Aggregation of the per-target reliability scores:
    - `Average`: Reliability score is calculated as an average of per-target reliability scores.
    - `Minimum`: An example's prediction is considered as reliable as its least reliable component.
    - `Maximum`: An example's prediction is considered as reliable as its most reliable component.
- `CalibrateHmcThreshold`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* : If set to Yes, the threshold is calibrated such that the difference between label cardinality of labeled examples and predicted unlabeled examples is minimal. Applies if `SelfTraining` is used for hierarchical multi-label classification.
- `PruningWhenTuning`:
  - *possible values*: an element of {`Yes`, `No`}
  - *default*: `No`
  - *description* :If set to Yes, the trees will be pruned while optimizing the w parameter of SSL-PCTs. Should be turned on if pruned trees are of interest.
- `InternalFolds`:
  - *possible values*: an integer, greater than 2
  - *default*: `5`
  - *description* : The number of fold for internal cross-validation, applies if `PossibleWeights` is a vector of values. Default is 5.

- `WeightScoresFile`:
  - *possible values*: a string that describes path to the file or `NO`
  - *default*: `NO`
  - *description* : the name of the file where predictive performance for each candidate w will be written (applies if `PossibleWeights` is a vector of values). By default, the file will not be written.
- `PossibleWeights`:
  - *possible values*: a real number from the interval $[0, 1]$ or a list of such numbers
  - *default*: `[0.0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]`
  - *description* : Specifies the w parameter for SSL-PCTs, i.e., the trade-off between target and descriptive spaces, where $w = 1$ means that supervisd learning will be performed, while $w = 0$ means that unsupervised learning will be performed. If vector is provided, each candidate w will be evaluated via internal cross validation on the labeled part of the training set (each run of internal cross validation will use available unlabeled data), and the final model will be built with the best w.

4.14 Relief

Note that one must ese `-relief` option to run the Relief algorithm.

- `Neighbours`:
  - *possible values*: integer between 1 and the number of training instances, or a list of such integers
  - *default*: `10`
  - *description* : This is the number of neighbours in the Relief algorithms. A warning is given when this falls out of bounds. In that case, we compute a ranking with the default value for this option. The same goes for every element of the list. In that case, the computational complexity of the algorithm approximately equals the complexity of Relief with the maximal number of neighbours.
- `Iterations`:
  - *possible values*: integer between 1 and the number of instances
  - *default*: `0.0`
  - *description* : This is the number of iterations in the Relief algorithms. If it is given as a proportion of all training instances, we convert it to the absolute value by rounding to the nearest integer. If only the value 1 (or 1.0) is given, we assume this means 100% in both cases. A warning is given when this falls out of bounds. In that case, we compute a ranking with the default value for this option. The same goes for every element of the list. In that case, the computational complexity of the algorithm approximately equals the complexity of Relief with the maximal number of iterations.
- `WeightNeighbours`:
  - *possible values*: `Yes` or `No`.
  - *default*: `No`
  - *description* : Boolean that determines whether we weight the influence of the neighbours. If set to `No`, the value specified under the option `WeightingSigma` has no influence.
- `WeightingSigma`:
  - *possible values*: any non-negative real number
  - *default*: `0.5`
  - *description* : determines how quickly the influence of the neighbours decreases with their position i (nearest, 2nd nearest etc.). The weights of the neighbours are proportional to $\exp(-(\sigma i)^2)$, so the value 0.0 corresponds to the case when `WeightNeighbours` is set to `No`.

4.15 Option trees

Settings for using CLUS to build Option Predictive Clustering Trees, triggered by using the `-option` command line option. These go in the separate section "OptionTree" in the settings file.

- `DecayFactor`:
  - *possible values*: A number from the $(0, 1]$ interval.
  - *default*: `0.9`
  - *description* : How quickly the tolerance for introducing option nodes decays with the depth of the node. Every level the `Epsilon` parameter is multiplied by this value. For example, suppose that `Epsilon` is 0.1 and `DecayFactor` is 0.5. At the root of the tree, options that have the heuristic score at least 90% as good as the optimal split would be considered. At the next level, the score would need to be 95% as good, then 97.5%, and so on.
- `Epsilon`:
  - *possible values*: A number from the $[0, 1]$ interval.
  - *default*: `0.1`
  - *description* : How close to the optimal heuristic value a split must be to be considered in an option node. The default value 0.1 means that the heuristic value must be at least 90% as good as the optimal heuristic value. Every level it is multiplied by the `DecayFactor`. For example, suppose that `Epsilon` is 0.1 and `DecayFactor` is 0.5. At the root of the tree, options that have the heuristic score at least 90% as good as the optimal split would be considered. At the next level, the score would need to be 95% as good, then 97.5%, and so on.
- `MaxNumberOfOptionsPerNode`:
  - *possible values*: A positive integer.
  - *default*: `5`
  - *description* : Determines the maximum number of options included in an option node. If there are more options satisfying the heuristic criterion, only those ranked up to this value (by their heuristic scores) will be selected.
- `MaxDepthOfOptionNode`:
  - *possible values*: A positive integer.
  - *default*: `3`
  - *description* : Determines the number of levels at which an option node can be introduced. The default value 3 means that option nodes can only be introduced at the top 3 levels of the tree.

4.16 kNN

- `K`:
  - *possible values*: a positive integer or a list thereof, e.g., `10` or `[1,5,10]`.
  - *default*: `[1,3]`
  - *description* : the number(s) of neighbors in the kNN algorithm. Computational complexity of the algorithm equals the complexity for the maximal number of neighbors.
- `Distance`:
  - *possible values*: an element of {`Euclidean`, `Chebyshev`, `Manhattan`}
  - *default*: `Euclidean`
  - *description* : the distance measure which is used when finding the nearest neighbours. This option (together with the option `AttributeWeighting`) defines the distance measures used in nearest neighbour search:

* `Euclidean`: $d(a,b) = \sqrt{\sum_i w_i d_i(a_i,b_i)^2}$,
* `Manhattan`: $d(a,b) = \sum_i w_i d_i(a_i,b_i)$,
* `Chebyshev`: $d(a,b) = \max_i w_i d_i(a_i,b_i)$,

where a and b are vectors of descriptive attributes, the weights w_i are defined by the option `AttributeWeighting`, and d_i is the distance in the space of the i-th attribute.

If the i-th attribute is numeric, then $d_i(a_i,b_i)$ is proportional to $|a_i - b_i|$ and linearly normalized to the interval $[0,1]$. If the i-th attribute is nominal, then $d_i(a_i,b_i) = 0$ if $a_i = b_i$ and $d_i(a_i,b_i) = 1$ otherwise.

- `SearchMethod`:
  - *possible values*: an element of {`BruteForce`, `VPTree`, `KDTree`, `Oracle`}
  - *default*: `BrutForce`
  - *description* : the method that is used when finding the nearest neighbors. If the number of attributes is high, `BruteForce` may be the fastest, since the structures such as VP- and KD-trees suffer from the *curse of dimensionality*.
- `DistanceWeighting`:
  - *possible values*: an element of {`Constant`, `OneOverD`, `OneMinusD`}
  - *default*: `constant`
  - *description* : When making predictions for a new example, this option is used to weight the contributions of the neighbors. The option `Constant` is equivalent to no weighting, whereas the options `OneMinusD` and `OneOverD` respectively correspond to the weights that equal $1 - d$(example, neighbour) and $1/(\varepsilon + d$(example, neighbour)), where $\varepsilon = 0.001$ prevents zero-division problems.
- `AttributeWeighting`:
  - *possible values*: a string that equals `none`, or starts with `RF`, or represents a list of double values, e.g., `none` or `RF` or `RF,100` or `[2.71828,3.14,1.618,0.0]`.
  - *default*: `none`
  - *description* : From the value of these option, the dimensional weights w_i are computed, i.e., the weights that control the contribution of each dimension (attribute). The weights are defined as follows:
    * `none`: $w_i = 1$
    * `RF,<number of trees>`: w_i equals the importance of the i-th attribute as computed from `RForest` ranking from the ensemble of the size `<number of trees>`.
    * `RF`: equivalent to `RF,100`
    * `<explicitly given list of weights>`: w_i equals the i-th element of the list.

Chapter 5

Command Line Parameters

CLUS is run from the command line. It takes a number of command line parameters that affect its behavior.

- `-xval` : in addition to learning a single model from the whole input dataset, perform a cross-validation. The XVal setting (page 14) determines the number of folds; the RandomSeed setting (page 12) initializes the random generator that determines the folds.
- `-fold` N: run only fold N of the cross-validation.
- `-rules` : construct predictive clustering rules (PCRs) instead of predictive clustering trees (PCTs).
- `-forest` : construct an ensemble instead of a single tree [12].
- `-beam` : construct a tree using beam search [11].
- `-sit` : run Empirical Asymmetric Selective Transfer [20].
- `-silent` : run CLUS with reduced screen output.
- `-info` : gives information and summary statistics about the dataset.
- `-ssl` : run CLUS in semi-supervised learning mode.
- `-relief`: run the generalized Relief algorithm
- `-knn`: run the nearest neighbors algorithm

Chapter 6

Output Files

6.1 Output File

When CLUS is finished, it writes the results of the run into an output file with the name *filename*.out. An example of such an output file is shown in Figures 6.1 to 6.4.

6.1.1 Used Settings

The first part of *filename*.out (shown in Figures 6.1 and 6.2) contains the values of the settings that were used for this run of CLUS, in the format used by the settings file. This part can be copied and pasted to *filename*.s and modified for subsequent runs.

6.1.2 Evaluation Statistics

The next part contains statistics about the results of this CLUS run.

Summary statistics about the running time of CLUS and about the size of the resulting models are given. Next, information on the models' predictive performance on the training set ("training set error") is given, as well as an estimate of its predictive performance on unseen examples ("test set error"), when available (this is the case if a cross-validation or an evaluation on a separate test set was performed).

Typically three models are reported: a "default" model consisting of a tree of size zero, which can be used as a reference point (for instance, its predictive accuracy equals that obtained by always predicting the majority class); an unpruned ("original") tree, and a pruned tree.

For classification trees the information given for each model by default includes a contingency table, and (computed from that) the accuracy and Cramer's correlation coefficient.

For regression trees, this information includes the mean absolute error (MAE), mean squared error (MSE), root mean squared error (RMSE), weighted RMSE, the Pearson correlation coefficient r and it square. In the weighted RMSE, the weight of a given attribute A is its normalization weight, which is $\frac{1}{\sqrt{\mathrm{Var}(A)}}$, with $\mathrm{Var}(A)$ equal to A's variance in the input data.

6.1.3 The Models

The output file contains the learned models, represented as decision trees. The level of detail in which the models are shown is influenced by certain settings.

6.2 Fimp File

If at least one feature ranking was computed, the corresponding results are given in at least one .fimp file. The file contains basic information about the parameters that influence the ranking and the table with the feature relevances.

Currently, we have three ensemble feature ranking methods:

- Symbolic (random forests, bagging and ensembles of extra trees),
- Genie3 (random forests, bagging and ensembles of extra trees),

- RForest (random forests and bagging),

and we have generalised Relief. All rankings can be computed in the case of classification, (multi-target) regression, (hierarchical) multi-label classification etc.

6.2.1 Nomenclature

If the ranking algorithm is one of the ensemble feature rankings, we produce one fimp for every number of iterations, specified under the `[Ensemble]/Iterations` field. For a given number of iterations, i.e., number of trees, *filename*`Trees`*<number of trees>*`.fimp` is created. In the case of Relief, only one fimp is produced, namely *filename*`.fimp`.

6.2.2 Structure

Every fimp contains

- header: contains basic description of the ranking(s) computed, e.g., the number of trees (for the ensemble methods) or number of the neighbours (for Relief),
- table with feature importances: each row (except for the table's header) corresponds to one attribute, which is described by
  - its index in the dataset,
  - its name,
  - list of ranks in the computed feature rankings,
  - list of importances in the computed feature rankings.

The columns in the table's header that belongs to feature ranks and importances contain the names of the target attributes, possibly with some additional information:

- Ensemble methods: no additional information, except for the multi-label classification where the measure that was used in feature ranking computation is specified.
- Relief: The number of iterations and neighbors used in the computation.

The first ranking, i.e., `overAll` ranking is the ranking that is computed with respect to all targets. If `FeatureRankingPerTarget` is set to `Yes` and there is more than one target attribute, the additional feature rankings are computed. In the exemplary files (see Figs. 6.5 and 6.6), we have two targets, namely `T1` and `T2`.

Following the example of the attribute `A1` in Fig. 6.5, here is how we read the fimp table. `A1` has index 1 in the dataset. It is ranked 4th in the overall ranking and the ranking with respect to the target `T1` (has the lowest importances in these two cases), and is ranked second in the ranking with respect to the target `T2`. Its importances in the overall, T1 and T2 ranking are 0.5, 0.6 and 1.2 respectively.

If only one ranking is computed, depending on the `[Ensemble]/SortRankingByRelevance` setting, the table is sorted increasingly by dataset index or decreasingly by the feature importances.

```
Clus run "weather"
******************

Date: 1/10/10 4:37 PM
File: weather.out
Attributes: 4 (input: 2, output: 2)
Missing values: No

[General]
Verbose = 1
Compatibility = Latest
RandomSeed = 0
ResourceInfoLoaded = No

[Data]
File = weather.arff
TestSet = None
PruneSet = None
XVal = 10
RemoveMissingTarget = No
NormalizeData = None

[Attributes]
Target = 3-4
Clustering = 3-4
Descriptive = 1-2
Key = None
Disable = None
Weights = Normalize
ClusteringWeights = 1.0
ReduceMemoryNominalAttrs = No

[Constraints]
Syntactic = None
MaxSize = Infinity
MaxError = 0.0
MaxDepth = Infinity

[Output]
ShowModels = {Default, Pruned, Others}
TrainErrors = Yes
ValidErrors = Yes
TestErrors = Yes
AllFoldModels = Yes
AllFoldErrors = No
AllFoldDatasets = No
UnknownFrequency = No
BranchFrequency = No
ShowInfo = {Count}
PrintModelAndExamples = No
WriteErrorFile = No
WritePredictions = {None}
WriteModelIDFiles = No
WriteCurves = No
OutputPythonModel = No
OutputDatabaseQueries = No
```

Figure 6.1: Example output file (part 1, settings).

```
[Nominal]
MEstimate = 1.0

[Model]
MinimalWeight = 2.0
MinimalNumberExamples = 0
MinimalKnownWeight = 0.0
ParamTuneNumberFolds = 10
ClassWeights = 0.0
NominalSubsetTests = Yes

[Tree]
Heuristic = VarianceReduction
PruningMethod = M5
M5PruningMult = 2.0
FTest = 1.0
BinarySplit = Yes
ConvertToRules = No
AlternativeSplits = No
Optimize = {}
MSENominal = No
```

Figure 6.2: Example output file (part 2, settings (ctd.)).

```
Run: 01
*******

Statistics
----------

FTValue (FTest): 1.0
Induction Time: 0.018 sec
Pruning Time: 0.001 sec
Model information
     Default: Nodes = 1 (Leaves: 1)
     Original: Nodes = 7 (Leaves: 4)
     Pruned: Nodes = 3 (Leaves: 2)

Training error
--------------

Number of examples: 8
Mean absolute error (MAE)
   Default        : [7.125,14.75]: 10.9375
   Original       : [2.125,2.75]: 2.4375
   Pruned         : [4.125,7.125]: 5.625
Mean squared error (MSE)
   Default        : [76.8594,275.4375]: 176.1484
   Original       : [6.5625,7.75]: 7.1562
   Pruned         : [19.4375,71.25]: 45.3438
Root mean squared error (RMSE)
   Default        : [8.7669,16.5963]: 13.2721
   Original       : [2.5617,2.7839]: 2.6751
   Pruned         : [4.4088,8.441]: 6.7338
Weighted root mean squared error (RMSE) (Weights [0.013,0.004])
   Default        : [1,1]: 1
   Original       : [0.2922,0.1677]: 0.2382
   Pruned         : [0.5029,0.5086]: 0.5058
Pearson correlation coefficient
   Default        : [?,?], Avg r^2: ?
   Original       : [0.9564,0.9858], Avg r^2: 0.9432
   Pruned         : [0.8644,0.861], Avg r^2: 0.7442
```

Figure 6.3: Example output file (part 3, statistics).

```
Default Model
*************

[18.875,77.25]: 8

Original Model
**************

outlook = sunny
+--yes: [32,52.5]: 2
+--no:  outlook = rainy
        +--yes: windy = yes
        |       +--yes: [9,92.5]: 2
        |       +--no:  [19,91.5]: 2
        +--no:  [15.5,72.5]: 2

Pruned Model
************

outlook = sunny
+--yes: [32,52.5]: 2
+--no:  [14.5,85.5]: 6
```

Figure 6.4: Example output file (part 4, learned models).

```
Ensemble method: Bagging
Ranking method: GENIE3
Ensemble size: 10
attributeDatasetIndex attributeName [overAllRank, T2Rank, T1Rank] [overAll, T2, T1]
-------------------------------------------------------------------------------
1 A1 [4, 4, 2] [0.5, 0.6, 1.2]
2 A2 [3, 3, 3] [0.7, 0.8, 0.8]
3 A3 [1, 1, 1] [5.3, 4.5, 4.6]
4 A4 [2, 2, 4] [2.2  2.1, 0.0]
```

Figure 6.5: An example of ensemble `.fimp` file

```
Ranking method: Relief with all combinations of
numbers of neighbours: [15]
numbers of iterations: [300]
attributeDatasetIndex attributeName <rank descriptions> <importances descriptions>
-------------------------------------------------------------------------------
1 A1 [4, 4, 2, 2] [0.5, 0.6, 1.1, 1.2]
2 A2 [3, 3, 3, 3] [0.7, 0.8, 0.8, 0.8]
3 A3 [1, 1, 1, 1] [5.3, 4.5, 4.5, 4.6]
4 A4 [2, 2, 4, 4] [2.2  2.1  0.1, 0.0]
```

Figure 6.6: An example of Relief `.fimp` file, where `<rank descriptions>` are given as `[overallIter300Neigh15Rank, T1Iter300Neigh15Rank, T2Iter300Neigh15Rank]` and `<importances descriptions>` are given as `[overallIter300Neigh15, T1Iter300Neigh15, T2Iter300Neigh15]`.

Chapter 7

Developer Documentation - this is not up to date!

7.1 Compiling Clus

Note: The CLUS download comes with a pre-compiled version of CLUS stored in the file Clus.jar. So, if you just want to run CLUS as it is on a data set, then you do not need to compile CLUS. You can run it by following the instructions in Section 2.1. On the other hand, if you wish to modify the source code of CLUS, or if you are using the SVN version, then you will need to compile the source code of CLUS. This can be done using the commands below or using the Eclipse IDE as pointed out in the next section.

The SVN developers' version of CLUS is available at `http://sourceforge.net/projects/clus/`.

(Windows)

```
cd C:\Clus\src
javac -d "bin" -cp ".;jars\commons-math-1.0.jar;jars\jgap.jar" clus/Clus.java
```

(Unix)

```
cd /home/john/Clus
javac -d "bin" -cp ".:jars/commons-math-1.0.jar:jars/jgap.jar" clus/Clus.java
```

This will compile CLUS and write the resulting .class files (Java executable byte code) to the "bin" subdirectory. Alternatively, use the "./compile.sh" script provided in the CLUS main directory.

7.2 Compiling Clus with Eclipse

In Eclipse, create a new project for CLUS as follows:

- Choose `File | New | Project`.
- Select "Java Project" in the dialog box.
- In the "New Java Project" dialog box:
  - Enter "Clus" in the field "Project Name".
  - Choose "Create project from existing source" and browse to the location where you unzipped Clus. E.g., `/home/john/Clus` or `C:\Clus`.
  - Click "Next".
  - Select the "Source" tab of the build settings dialog box. Change "Default output folder" (where the class files are generated) to: "Clus/bin".
  - Select the "Libraries" tab of the build settings dialog box. Click "Add external jars" and add in this way these three jars:
    Clus/jars/commons-math-1.0.jar
    Clus/jars/jgap.jar
    Clus/jars/weka.jar

- Click "Finish".

- Select the "Navigator" view (Choose Window — Show View — Navigator)
  - Right click the "Clus" project in this view.
  - Select "Properties" from the context menu.
  - Select the "Java Compiler" tab.
  - Set the "Java Compliance Level" to 5.0.

Now CLUS should be automatically compiled by Eclipse. To run CLUS from Eclipse:

- Set as main class "clus.Clus".
- Set as arguments the name of your settings file (appfile.s).
- Set as working directory, the directory on the file system where your data set is located.

7.3 Running Clus after Compiling the Source Code

These instructions are for running CLUS after you compiled its source code (using the instructions "Compiling Clus" or "Compiling Clus with Eclipse"). To run the pre-compiled version that is available in the file "Clus.jar", see Section 2.1.

(Windows)

```
cd path\to\appfile.s
java -cp "C:\Clus\bin;C:\Clus\jars\commons-math-1.0.jar;C:\Clus\jars\jgap.jar"
  clus.Clus appfile.s
```

(Unix)

```
cd path/to/appfile.s
java -cp "$HOME/Clus/bin:$HOME/Clus/jars/commons-math-1.0.jar:$HOME/Clus/jars/jgap.jar"
  clus.Clus appfile.s
```

Alternatively, use the "./clus.sh" script provided in the CLUS main directory after adjusting the line that defines CLUS_DIR at the top of the script.

7.4 Code Organization

Here we only provide a rough guide to the CLUS code by listing some of the key classes and packages.

`clus/Clus.java` the main class with the main method, which is called when starting CLUS.

`clus/algo` package with learning algorithms, e.g., sub-package `clus/algo/tdidt` includes tree learning algorithm and `clus/algo/rules` includes rule learning algorithm, `clus/algo/split` includes classes used for generating conditions in both trees and rules.

`clus/data` package with sub-packages and classes related to reading and storing of the data.

`clus/error` package where different error estimation measures are defined.

`clus/ext` some extensions of base tree learning methods can be found here, e.g., sub-package `hierarchical` contains extensions needed for hierarchical classification, `ensembles` contains ensembles of trees and `timeseries` contains extensions for predicting time-series data.

`clus/heuristic` contains classes implementing heuristic functions for tree learning (heuristics for rule learning are located in package `clus/algo/rules`).

`clus/main` contains some important support classes such as:

`ClusRun.java`

`ClusStat.java`

`ClusStatManager.java`

`Settings.java` all the CLUS settings are defined here.

`clus/model` classes used for representations of models can be found here, including tests that appear in trees and rules (`clus/model/test`).

`clus/pruning` contains methods for tree pruning.

`clus/statistic` contains classes used for storing and manipulating different information and statistics on data. The key classes are:

`ClusStatistic.java` super class for all statistics used in CLUS.

`ClassificationStat.java` class for storing information on nominal attributes (e.g., counts for each possible nominal value)

`RegressionStat.java` class for storing information on numeric attributes (e.g., sums of values and sums of squared values).

`CombStat.java` class for storing information on nominal and numeric attributes (contains `ClassificationStat` and `RegressionStat` classes).

`clus/tools` contains some support code, e.g., sub-package `optimization` contains optimization procedures used in rule ensemble learning.

`clus/weka` contains classes for interfacing with Weka machine learning package.

Acknowledgments

The research involved in development of this software was supported by the Research Foundation Flanders (FWO-Vlaanderen) and the Slovenian Research Agency. We thank all our collaborators who have contributed to the development of CLUS, and in particular (in inverse alphabetical order) L. Schietgat, I. Slavkov, D. Kocev, V. Gjorgjioski, E. Fromont and T. Aho.

Bibliography

[1] Timo Aho, Bernard Ženko, and Sašo Džeroski. Rule ensembles for multi-target regression. In Wei Wang, Hillol Kargupta, Sanjay Ranka, Philip S. Yu, and Xindong Wu, editors, *Proceedings of Ninth IEEE International Conference on Data Mining (ICDM 2009), December 6-9, 2009, Miami Beach, Florida, USA*, pages 21–30. IEEE Computer Society, 2009.

[2] H. Blockeel, S. Džeroski, and J. Grbović. Simultaneous prediction of multiple chemical parameters of river water quality with Tilde. In *Proceedings of the 3rd European Conference on Principles of Data Mining and Knowledge Discovery*, volume 1704 of *Lecture Notes in Artificial Intelligence*, pages 32–40. Springer, 1999.

[3] H. Blockeel, L. De Raedt, and J. Ramon. Top-down induction of clustering trees. In *Proceedings of the 15th International Conference on Machine Learning*, pages 55–63, 1998.

[4] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[5] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Wadsworth International Group, Belmont, CA, USA, 1984.

[6] R. Caruana. Multitask learning. *Machine Learning*, 28:41–75, 1997.

[7] Bojan Cestnik. Estimating probabilities: A crucial task in machine learning. In L. Aiello, editor, *Proceedings of the Ninth European Conference on Artificial Intelligence (ECAI 90)*, pages 147–149, London, UK/Boston, MA, USA, 1990. Pitman.

[8] P. Clark and R. Boswell. Rule induction with CN2: Some recent improvements. In Yves Kodratoff, editor, *Proceedings of the Fifth European Working Session on Learning*, volume 482 of *Lecture Notes in Artificial Intelligence*, pages 151–163. Springer-Verlag, 1991.

[9] Mark Culp and George Michailidis. An iterative algorithm for extending learners to a semi-supervised setting. *Journal of Computational and Graphical Statistics*, 17(3):545–571, 2003.

[10] M. Garofalakis, D. Hyun, R. Rastogi, and K. Shim. Building decision trees with constraints. *Data Mining and Knowledge Discovery*, 7(2):187–214, 2003.

[11] D. Kocev, S. Džeroski, and J. Struyf. Beam search induction and similarity constraints for predictive clustering trees. In *5th Int'l Workshop on Knowledge Discovery in Inductive Databases: Revised Selected and Invited Papers*, 2007. To appear.

[12] D. Kocev, C. Vens, J. Struyf, and S. Džeroski. Ensembles of multi-objective decision trees. In *Proceedings of the 18th European Conference on Machine Learning*, pages 624–631. Springer, 2007.

[13] Christian Leistner, Amir Saffari, Jakob Santner, and Horst Bischof. Semi-supervised random forests. In *Proceedings of the 12th International Conference on Computer Vision*, pages 506–513. IEEE, 2009.

[14] Jurica Levatić, Michelangelo Ceci, Dragi Kocev, and Sašo Džeroski. Self-training for multi-target regression with tree ensembles. *Knowledge-Based Systems*, 123:41–60, 2017.

[15] Jurica Levatić, Michelangelo Ceci, Dragi Kocev, and Sašo Džeroski. Semi-supervised classification trees. *Journal of Intelligent Information Systems*, 49(3):461–486, 2017.

[16] Jurica Levatić, Dragi Kocev, Michelangelo Ceci, and Sašo Džeroski. Semi-supervised trees for multi-target regression. *Information Sciences*, 450:109–127, 2018.

[17] Ryszard S. Michalski. On the quasi-minimal solution of the general covering problem. In *Proceedings of the Fifth International Symposium on Information Processing (FCIP 69)*, volume A3, Switching Circuits, pages 125–128, Bled, Yugoslavia, 1969.

[18] Vanja Mileski, Sašo Džeroski, and Dragi Kocev. Predictive clustering trees for hierarchical multi-target regression. In *Proceedings of the 16th International Symposium on Intelligent Data Analysis XVI LNCS*, page In Press. Springer-Verlag, 2017.

[19] Matej Petković, Dragi Kocev, and Sašo Džeroski. Feature ranking for multi-target regression with tree ensemble methods. In *Lecture Notes in Computer Science*, volume 10558, 2017.

[20] B. Piccart, J. Struyf, and H. Blockeel. Empirical asymmetric selective transfer in multi-objective decision trees. In *Proceedings of the 11th International Conference on Discovery Science*, volume 5255 of *Lecture Notes in Artificial Intelligence*, pages 64–75, 2008.

[21] J. Ross Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[22] J. Ross Quinlan. Learning with continuous classes. In A. Adams and L. Sterling, editors, *Proceedings of the Fifth Australian Joint Conference on Artificial Intelligence, Hobart, Australia, November 16-18, 1992*, pages 343–348, Singapore, 1992. World Scientific.

[23] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Francisco, CA, USA, 1993.

[24] Jun Sese and Shinichi Morishita. Itemset classified clustering. In Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi, editors, *Knowledge Discovery in Databases: PKDD 2004, Proceedings of the Eighth European Conference on Principles and Practice of Knowledge Discovery in Databases, Pisa, Italy, September 20-24, 2004*, Lecture Notes in Computer Science, pages 398–409, Berlin, Germany, 2004. Springer.

[25] J. Struyf and S. Džeroski. Constraint based induction of multi-objective regression trees. In *Knowledge Discovery in Inductive Databases, 4th International Workshop, KDID'05, Revised, Selected and Invited Papers*, volume 3933 of *Lecture Notes in Computer Science*, pages 222–233, 2006.

[26] Celine Vens and Fabrizio Costa. Random forest based feature induction. In *Proceedings of IEEE International Conference on Data Mining*, 2011.

[27] Celine Vens, Jan Struyf, Leander Schietgat, Sašo Džeroski, and Hendrik Blockeel. Decision trees for hierarchical multi-label classification. *Machine Learning*, 73(2):185–214, 2008.

[28] Bernard Ženko. *Learning predictive clustering rules*. PhD thesis, University of Ljubljana, Faculty of computer and information science, Ljubljana, Slovenia, 2007.