R S D

A System for Relational Subgroup Discovery through First-Order Feature Construction

v1.0 – release candidate version

– User's Manual –

Filip Železný

March 17, 2003

Contents

1 Introduction

RSD is a system for relational subgroup discovery in individual-centered domains, based on principles described in

> N. Lavrač, F. Železný, P. Flach: *Relational subgroup discovery through first-order feature construction*, in: Stan Matwin and Claude Sammut (eds.): *Proceedings of the 12th international conference on inductive logic programming.* 6/2002, Springer-Verlag.

An extended treatment of the theoretical principles is presently being submitted. RSD, however, can be **useful as well in tasks other than subgroup discovery**, as long as they require **to generate a propositionalized representation** of classified relational data achieved **through constructing first-order features**.

RSD is implemented in YAP Prolog. YAP can be obtained at `http://sourceforge.net/projects/yap` in versions for various operating systems. The RSD package can be downloaded from `http://labe.felk.cvut.cz/~zelezny/rsd`. Questions, comments and suggestions should be sent to `zelezny@fel.cvut.cz`. Following are the main advantages of RSD.

- Syntactical feature construction controlled by mode declarations very similar to those used in the popular systems Progol and Aleph.
- The user can specify various constraints (syntactical or data-related) used by RSD's powerful pruning rules towards performance gains.
- Filtering of irrelevant (unneeded) features.
- Generation of propositionalized data representation compatible with the popular systems CN2 and Weka.
- Automated train-test splitting and stratified cross-validation process in RSD's rule induction component.

RSD consists of three Prolog programs. Their respective function is briefly as follows.

`featurize.pl` Using specified *mode-language* declarations, this program identifies all first-order literal conjunctions that by definition form a feature (1. all output variables[1] must also appear as input variables, 2. feature must not be decomposable into two separate features), and at the same time comply to user-defined syntactical constraints. Such features do not contain any constants and the task is completed independently of the input data.

[1] I will clarify in a while what *input* and *output* variables mean here.

`process.pl` Here the features get confronted with the input data. This component extends the feature set by variable instantiations; certain features are copied several times with some variables (specified by the user) substituted by constants detected by inspecting the input data. During this process, some irrelevant features are identified and eliminated. The program also generates propositionalized representations of the input data using the generated feature set, i.e., it creates a relational table consisting of binary attributes corresponding to the truth values of features with respect to instances of data. The resulting table can be produced in a plain text format or in the format acceptable by CN2, Weka or the following component.

`rules.pl` Finds interesting subgroups in the propositionalized data set. An interesting subgroup is a subset of instances which is sufficiently large and has a statistical distribution of target attribute values significantly different than that found in the entire data set. By a simple configuration of settings, this program can also be forged to behave as a standard inducer of rule-based predictive models, such as CN2.

Mode declarations, background knowledge (i.e. all data excluding the main table containing the target attribute) and all parameter/constraint settings are specified in the `<filename>.b` file by means of Prolog facts/clauses (settings can also be applied on-line when running the respective program components). The `.b` file is loaded by all three program components.

The relation, whose instances are classified, is represented by

- either the `<filename>.f` and `<filename>.n` files, containing positive and negative (respectively) instances in the form of unary Prolog facts of the target predicate. This variant is suitable only for two-class data.

- or the `<filename>.pl` file containing binary Prolog facts of the target predicate, where the first argument of each fact is assumed to define the class of the instance. This variant applies to general multi-class problems.

The files relevant to one of the two above itemized options are loaded only by the `process.pl` component. It detects automatically which file(s) is (are) available in the current (or specified) directory and consequently which option of those above applies.

Note that the fact that the main relation predicate must be unary (or binary if the first argument is the class value) means that the predicate serves merely to address instances by their name (or by an identifying constant, in general), whereas their structural description is to be found in the background knowledge. If, however, the description is simple, the user may as well incorporate this description into a compound term, and use such terms as the main predicate argument instead of the name. This is the case of the East-West Trains example presented in the following Quick Start section.

Let us digress a bit. If you find it inconvenient that RSD wants you to provide data in Prolog, you may want to try the freely available data convertor Sumatra TT, capable of connecting to data sources of an extensive variety of types (exceeding the command of a mortal person) and producing a Prolog form thereof. The system is described in the award-winning paper

> P. Aubrecht, F. Železný, P. Mikšovský, O. Štěpánková: *SumatraTT: Towards a Universal Data Preprocessor.* In: *Proceedings of the 16th European Meeting on Cybernetics and System Research*, vol. 2, p. 818-823. Vienna, Austria, 4/2002, Austrian Society for Cybernetics Studies.

This software comes with a fancy graphical interface and can be downloaded from `http://krizik.felk.cvut.cz:8080/SumatraReg/index.html`.

Some additional files are engaged in a typical process of using RSD. The `featuri ze.pl` program component produces a set of features without constants in the file `<filename>_frs_noinst.pl` and further a 'symbolic' representation of these features in the file `<filename>_frs.smb` (this file is not significant to the user). The component `process.pl` loads and uses both of these files to produce the 'instantiated' and filtered feature set `<filename>_frs.pl` and also the propositional representation of a chosen form; for example a `<filename>.cov` file is created to feed the `rules.pl` program, the rule induction component of RSD.

Figure 1 shows the interconnection of individual components and files. Note that the program CN2-SD is a variant of the CN2 program suited for subgroup discovery, described in N. Lavrač et al.: *Rule Induction for Subgroup Discovery with CN2-SD*, IDDM 2002.

2 Quick Start

I assume you decompressed the RSD download package into a chosen directory, where an `rsd` subdirectory is created. By `<rsd>` I denote the complete path to RSD, including the `rsd` directory. Change to `<rsd>/samples/trains`, containing an example data set representing the well-known East-West trains. I also assume the YAP Prolog is in your path of executables. Run YAP, obtaining roughly the following in the console:

```
$ yap <Enter>
[ Restoring file /usr/local/lib/Yap/startup ]
[ YAP version Yap-4.3.23 ]
?-
```

Proceed as follows to consult the `f.pl` file, which is actually a symbolic link to `<rsd>/code/featurize.pl`.

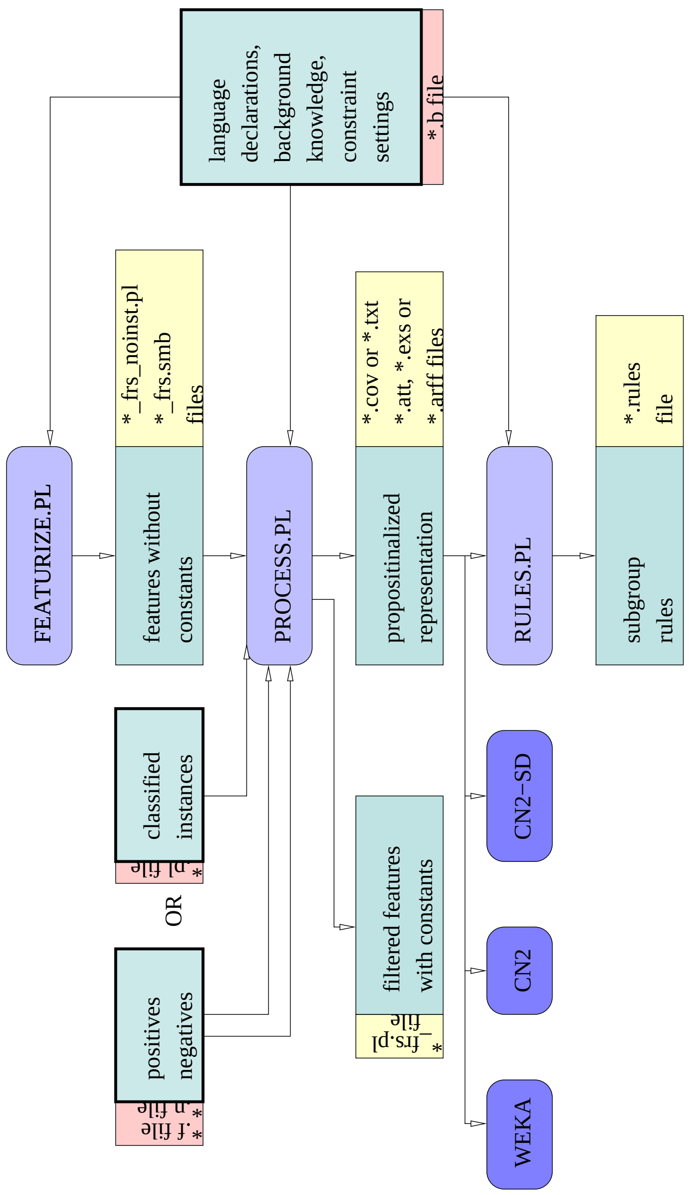
language declarations, background knowledge, constraint settings
*.b file
FEATURIZE.PL
features without constants
*_frs_noinst.pl *_frs.smb files
PROCESS.PL
propositinalized representation
*.cov or *.txt *.att, *.exs or *.arff files
RULES.PL
subgroup rules
*.rules file
classified instances
*.pl file
OR
positives negatives
*.f file *.n file
filtered features with constants
*_frs.pl file
CN2−SD
CN2
WEKA

Figure 1: The cooperation of individual RSD (and external) components through files. Rounded boxes correspond to program codes, other boxes represent data files. Bold-framed boxes denote input files provided by the user.

```
?- [f]. <Enter>
  [ consulting /rsd/trains/f.pl... ]
  [ consulted /rsd/trains/f.pl in module user, 30 msec 51512
  bytes ]

yes
?-
```

Now to read the `trains.b` file present in the current directory, type[2]

```
r(trains). <Enter>.
```

When the file is read, type

```
s. <Enter>
```

to have the program show you the features constructible on the basis of the declarations in the `trains.b` file. The following should appear.

```
f(1,A):-hasCar(A,B),carshape(B,C),instantiate(C).
f(2,A):-hasCar(A,B),carshape(B,C),carlength(B,D),
instantiate(C),instantiate(D).
f(3,A):-hasCar(A,B),carshape(B,C),carlength(B,D),has_sides(B,E),
instantiate(C),instantiate(D),instantiate(E).
f(4,A):-hasCar(A,B),carshape(B,C),carlength(B,D),has_roof(B,E),
instantiate(C),instantiate(D),instantiate(E).
f(5,A):-hasCar(A,B),carshape(B,C),carlength(B,D),has_wheels(B,E),
instantiate(C),instantiate(D),instantiate(E).
f(6,A):-hasCar(A,B),carshape(B,C),carlength(B,D),has_load(B,E),
instantiate(C),instantiate(D),loadshape(E,F),instantiate(F).
```

(...etc.)

Note that the `instantiate/1` literals, which appear in the features, identify variables that will be substituted by constants in a following step described further on. To write the features into the output files (named `trains_frs_noins .pl and trains_frs.smb`), do as follows

```
?- w. <Enter>

0.781 seconds taken to construct features.

yes
?-
```

[2]Note that if a filename was used containing special symbols (unlike here) that disqualify the name from being a Prolog atom, it would have to be enclosed in apostrophes, e.g. `r('trains-version1').` etc.

Now exit YAP by pressing `CTRL+D`, and run it again. This time you invoke `p.pl` linked to `<rsd>/code/process.pl`

```
$ yap <Enter>
[ Restoring file /usr/local/lib/Yap/startup ]
[ YAP version Yap-4.3.23 ]
?- [p]. <Enter>
[ consulting /rsd/trains/p.pl... ]
[ consulted /rsd/trains/p.pl in module user, 40 msec
80232 bytes ]

yes
?-
```

Then again you read the declaration file along with the outputs of the previous step and the classified data, as follows.

```
?- r(trains). <Enter>

20 examples read.
[ reconsulting /rsd/trains/trains.b... ]
[ reconsulted /rsd/trains/trains.b in module user, 10 msec
13220 bytes ]
[ reconsulting /rsd/trains/trains_frs_noinst.pl... ]
[ reconsulted /rsd/trains/trains_frs_noinst.pl in module user,
 20 msec 43880 bytes ]

Congratulations! You succeeded loading all neccessary files.
yes
?-
```

Now again, if you type

```
s. <Enter>
```

the program will show you a set of features. This time this is the set of non-redundant features which contain constants extracted from the data set and comply to constraints related to feature's coverage on the data. You should see the following:

```
f(1,A):-hasCar(A,B),carshape(B,u_shaped).
f(2,A):-hasCar(A,B),carshape(B,bucket).
f(3,A):-hasCar(A,B),carshape(B,hexagon).
f(4,A):-hasCar(A,B),carshape(B,ellipse).
f(5,A):-hasCar(A,B),carshape(B,rectangle),carlength(B,long).
f(6,A):-hasCar(A,B),carshape(B,rectangle),carlength(B,short).
```

(...etc.)

Now write the expanded feature file as illustrated below. The program comes up with a progress meter accompanied with a very funny message, which fortunately appears only once.

```
?- w. <Enter>

Warning: progress meter is approximate and takes no
legal responsibility for accurate estimates.

0%|-----------------------------------------------|100%
  ************************************************
0.341 seconds taken to expand features.
Hmmm, that was a quick one!
yes
```

Finally let us create a propositional representation (named `trains.cov`) acceptable by the RSD rule inducer.

```
?- w(rsd,trains).
0%|-----------------------------------------------|100%
  ************************************************
0.09 seconds taken to write coverage file(s).
Hmmm, that was a quick one!
yes
```

Note that changing the first argument of `w/2` to `weka` or `cn2` would produce files acceptable by the respective systems, `text` would produce a plain text file, and that appropriate file name extension is added automatically. To proceed to the rule induction component, exit YAP (`CTRL+D`), relaunch it, and in a manner analogous to the above procedures, type

```
[r]. <Enter>
```

to consult the linked file `<rsd>/code/rules.pl`. To read the `train.b` file along with the result of the preceding step (i.e. the propositionalized representation `trains.cov`), you again – surprise – use the command

```
r(trains). <Enter>
```

To induce rules, type

```
i. <Enter>
```

obtaining the following in response.

```
Inducing rules.

Class: east  (prior = 0.5)

IF 93 AND 69 AND 16 THEN Class = east
Evaluation: 0.225 Significance: 9.0 Class distribution: [10,1]
Remaining % of initial total weight = 75.0

IF 93 AND 112 AND 16 THEN Class = east
Evaluation: 0.15 Significance: 12.0 Class distribution: [9,0]
Remaining % of initial total weight = 68.0

IF 106 THEN Class = east
Evaluation: 0.062 Significance: 7.0 Class distribution: [5,0]
Remaining % of initial total weight = 65.0

IF 9 AND 106 THEN Class = east
Evaluation: 0.048 Significance: 7.0 Class distribution: [5,0]
Remaining % of initial total weight = 64.0
```

(...etc.)

When rules have been generated, you may save them by typing

```
w. <Enter>
```

(this will create the `trains.rules` file). When you run `rules.pl` newly, you may load these rules back by typing

```
rr. <Enter>
```

(after the initial command `r(trains).`) instead of re-inducing them. This may be used for later evaluation purposes (explained further in this text). Finally, the command

```
s. <Enter>
```

serves to show the rules currently in the memory.

3 Using RSD

3.1 Language declarations

Every effort has been taken to make mode-language declarations as similar as possible to the popular systems Progol and Aleph, so that existing declarations and background knowledge can be recycled.

All predicates which can appear in the features are declared in the `*.b` file. There is exactly one *head* declaration describing the predicate of the classified (main) relation. It has the form

```
:-modeh(1,<S>(+<T>)).
```

where `<S>` is the predicate symbol of the main relation (found in the `.f` and `.n`, or `.pl` file(s)) and `<T>` specifies the *type* of the variable that addresses learning instances. It may be an arbitrary atom (a self-explaining name is recommended though). An appearance of this variable type in body predicates (see below) will denote that an instance-addressing variable should occur in the place of that appearance. The number 1 and the + sign in the head declaration are included only for compatibility sakes. Note that a unary predicate is declared by the head declaration even if one works with a `.pl` data file containing binary facts; the declaration merely serves to identify the main predicate symbol and the instance key.

The *body* declarations specify predicates which can appear as literals in the feature body. They have the form

```
:-modeb(<R>,<S>(<M1><T1>,<M2><T2>, ... ,<Mn><Tn>)).
```

where

- `<R>`, the *recall*, is a number which specifies how many times the predicate can appear in one feature body with the same input variables (those whose *mode* is +, see below). This can be used for predicates giving multiple answers (outputs) to a given input.
- `<S>` is the symbol of the declared predicate.
- `<Mi>` is the *mode* of the i^{th} argument variable, which may either be + (input variable) or $-$ (output variable). I shall explain the meaning of the *mode* in a moment.
- `<Ti>` is the *type* of the i^{th} argument variable, which is an arbitrary atom.

The modes and types are used to constrain the feature-language bias, similarly again to the systems Progol and Aleph. They are employed as the following rule dictates.

> *A predicate will only be considered as a literal at a given place in the feature definition, if each of its input arguments is of a type equal to the type of an output argument of some preceding literal, or equal to the type T found in the* `modeh` *declaration.*

Predicates, which are declared by `modeb/2` with no output variables, are called *property predicates*. Other predicates declared by `modeb/2` are called *structural predicates*.

Unlike Progol, the user does not specify value-domains of variable types. Types are here used solely for purposes of variable matching. Unlike both Aleph and Progol, there is no variable mode labelled by the sign #, which in these systems means that a constant value should be put in the given argument place. RSD does not extract constants from a single example, as in the mentioned systems, but rather selects them 'carefully' from the whole data set in the following manner.

There is one special reserved unary property predicate called `instantiate/1`, which may not occur in the background knowledge. It specifies a variable that should be substituted with a constant during the subsequent feature processing by the `process.pl` component. Let us illustrate this on an example. Out of the following declarations[3] in the East-West trains domain

```
:-modeh(1, train(+train)).
:-modeb(1, hasCar(+train, -car)).
:-modeb(1, hasLoad(+car, -load)).
:-modeb(1, hasShape(+load, -shape)).
:-modeb(*, instantiate(+shape)).
```

exactly one feature would be generated:

```
f(1,A) :- hasCar(A,B),hasLoad(B,C),hasShape(C,D),instantiate(D).
```

However, the component `process.pl` would take `f1` and substitute it by a set of features, in each of which the `instantiate/1` literal is removed and the `D` variable is substituted by a constant, making the body of `f1` provable in the data. Provided they contain a sufficient number of trains with a rectangle load, the following feature will appear among those created out of `f1`:

```
f(1,A) :- hasCar(A,B),hasLoad(B,C),hasShape(C,rectangle).
```

A similar principle applies for features with multiple occurrences of the literal `instantiate/1`. Arguments of this literal within the feature form a set of variables ϑ; only those (complete) instantiations of ϑ making the feature's body provable on the input database will be considered.

3.2 Settings

Each of the three RSD components has a number of configurable parameters, which can be specified in the `.b` file by using the `set/2` command in the following form

```
     :-set(<parameter>,<value>)
```

The set of parameter settings in the `.b` file is common to all three RSD components that load the `.b` file; each one will obviously apply the relevant subset. Tables 1 to 4 provide explanations to these settings. For the components `featurize.pl` and `process.pl` there is a special group of formatting-related settings. These define, for example, whether constructed features should contain variables such as `A, B, C, ...` or `Car1, Car2, Car3 ...`, where the feature numbers should be put, etc.

[3]The recall parameter has no effect on the `instantiate/1` predicate.

`featurize.pl`

—

Essential Settings

Parameter	**Value and Function**
clauselength	An integer specifying the maximum length of a feature body. **Default** is 8.
depth	An integer specifying the maximum *depth* of variables found in a feature body. See e.g. S. Muggleton: *Inverse Entailment and Progol*, New Gen. Computing 13, 1995 about *variable depth*. **Default** is 4.
max_occ	A compound term (Symbol/Arity, Occurrences) specifying the maximum number of occurrences of the predicate with given symbol and arity in a feature body. Not limited by default.
pruning	Value is `on` or `off`. If off, admissible search-space pruning functions are not used. This is not recommended. **Default** is `on`.

Formatting Specifications

Parameter	**Value and Function**
head_name	An atom specifying the head symbol of features. **Default** is `f`.
feature_num	One of `argument`, `end_of_name`. In the former case, feature heads are numbered as `f(1,A)`, `f(2,A)`, etc., otherwise `f1(A)`, `f2(A)`, etc. **Default** is `argument`.
format_vars	One of `alphabet`, `all_cap_type` and `first_cap_type` `alphabet`: Names of used variables are subsequently A, B, C, ... `all_cap_type`: Variables have the name of their type, followed by an integer, for the Trains domain e.g. `CAR1`, `CAR2`, ... `first_cap_type`: Same as above, but `Car1`, `Car2`, ... **Default** is `alphabet`.

Table 1: Parameter settings relevant to the `featurize.pl` component of RSD. Note that the default values of the formatting specifications should be used to create an output file correctly processable by `process.pl`.

`process.pl`

—

Essential Settings

Parameter	**Value and Function**
negation	One of `now`, `later`, `none`. `now`: To features generated by `featurize.pl`, `process.pl` will also add their versions where the complete body is negated. (Negations of individual literals can be done by suitably defining background knowledge predicates.) `later`: tells the program that an inducer capable of negating features will be applied on the propositionalized representation. This influences functions described below. **Default** is `none`.
min_coverage	An integer *mc*. All features (including negated versions) covering fewer than *mc* instances will be discarded. However, if `negation` is `later` (see above), a feature is discarded only if both (a) coverage thereof and (b) the coverage of its negated version are smaller than *mc*. **Default** is `1`.
filtering	One of `true`, `false`. If `true`, each feature will be discarded if (a) it covers the same set of instances as some previously constructed feature, or (b) it covers all instances. **Default** is `true`.
splitting	Either `false`, or a number in $(0, 1)$, or an integer. If `false`: the resulting propositional representation (call it RPR) will describe all instances of the input data. If in $(0, 1)$: RPR will be split into train and test sets. The number then indicates the proportion of instances to be put in the test split. Supported for RSD- and CN2-formatted RPR's. (For Weka, train/test splitting is a built-in). If integer N: RPR is a set of N stratified cross-validation folds. Supported only when creating RPR in an RSD rule-inducer format. Cross-validation splits for the CN2 format is not yet supported by RSD. For Weka, cross-validation splitting is a built-in. No kind of splitting is at the moment available for plain-text RPR (it has to be done manually). **Default** is `false`.

Table 2: Essential parameter settings relevant to the `process.pl` component of RSD. 'RPR' abbreviates 'resulting propositional representation'.

`process.pl`

—

Formatting Specifications

Parameter	**Value and Function**
true_sign	Determines the sign used to indicate that a feature holds for an instance in the RPR attribute table. **Default** is '+'.
false_sign	See above and guess :-). **Default** is '-'.
separator	The symbol for separating attributes in a row of the RPR attribute table. **Default** is ' ' (space).
terminator	The symbol that should appear in the end of rows in the RPR attribute table. **Default** is '' (empty char).
meter_width	Integer specifying the width (in characters) of the progress meter displayed when conducting an exhaustive job. **Default** is 50.

Table 3: Formatting specifications relevant to the `process.pl` component of RSD. 'RPR' abbreviates 'resulting propositional representation'. Note that the first four settings apply only when generating a plain-text RPR, and are overridden when creating files for specified learning systems (CN2, Weka, RSD rule-inducer).

3.3 Basic Commands

Although each of the three components of RSD provides a different set of services, the basic user's control of all the components is quite unified. Therefore the description below is common to all the three programs. The commands are always issued in the YAP console after having consulted the respective program code (e.g. for the `featurize.pl` component this was done in the Quick Start section by `[f]. <Enter>` as the the file `f.pl` linked to the `<rsd>/code/featurize.pl` program was present in the current directory).

In the following, expressions of the form `S/A` correspond to a predicate of the symbol `S` and arity `A`.

`r(`*filename*`).` Reads input file(s). All three program components read the *filename*`.b` file, but `rules.pl` will not complain if it does not find the file (it simply won't apply any settings then). The `process.pl` will also read the *filename*`_frs_noinst.pl` and *filename*`_frs.smb` files, which have been created by `featurize.pl`, and the classified data file(s) (see Section 1). The `rules.pl` will also read the *filename*`.cov` file containing the propositional representation created by `process.pl`.

`s.` Shows the product on the screen: On this command, `featurize.pl` will create non-instantiated features and list them, `process.pl` will process features and list the resulting set. The component `rules.pl` upon this `rules.pl`

—

Essential Settings

Parameter	**Value and Function**
beam_width	An integer specifying the width of the beam in the beam-search performed to find rules. **Default** is 3.
max_length	An integer specifying the maximum length of an induced rule in literals. **Default** is 3.
evalfn	One of `acc`, `wracc`. Specifies the heuristic rule-evaluation function (accuracy, or weighted relative accuracy). **Default** is `wracc`.
acc_est	One of `freq`, `laplace`. Specifies how rule accuracy should be estimated (plain frequency count, or the Laplace estimate). **Default** is `freq`.
search	One of `cover`, `weighted`. If `cover`, then the standard instance-covering algorithm is used in the rule-induction cycle. If `weighted`, then the *instance-weighting* approach is taken. **Default** is `weighted`.
weight_threshold	Number $c \in (0, 1)$. In an *instance-weighting* induction, generation of rules for the current class is stopped once the total weight of the class instances falls below $c.w$ where w is the initial total weight of the class instances. **Default** is 0.1.
gamma	In an *instance-weighting* induction, number specifying the γ parameter in the weight-decay formula. **Default** is 1.
sig_threshold	Number specifying the minimum significance value of an acceptable rule. **Default** is 0.
eval_threshold	Number specifying the minimum heuristic evaluation value of an acceptable rule. **Default** is 0.
max_rules_for_class	Integer specifying the maximum number of rules that should be generated for one class. **Default** is 10.
stop_on_sig	One of `yes`, `no`. If `yes`, then rule-generation for the current class terminates once the best rule in the search space *deceeds* the minimum significance value specified by `sig_threshold`. **Default** is `no`. *Remark:* look at `http://130.88.203.73/asktheexperts/faq/aboutwords/exceed` if you doubt the verb *to deceed* (my spell-checker does).
stop_on_eval	Same as `stop_on_sig`, but related to the minimum evaluation determined by `eval_threshold`.

Table 4: Parameter settings relevant to the `rules.pl` component of RSD.

command will show the induced rules, but the rules must have been previously induced, or read from a `.rules` file (see the `i/0` command below).

`w.` The same as the `s.` command above, but instead of displaying on the screen, the results are written into appropriate files. File name is created by taking *filename* from the initial `r/1` command (see above) and adding appropriate extensions.

`w(`*filename*`).` Just like `w/0` above, but allows to specify a different file name (extensions again added automatically).

`w(`*system*, *filename*`).` Only for `process.pl`, after having used the `w/0` or `s/0` command: creates a file containing the propositional representation of the data using the generated features. The format is determined by the *system* argument, which can be one of `rsd` (for the RSD rule-inducer component), `cn2`, `weka` or `text` (see the formatting parameters in Table 3 for the last option). File name extension is added automatically.

`i.` Only for `rules.pl`: induces rules. Each rule will have a conjunction of feature numbers in its antecedent (the meaning of individual features has to be looked up in the `_frs.pl` file), the class name in the consequent, and a few characteristic quantities attached, calculated on the training data.

`rr.` Only for `rules.pl`: reads rules from the file *filename*`.rules` (erasing any rules currently in memory), where *filename* is taken from the initial `r/1` command.

`rr(`*filename*`).` Just like `rr/0` above but allows to specify a different file name (extension added automatically).

Two special commands, common to all three components are the following.

`set(`*parameter*, *value*`).` This command changes a parameter as described in Section 3.2. Another usual way to specify parameters is to put this command into the `.b` file in the spirit of Section 3.2. Obviously, issuing this command will overwrite the corresponding setting read previously from the `.b` file. For example, you may want to repeatedly create features, induce rules, etc., each time with altering constrain settings, in a single session with an RSD component.

`st.` Shows current parameter settings on the screen (excluding those that currently acquire the default value). Note that this listing may also contain settings not relevant to the currently running RSD component, as the settings may have been specified commonly for all components in the `.b` file.

Finally, the two following commands (relevant to `featurize.pl`) may be in principle issued interactively, but it is not recommended to do so.

`modeh/2` The head-literal declaration command.

`modeb/2` The body-literal declaration command.

Rather, declarations should be a part of the `.b` file, as explained in Section 3.1.

3.4 Assessment Functions

The components `process.pl` and `rules.pl` have special built-in functions enabling for a statistical assessment of discovered rules, from the point of view of subgroup interestingness. Also, detecting and discarding rules not lying on the convex-hull of the ROC curve is supported.[4]

Although RSD has some tentative functions allowing to test the *predictive classification* capacity of induced rule models, they are not yet supported 'officially', as the primary scope of RSD is (surprise:) subgroup discovery.

We will now illustrate the use of the assessment functions again on the East-West trains example domain for clarity. Note, however, that data splitting (and especially cross-validation) on this toy domain counting 20 instances may yield funny results.

3.4.1 Quality Calculations and Train-Test Validation

The assessment procedure starts already when producing the propositional representation by the component `process.pl`, where the user specifies the ratio of the number of testing instances to the number of all instances. After features have been created by the command `s.` or `w.`, specify the ratio in the YAP console (if you haven't set it in the `.b` file already), e.g. 0.2, as follows

```
?- set(splitting,0.2). <Enter>

yes
```

Then, after issuing the command

```
w(rsd,trains).  <Enter>
```

two files are generated: `trains.cov` and `trains_test.cov`, of which the latter contains the description of 20% of instances.[5] The two splits are stratified, which means that they have the same (or very close) distribution of the target-class values.

Subsequently, after launching the `rules.pl` component, the training file is read by

```
r(trains).
```

[4]An explanation of the method of creating subgroup-discovery suited ROC curves as well as of the significance and other characteristics can be found in the paper referred to in the Introduction, but is beyond the scope of this manual.

[5]When generating the split files for CN2 by `w(cn2,trains).`, the files will be: (a) `trains.att` – the attribute description file, (b) `trains_train.exs` – the training examples, and (c) `trains_test.exs` – the testing examples.

and rules are induced by the `i.` command. Say you first want to check the average quality of the induced rules, as it appears to be on the training data. Proceed as follows.

```
?- sq. <Enter>

Data file = trains
Number of rules = 20
Average Length of Rule = 1.95
Average Significance = 4.44
Average Coverage = 22.5% of instances
Average (over classes) Area Under Roc = 0.8336

yes
```

The `sq.` command calculates and reports self-explaining rule characteristics, averaged over all induced rules (subgroups). Since the ROC diagram is constructed for each class separately, the last reported number is further averaged over all classes.

The following commands will read[6] the testing file (training instances are automatically erased from the memory), and calculate the subgroup characteristics on the testing instances.

```
?- r('trains_test'). <Enter>
 [ consulting trains_test.cov... ]
 [ trains_test.cov consulted 21228 bytes in 0 seconds ]

trains_test.b file not found, no settings changed.

yes
   ?- sq. <Enter>

Data file = trains_test
Number of rules = 20
Average Length of Rule = 1.95
Average Significance = 0.38
Average Coverage = 16.25% of instances
Average (over classes) Area Under Roc = 0.9375

yes
```

The message "`trains_test.b file not found, no settings changed.`" is not an error, it merely means that only data, not settings, have been replaced.

[6]Remind that that quoting marks must now be used in the argument of `r/1` due to the underline character in the name.

Now you may discard all rules, which do not lie on the convex hull of the ROC curve belonging to the class present in the consequent of the respective rule, which is done by the command

```
dr.  <Enter>
```

(it can only be issued *after* the `sq.` command which constructs the ROC curves). In the case of this example, only one rule is discarded and the new average subgroup characteristics can again be checked by the `sq.` command. However, this approach is not totally correct because rules would be filtered according to the ROC curves generated on the basis of the *testing*, not *training* data. The correct and exhaustive sequence of commands (which is also followed by the automated validation procedure described in the next Section) would thus be as follows.

1. `r(trains).  <Enter>`
   (Reads the training data.)

2. `i.  <Enter>`
   (Induces rules.)

3. `w.  <Enter>`
   (Writes the induced rules into the file `trains.rules`.)

4. `sq.  <Enter>`
   (Calculates and shows characteristics of unfiltered rules on training data.)

5. `dr.  <Enter>`
   (Discards rules below ROC convex hull.)

6. `w('trains-ch').  <Enter>`
   (Writes the filtered rules into the file `trains-ch.rules`.)

7. `sq.  <Enter>`
   (Calculates and shows characteristics of filtered rules on training data.)

8. `rr.  <Enter>`
   (Reads back the unfiltered rules, erasing the filtered rules from memory.)

9. `r('trains_test').  <Enter>`
   (Reads the testing data.)

10. `sq.  <Enter>`
    (Calculates and shows characteristics of unfiltered rules on testing data.)

11. `rr('trains-ch').  <Enter>`
    (Reads back the filtered rules.)

12. `sq.  <Enter>`
    (Calculates and shows characteristics of filtered rules on testing data.)

3.4.2 Automating the Cross-Validation Procedure

Performing the validation procedure for a number of cross-validation folds would be tiresome, therefore RSD provides means of automation.

First of all, the stratified cross-validation folds are again produced by the `process.pl` component as in the example above, however, the `splitting` parameter is set this time to an integer N defining the number of folds. For example, using

```
set(splitting,5).  <Enter>
```

and then

```
w(rsd,trains).  <Enter>
```

5 cross-validation folds will be produced, represented by 5 pairs of files

`trains1.cov`, `trains1_test.cov` ... `trains5.cov`, `trains5_test.cov`.

When the rule inducer `rules.pl` is launched, one may read in the `trains.b` file (if there are any settings to be applied) by

```
['trains.b'].  <Enter>
```

instead of using the `r(trains).` command (this is because one does not assume the file `trains.cov` to exist in the cross-validation case). The rest of the job is very simple. Just type

```
validate(trains).  <Enter>
```

The program will automatically detect the number of cross-validation folds present in the current directory (and even if there are none, it will check if files of a single train-test split exist) and perform an adequate number of induction - validation procedures, with progress commented on the screen. After they are completed, the following message will appear

```
Cross-validation completed with 5 folds.  Results stored in
files:  'report.txt', 'report-test.txt', 'report-ch.txt' and
'report-ch-test.txt'.
yes
```

The mentioned files located in the current directory then contain tabulated results measured for (respectively): all rules on the train set, all rules on the test set, convex-hull rules on the train set and convex-hull rules on the test set. Here the convex-hull rules are always obtained by inspecting the ROC curves created on the basis of the train sets.

Furthermore, induced rules are stored for each training fold. All rules generated on fold number 1 are stored in the file `trains1.rules`, convex-hull rules for this fold are found in `trains1-ch.rules`, and similarly for the rest of folds.

4 Frequently Asked Questions (none yet)

Got any? Mail me at `zelezny@fel.cvut.cz`. All questions, besides being answered, will enter a lottery with a valuable first (and only) prize (yet to be determined). Drawing on April 30, 2003.