NAME

ripper − learns a rule set from examples

SYNOPSIS

ripper [options] filestem

DESCRIPTION

Ripper is a program for inducing classification rules from a set of preclassified examples; as such it is broadly similar to learning methods such as neural nets, nearest neighbor, and decision tree induction systems such as CART, C4.5 and ID3. The user provides a set of examples, each of which has been labeled with the appropriate *class.* Ripper will then look at the examples and find a set of rules that will predict the class of later examples.

Ripper has several advantages over other learning techniques. First, ripper's hypothesis is expressed as a set of if-then rules. These rules are relatively easy for people to understand; if the ultimate goal is to gain insight into the data, then ripper is probably a better choice than a neural network learning method, or even a decision tree induction system like CART. Second, ripper is asymptotically faster than other competitive rule learning algorithms; this means that it will be much faster on large datasets. Third, ripper allows the user to specify constraints on the format of the learned if-then rules. If there is some prior knowledge about the concept to be learned, then these constraints can often lead to more accurate hypotheses. Fourth, ripper allows attributes to be either nominal, continuous, or "set-valued". The value of a set-valued attribute is a set of atoms: for example, a set-valued attribute could be used to encode the set of words that appeared in the body of a document. Recent versions of Ripper also support bag-valued attributes.

OPTIONS TO RIPPER

The sole argument to ripper is a *filestem* that determines the name of ripper's input files. (The input files for ripper are described below.) The options to ripper have the following meanings.

−c Expect noise-free data.

−n Expect noisy data (the default.)

−k*num* Estimate error rates with k-fold cross-validation. The training is split into k disjoint partitions, and the learning algorithm is trained on every collection of k-1, and then tested in the remaining partition.

−l Estimate error rate with leave-one-out cross-validation (ie, N-fold cross-validation where N is size of training set.)

−v*lev* Set the trace level ("verbosity") to *lev,* which must be either 0, 1, 2, or 3. The default is 0.

−a*ordering* Before learning, ripper first heuristically orders the classes; by one of the following methods: +freq, order by increasining frequency (the default); -freq, order by decreasing frequency; given, order classes as in the names file; mdl, use heuristics to guess an optimal ordering; unordered (see below).

After arranging the classes ripper finds rules to separate class1 from classes class2, ... classn, then rules to separate class2 from classes class3, ... classn, and so on. The final class classn will become the default class. The end result is that rules for a single class will always be grouped together, but rules for classi are possibly simplified, because they can assume that the class of the example is one of classi, ... classn. If an example is covered by rules from two or more classes, then this conflict is resolved in favor of the class that comes first in the ordering.

With the '-aunordered' option, ripper will separate each class from the remaining classes, thus ending up with rules for every class. Conflicts are resolved by deciding in favor of the rule with lowest training-set error.

−s Read the training data from standard input, rather than from filestem.data.

−g*filename*

Use grammar file filename.gram.

–f*filename*
Use names file filename.names.

–O*n* Control optimization of rules. Ripper makes n optimization passes over the rules it learns. The default is n=2.

–M*n* Use statistics collected on a class-stratified subsample of *n* examples (instead of the entire dataset) to make certain frequently repeated decisions. For very large datasets, RIPPER using subsamples of a few hundred or a few thousand will typically produce a slightly inferior ruleset; however, it will run much more quickly than RIPPER without subsampling.

–I*n* Discretize continuous attributes into *n* equal-frequency segments. (If *num* is zero, discretize into the maximal possible number of segments.) Default is to not discretize continuous values. Discretization usually speeds up ripper on large datasets with many continuous values, but may cost in accuracy.

–G Print the grammar and exit. This is sometimes useful when one would like to make a change to the default grammar.

–N Print a names file and exit. This is sometimes useful when one would like to generate a names file for use by C4.5. (Ripper can usually infer the types of an attribute from a dataset, so a names file for Ripper is optional.)

–R Randomize operation. (By default, a fixed random seed is used.)

–!*string* Allow or disallow negative tests in rules. If the string contains a "s", then negative tests of the form "attribute !˜ value" for set- and bag- valued attributes will be allowed in rules. (The symbol "!˜" stands for "does not contain".) If the string contains an "n", then negative tests of the form "attribute != value" for nominal attributes will be allowed in rules.

–D*n* Change the maximum "decompression".

–S*n* Simplify the hypothesis more (n>1) or less (n<1).

–L*n* Change the "loss" ratio, ie the ratio of the cost of a false negative to the cost of a false positive. A value of n>1 will usually improve recall of the minority classes, and a value of n<1 will usually improve precision.

–A Add redundant tests to rules. This sometimes improves precision and readability, principly for set- or bag-valued attributes that contain sets of English words.

–F*n* Force each rule to cover at least *n* examples.

INPUT FILES

The files read and written by ripper are of the form *filestem.ext* where *filestem* is the first and only argument to ripper. All of ripper input files are free format (i.e. white space is not important.) Anything following a percent sign character but on the same line is a comment.

Ripper expects to find four files: a **data file** called *filestem.data* containing some preclassified examples, a **test file** called *filestem.test* that contains some additional preclassified examples to be used as test cases, a **names file** called *filestem.names* defining the names of the classes and attributes used in the data file, and a **grammar file** called *filestem.gram* defining the rules that are allowed to be used in a hypothesis. Except for the grammar file, the format for these files is roughly the same as used by C4.5. The format will be described in more detail below. The last three files are optional. If there is no test file ripper will either not test its learned rule set, or (if directed by the user to do so through the **-k** or **-y** options) ripper will use cross-validation to test its learned rule set. If there is no names file, ripper will assign arbitrary names to the attributes and classes, and will try to figure out the types of the attributes from the data. If there is no grammar file, ripper will use the default grammar described below.

Ripper also creates a file *filestem.hyp* containing the ruleset or rulesets it found, in a format that is intended to be computer-readable.

An example for ripper is described by a fixed set of *attributes.* These attributes can be either continuous, nominal, set-valued, or bag-valued. Continuous attributes have real-number values. The value of a nominal attribute is one of a fixed set of symbolic values, for example "on, off" or "low, medium, high". The value of a set- or bag-valued attribute is a set of atoms (rather than a single symbolic value.) These attributes, as well as the classes that are to be predicted, are defined in the *names file.*

The names file contains first, a comma-separated list of atoms representing the classes, terminated by a period. (An *atom* contains only letters, digits, and the underscore character, and must begin with a letter. Alternatively, an atom is any sequence of characters enclosed in single quotes.) The list of classes is followed by a list of *attribute definitions.* Each attribute definition consists of the name of the attribute, e.g. "height" or "sex"; a colon; and either the atom *continuous* if the attribute is continuous, the atom *set* if the attribute is set-valued, the atom *bag* if the attribute is bag-valued, the atom *symbolic* if the attribute can take on any symbolic value, or a comma-separated list of atoms representing possible symbolic values of the attribute, if the attribute is nominal. Finally, every attribute definition must be terminated by a period.

Ripper also supports *ignored* and *suppressed* attributes. Ignored attributes are completely ignored by the learning system. To define an ignored attribute, use a declaration of the form *attribute_name: ignore.* Suppressed attributes are similar, except that while they are not used in Ripper's hypotheses, the number of values of the attribute does effect MDL-based pruning. Hence, suppressing an attribute that was not used in a hypothesis should not change Ripper's performance in any way. An attribute is "suppressed" by inserting the keyword *suppressed* after the colon in the atrribute's definition.

The *data file* contains a set of classified examples. Each example is a comma-separated list of attribute values, followed by an atom indicating the class of the example, followed by a period. (It is usually convenient to have one example per line, but this is not required.) Attribute values are given in the same order that attributes are defined in the names file; most of the usual syntaxes for numbers are supported. Set- and bag-valued attributes are specified by simply enumerating the elements of the set, separated with whitespace. Unknown attributes are indicated with a question mark token.

Examples can also be given a weight, by inserting *:w* between the class name and the terminating period (where *w* is a real number, the default value for which is one).

The *test file* is formatted in the same way as the data file.

Finally, the *grammar file* contains a description of a context-free grammar, roughly in BNF notation. The grammar file is optional for ripper, and most users will be probably not want to change the default grammar; however we will describe it here for completeness. The terminal symbols of the grammar are tests on the values of attributes defined in the names file; each sentence generated by the grammar is thus a sequence of attribute-value tests. Ripper will read in this grammar and constrain its learning component so that every rule generated by ripper will have as an antecedent a sequence of attribute-value tests that is a sentence of the grammar. The grammar thus is a way for the user to guide ripper's choice of rules.

More specifically, the grammar file contains a series of *grammar rules.* Each grammar rule consists of an atomic *left-hand side* followed by the token "-->" followed by a comma-separated list of *grammar symbols* followed by a period. A *grammar symbol* is either a nonterminal symbol (which is simply an atom that appears on the left-hand side of some grammar rule) or a *terminal symbol.* A terminal symbol is of the form *attribute op value* where *attribute* is the name of an attribute (e.g. "height") and value is a valid value for that attribute. An operator *op* must be one of the tokens "=", "!=", ">=", "<=", "~" or "!~". Terminal symbols of the form *attribute op* * are also allowed, in which case any possible value is allowed.

The condition *attribute ~ symbol* is used for set- and bag-valued attributes. The condition *attribute ~ symbol* is true of an example if *attribute* is set-valued and *symbol* is contained in the set. The condition *attribute !~ symbol* is true if *symbol* is not present in the set. For bags, the condition *attribute ~ symbol__k* is true if *attribute* contains at least *k* instances of *symbol.* The condition *attribute !~ symbol__k* is treated analogously.

Often one will have several grammar rules with the same left-hand side, but different right-hand sides. In this case one may use the syntax

LHS --> RHS1 | RHS2 | ... | RHSk

rather than the wordier

```
LHS --> RHS1
 ...
LHS --> RHSk
```

Finally, prefixing a grammar rule with an exclamation point indicates to ripper that sentences generated using that grammar rule have a lower priority; if possible, ripper will build a hypothesis without using low-priority sentences. Even lower priorities can be assigned by prefixing grammar rules with a string of two or more exclamation points.

THE DEFAULT GRAMMAR

When learning rules to predict the class "class", ripper will expect to find some left-hand side of the form "body_class" to use as the start symbol of the grammar; if this is not present, ripper will use the atom "body" as the start symbol. If this is not present, ripper will construct the following default grammar:

```
body --> body_conds.
body_conds --> .
body_conds --> cond,body_conds.
cond --> attr1_cond.
 ...
cond --> attrk_cond.
```

where *attr1, ..., attrk* are the names of the attributes defined in the names file. If discretization is used, then for each continuous attribute *cattr,* the default grammar also contains the rules

```
cattr_cond --> cattr>=t1.
cattr_cond --> cattr<=t1.
...
cattr_cond --> cattr>=tn.
cattr_cond --> cattr<=tn.
```

where *t1, ..., tn* are ripper's discretization of the training data. Otherwise, the grammar will contain the rules

```
cattr_cond --> cattr>= '*'.
cattr_cond --> cattr<= '*'.
```

For a nominal attribute *nattr* the default grammar contains the rule

```
nattr_cond --> nattr = '*'.
```

For a set- or bag-valued attribute *sattr* the default grammar contains the rules

```
sattr_cond --> sattr ~ '*'.
sattr_cond --> sattr !~ '*'.
```

If the grammar file is missing or empty, then the default grammar will be used. If the grammar contains definitions of some but not all of the nonterminal symbols used in the default grammar, they will override the default definitions.

FILES

ripper
filestem.data (data file)
filestem.names (names file)
filestem.gram (grammar file)
filestem.test (unseen data)
filestem.hyp (learned rules)

Some sample input files are also available from wcohen@research.

SEE ALSO

The man page for *ripperaux* contains brief descriptions of some additional useful programs for working with ripper rulesets and/or datasets.

Ripper's input files are more-or-less compatible with Quinlan's *C4.5* tree-learning system.

The papers "Fast efficient rule learning" (Cohen, ML95) and "Learning trees and rules with set-valued features" (Cohen, AAAI96) describe the algorithms used in Ripper in more detail.

USING RIPPER TO CLASSIFY TEXT

I am frequently asked about tools to preprocess text so that it can be easily digested by Ripper. I don't have any tools to distribute, largely because I think it would be hard to have any tool that is sufficiently general to handle all the necessary cases, but still substantially simpler than a general-purpose text processing language like Perl.

My current recommendation in feeding Ripper is to use something like Perl (or whatever suits you) to convert punctuation to white space, and coerce everything to lower case, and then feed the result into Ripper as a single set (or perhaps bag). If you use sets, it is not necessary to remove duplicate tokens. Be careful to remove the punctuation symbols percent sign (%), comma (,), colon (:), single quote ('), and period (.), all of which have special meaning to Ripper. If you want a non-default tokenization, then you must surround each token with single quotes. Stemming doesn't seem to make a big difference on the benchmarks I've tried. Coercion to lower case also means that you can safely use any upper-case or mixed-case symbol as an attribute or class name.

BUGS

Attribute names, attribute values, grammar symbols, and class names are all put in the same name space, so you can't use the same symbol for, say, a class name and a possible set value. This is awkward when you're using set- or bag-valued attributes to handle text.

Ripper doesn't actually check the range of symbolic attributes for consistency with declaration in the names file.

The response of ripper to the -S and -L options is sometimes rather abrupt---i.e. small changes can sometimes have drastic consequences.