RIPPER Algorithm

Rahul Singh

1/17/2018

Classification Rules

Repeated Incremental Pruning to Produce Error Reduction (RIPPER) algorithm (one of the covering algorithms)

Classification rules show knowledge in logical forms of if-else statements that assign a class to unlabeled examples (in terms of antecedent and consequent).

The antecedent comprises certain combinations of feature values, while the consequent specifies the class value to assign when the rule's conditions are met.

Rule learners are used similar to dicision tree learners. Practical applications include- identifying conditions in manufacturing units that lead to failure, describing key characteristics of groups of people for customer segmentation, finding conditions that come before increase or drops in prices of shares of stock.

Some advantages over trees: Unlike a tree, which must be applied from top-to-bottom through a series of decisions, rules are propositions that can be read much like a statement of fact. Additionally, for reasons that will be discussed later, the results of a rule learner can be more simple, direct, and easier to understand than a decision tree built on the same data.

Note that rules can be generated using decision trees. But decision trees bring a particular set of biases to the task that a rule learner avoids by identifying the rules directly.

Concept of Separate and Conquer. The process involves identifying a rule that covers a subset of examples in the training data, and then separating this partition from the remaining data. As the rules are added, additional subsets of the data are separated until the entire dataset has been covered and no more examples remain.

As the rules seem to cover portions of the data, separate and conquer algorithms are also known as covering algorithms, and the resulting rules are called covering rules.

ZeroR: Zero Rules, literally learns no rules and predicts the most common class

1R algorithm: One Rule or OneR improves over ZeroR. Interestingly the accuracy of this algorithm can approach that of much more sophisticated algorithms.

Pros of 1R algorithm: generates a single easy to understand rule, can serve as benchmark for complex algorithms

Cons of 1R algorithm: uses only a single feature, too simple or may be too basic for some tasks.

The way 1R algorithm works is simple. For each feature, 1R divides the data into groups based on similar values of the feature. Then, for each segment, the algorithm predicts the majority class. The error rate for the rule based on each feature is calculated and the rule with the fewest errors is chosen as the one rule.

RIPPER Algorithm

More sophisticated rule learning algorithm. A first step toward solving these problems was proposed in 1994 by Johannes Furnkranz as Incremental Reduced Error Pruning (IREP) algorithm which used a combination of pre-pruning and post-pruning methods that grow very complex rules and prune them before separating the instances from the full dataset. Although this strategy helped the performance of rule learners, decision trees often still performed better.

William W. Cohen introduced the Repeated Incremental Pruning to Produce Error Reduction (RIPPER) algorithm, which improved upon IREP to generate rules that match or exceed the performance of decision trees.

Pros of RIPPER Algorithm: easy to understand, efficient on large noisy datasets, generally produces a simpler model than a comparable decision tree

Cons of RIPPER Algorithm: May generate rules which defy common sense, not ideal for working with numeric data, might not perform as well as more complex models

Three main steps in RIPPER algorithm: grow (separate and conquer; reach a point when entropy no longer reduces), prune (trace back to reach better entropy), optimize.

Problems with large and complex datasets: Eventhough the algorithm can model complex data, just like decision trees, it means that the rules can quickly become more dif cult to comprehend.

Classi cation rules can also be obtained directly from decision trees. Beginning at a leaf node and following the branches back to the root, you will have obtained a series of decisions. These can be combined into a single rule.

The chief downside to using a decision tree to generate rules is that the resulting rules are often more complex than those learned by a rule learning algorithm. The divide and conquer strategy employed by decision trees biases the results differently than that of a rule learner. On the other hand, it is sometimes more computationally efficient to generate rules from trees.

Note: The C5.0() function in the C50 package will generate a model using classification rules if you specify rules = TRUE when training the model.

Why greedy learning algorithms won't work? Decision trees and rule learners are known as greedy learners because they use data on a first-come, first-served basis. Both the divide and conquer heuristic used by decision trees and the separate and conquer heuristic used by rule learners attempt to make partitions one at a time, finding the most homogeneous partition first, followed by the next best, and so on, until all examples have been classified.

The downside to the greedy approach is that greedy algorithms are not guaranteed to generate the optimal, most accurate, or smallest number of rules for a particular dataset. By taking the low-hanging fruit early, a greedy learner may quickly find a single rule that is accurate for one subset of data; however, in doing so, the learner may miss the opportunity to develop a more nuanced set of rules with better overall accuracy on the entire set of data. However, without using the greedy approach to rule learning, it is likely that for all but the smallest of datasets, rule learning would be computationally infeasible.

Though both trees and rules employ greedy learning heuristics, there are subtle differences in how they build rules. Perhaps the best way to distinguish them is to note that once divide and conquer splits on a feature, the partitions created by the split may not be re-conquered, only further subdivided. In this way, a tree is permanently limited by its history of past decisions. In contrast, once separate and conquer nds a rule, any examples not covered by all of the rule's conditions may be re-conquered.

On one hand, because rule learners can reexamine cases that were considered but ultimately not covered as part of prior rules, rule learners often find a more parsimonious set of rules than those generated from decision trees. On the other hand, this reuse of data means that the computational cost of rule learners may be somewhat higher than for decision trees.

```r
mushrooms <- read.csv("~/Desktop/RIPPER Algorithm/mushrooms.csv", stringsAsFactors = TRUE)
#Source: http://archive.ics. uci.edu/ml
```

```r
str(mushrooms)
```

```
## 'data.frame':    8124 obs. of  23 variables:
##  $ type                    : Factor w/ 2 levels "e","p": 2 1 1 2 1 1 1 1 2 1 ...
##  $ cap_shape               : Factor w/ 6 levels "b","c","f","k",..: 6 6 1 6 6 6 1 1 6 1 ...
##  $ cap_surface             : Factor w/ 4 levels "f","g","s","y": 3 3 3 4 3 4 3 4 4 3 ...
##  $ cap_color               : Factor w/ 10 levels "b","c","e","g",..: 5 10 9 9 4 10 9 9 9 10 ...
##  $ bruises                 : Factor w/ 2 levels "f","t": 2 2 2 2 1 2 2 2 2 2 ...
##  $ odor                    : Factor w/ 9 levels "a","c","f","l",..: 7 1 4 7 6 1 1 4 7 1 ...
##  $ gill_attachment         : Factor w/ 2 levels "a","f": 2 2 2 2 2 2 2 2 2 2 ...
##  $ gill_spacing            : Factor w/ 2 levels "c","w": 1 1 1 1 2 1 1 1 1 1 ...
##  $ gill_size               : Factor w/ 2 levels "b","n": 2 1 1 2 1 1 1 1 2 1 ...
##  $ gill_color              : Factor w/ 12 levels "b","e","g","h",..: 5 5 6 6 5 6 3 6 8 3 ...
##  $ stalk_shape             : Factor w/ 2 levels "e","t": 1 1 1 1 2 1 1 1 1 1 ...
##  $ stalk_root              : Factor w/ 5 levels "?","b","c","e",..: 4 3 3 4 4 3 3 3 4 3 ...
##  $ stalk_surface_above_ring: Factor w/ 4 levels "f","k","s","y": 3 3 3 3 3 3 3 3 3 3 ...
##  $ stalk_surface_below_ring: Factor w/ 4 levels "f","k","s","y": 3 3 3 3 3 3 3 3 3 3 ...
##  $ stalk_color_above_ring  : Factor w/ 9 levels "b","c","e","g",..: 8 8 8 8 8 8 8 8 8 8 ...
##  $ stalk_color_below_ring  : Factor w/ 9 levels "b","c","e","g",..: 8 8 8 8 8 8 8 8 8 8 ...
##  $ veil_type               : Factor w/ 1 level "p": 1 1 1 1 1 1 1 1 1 1 ...
##  $ veil_color              : Factor w/ 4 levels "n","o","w","y": 3 3 3 3 3 3 3 3 3 3 ...
##  $ ring_number             : Factor w/ 3 levels "n","o","t": 2 2 2 2 2 2 2 2 2 2 ...
##  $ ring_type               : Factor w/ 5 levels "e","f","l","n",..: 5 5 5 5 1 5 5 5 5 5 ...
##  $ spore_print_color       : Factor w/ 9 levels "b","h","k","n",..: 3 4 4 3 4 3 3 4 3 3 ...
##  $ population              : Factor w/ 6 levels "a","c","n","s",..: 4 3 3 4 1 3 3 4 5 4 ...
##  $ habitat                 : Factor w/ 7 levels "d","g","l","m",..: 6 2 4 6 2 2 4 4 2 4 ...
```

```
#we drop veil type because it only has one factor; shows incorrect coding
mushrooms$veil_type <- NULL
```

Transforming the dataset

```
#Below function transforms the type data
transformClassData <- function(key){
  switch (key,
    'p' = 'poisonous',
    'e' = 'edible'
  )
}

#Below function transforms the cap-shape column
transformCapShapeData <- function(key){
  switch (key,
          'b' = 'bell',
          'c' = 'conical',
          'x' = 'convex',
          'f' = 'flat',
          'k' = 'knobbed',
          's' = 'sunken'
  )
}

#Below function transforms the cap-surface column
transformCapSurfaceData <- function(key){
  switch (key,
          'f' = 'fibrous',
          'g' = 'grooves',
          'y' = 'scaly',
          's' = 'smooth'
  )
```

```r
}

#Below function transforms the cap-color column
transformCapColorData <- function(key){
  switch (key,
          'n' = 'brown',
          'b' = 'buff',
          'c' = 'cinnamon',
          'g' = 'gray',
          'r' = 'green',
          'p' = 'pink',
          'u' = 'purple',
          'e' = 'red',
          'w' = 'white',
          'y' = 'yellow'
  )
}

#Below function transforms the odor column
transformOdorData <- function(key){
  switch (key,
          'a' = 'almond',
          'l' = 'anise',
          'c' = 'creosote',
          'y' = 'fishy',
          'f' = 'foul',
          'm' = 'musty',
          'n' = 'none',
          'p' = 'pungent',
          's' = 'spicy'
  )
}

#Below function transforms the population column
transformPopulationData <- function(key){
  switch (key,
          'a' = 'abundant',
          'c' = 'clustered',
          'n' = 'numerous',
          's' = 'scattered',
          'v' = 'several',
          'y' = 'soletary'
  )
}

#Below function transforms the habitat column
transformHabitatData <- function(key){
  switch (key,
          'g' = 'grasses',
          'l' = 'leaves',
          'm' = 'meadows',
          'p' = 'paths',
          'u' = 'urban',
           'w' = 'waster',
           'd' = 'woods'
  )
}
```

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
mushrooms = mushrooms %>% mutate_all(funs(as.factor))
```

```
table(mushrooms$type)
```

```
##
##    edible poisonous
##      3916      4208
```

```
#e is for edible and p for poisonous
#if we used ZeroR then it would predict the mushroom to be edible
```

Training the model on dataset

```
#RWeka package is used to implement 1R
library(RWeka)
```

```
mushroom_1R <- OneR(type ~ ., data = mushrooms)
#OneR function takes in class to be predicted ~ predictors specifying the features, data frame
# ~ . formula allows to consider all the possible features in the mushroom data
mushroom_1R #to examine the rules created by the algorithm; this output shows that odor feature was sel
```

```
## odor:
##  almond  -> poisonous
##  anise   -> edible
##  creosote    -> edible
##  fishy   -> poisonous
##  foul    -> edible
##  musty   -> poisonous
##  none    -> edible
##  pungent -> edible
##  spicy   -> edible
## (8004/8124 instances correct)
```

```
#we can see that 8004/8124 were predicted correctly
```

Evaluating model performance

```
summary(mushroom_1R)
```

```
##
## === Summary ===
##
## Correctly Classified Instances        8004               98.5229 %
## Incorrectly Classified Instances       120                1.4771 %
## Kappa statistic                          0.9704
## Mean absolute error                      0.0148
## Root mean squared error                  0.1215
## Relative absolute error                  2.958  %
## Root relative squared error             24.323  %
## Total Number of Instances             8124
##
## === Confusion Matrix ===
##
##     a    b   <-- classified as
##  3796  120 |    a = edible
##     0 4208 |    b = poisonous
```

In confusion matrix we can see that s is edible and b is poisonous. But the problem is that the model predicted 120 poisonous mushrooms as edible- a dangerous mistake, people can die because of such an error if one were to follow this model!

Improving the model performance

```r
#we use JRip a java based implementation of the RIPPER rule learning algorithm; this is available in RW
mushroom_JRip <- JRip(type ~ ., data = mushrooms)
#JRip function takes in the column in dataframe that needs to be predicted; data that is taken into acc
```

```r
#to examine the rules created by the classifier
mushroom_JRip
```

```
## JRIP rules:
## ===========
##
## (odor = creosote) => type=edible (2160.0/0.0)
## (gill_size = n) and (gill_color = b) => type=edible (1152.0/0.0)
## (gill_size = n) and (odor = none) => type=edible (256.0/0.0)
## (odor = anise) => type=edible (192.0/0.0)
## (spore_print_color = r) => type=edible (72.0/0.0)
## (stalk_surface_below_ring = y) and (stalk_surface_above_ring = k) => type=edible (68.0/0.0)
## (habitat = meadows) and (cap_color = white) => type=edible (8.0/0.0)
## (stalk_color_above_ring = y) => type=edible (8.0/0.0)
##  => type=poisonous (4208.0/0.0)
##
## Number of Rules : 9
```

```r
#to predict you use the predict funtion
#example mushroom_prediction <- predict(mushroom_classifier which is the model trained by Jrip example
```

We can observe that the JRip has created 9 rules to predict.