Classification trees

STAT30270 – Statistical Machine Learning

Contents

1 Classification trees

The main functionalities to fit and plot classification trees are available in packages `rpart` and `partykit`.

```
# load the packages
library(rpart)
library(partykit)
```

We consider an example of application analyzing a dataset concerning angiografic heart disease diagnosis. The data are available in the file `data_heart_cleveland.RData`. More information is available online at `https://archive.ics.uci.edu/dataset/45/heart+disease`.

The target variable is `num` and denotes the presence/absence of an heart disease. The description of the predictor variables is provided in the table below.

Name	Description	Name	Description
`age`	Age of patient	`thalach`	maximum heart rate achieved
`sex`	Sex, 1 for male	`exang`	exercise induced angina
`cp`	chest pain	`oldpeak`	ST depression induced
`trestbps`	resting blood pressure	`slope`	slope of peak exercise ST
`chol`	serum cholesterol	`ca`	number of major vessel
`fbs`	fasting blood sugar larger 120mg/dl	`thal`	probably thalassemia
`restecg`	resting electroc. result	`num`	diagnosis of heart disease (angiographic disease status)

We can easily fit and plot a classification tree using the `rpart` function and the plotting functionalities. Note that the response variable must be coded as a factor variable, otherwise `method = "class"` needs to be specified. By default, the splits are computed using the Gini impurity measure, the entropy measure can be used controlling the argument `parms`, setting `split` to `"information"`. See `?rpart` and the vignette of the package here `https://CRAN.R-project.org/package=rpart` for further information.

```
# load and prepare the data
load("data_heart_cleveland.RData")
table(data_heart$num)  # check class distribution
```

```
# fit and plot tree
ct <- rpart(num ~ ., data = data_heart,
            parms = list(split = "information"))
plot( as.party(ct), cex = 0.5 )
```

We can look at the different splits and nodes in details simply by printing the classification tree object. The function `summary` provides much more information. The output is quite verbose, and it is usually easier to interpret the plotted tree, however some additional information can be extracted.

```
ct
summary(ct)
```

Similarly to logistic regression, we can use the function `predict` to perform predictions and calculate the class probabilities according to the fitted tree. See also the help page `?predict.rpart`. By default the function returns estimated probabilities, the class allocation (obtained by MAP rule) can be returned using argument `type`.

```
# compute class probabilities according to the tree
probs <- predict(ct)
head(probs)

# in-sample predictions
class <- predict(ct, type = "class")
head(class)

table(data_heart$num, class)   # classification table
```

We can compute various predictive performance measures. For example, we can use package `ROCR` to visualize the ROC curve and compute the area under the ROC curve (classes are not imbalanced in this example).

```
library(ROCR)
pred_obj <- prediction(probs[,2], data_heart$num)   # probs[,2] contains probability of class "yes"
roc <- performance(pred_obj, "tpr", "fpr")
plot(roc)
abline(0,1, col = "darkorange2", lty = 2)

# compute the area under the ROC curve
auc <- performance(pred_obj, "auc")
auc@y.values
```

1.1 Task

- **Take some time to read and understand the printed output from the classification tree** `ct`, also with the aid of the documentation. What do the quantities `split`, `n`, `loss`, `yval` and `yprob` indicate in the printed output?
- Split the data into training and test sets and compute and assess the quality of the out-of-sample predictions.

2 Complexity of a tree

The `rpart` function has several default settings that are used to create and control the size of the tree. See `?rpart.control`. In particular, the tree growing method does not split any node with less than 20 observations, controlled by argument `minsplit`. Reducing the value of `minsplit` will lead to more complex trees, as nodes with a smaller number of observations are allowed, so smaller regions of the input space are considered. Moreover, the complexity of the structure of a tree is generally controlled by argument `cp` (default is 0.01). The smaller the value of `cp`, the more complex will be the tree,

that is the more nodes will be present.

```
# fit a more complex tree
ct <- rpart(num ~ ., data = data_heart, cp = 0.01/2)
plot( as.party(ct), cex = 0.2 )

# predictions
probs <- predict(ct)
class <- predict(ct, type = "class")
table(data_heart$num, class)
pred_obj <- prediction(probs[,2], data_heart$num)
auc <- performance(pred_obj, "auc")
auc@y.values
```

More complex trees are very difficult to interpret and will overfit the data, potentially predicting perfectly the classification of the units used to construct the tree itself. This is actually a problem, because, despite the tree is performing very well on the data used for construction, it will likely perform very poorly in classifying new unseen data. This happens because a complex tree will tend to learn features only specific of the current data which cannot actually be generalized to new data. In the example below, increasing further the complexity of the tree will increase the in-sample classification accuracy, leading to a model which perfectly classifies the observations, but performs poorly in terms of generalized predictive performance.

```
# split data in training and out-of-sample
set.seed(1919)
n <- nrow(data_heart)
train <- sample(1:n, 0.8*n)
test <- setdiff(1:n, train)

# complex tree
ct1 <- rpart(num ~ ., data = data_heart[train,],
             cp = 0.01/4)

# more complex tree
ct2 <- rpart(num ~ ., data = data_heart[train,],
             cp = 0.01/5, minsplit = 2)
# plot( as.party(ct3), cex = 0.2 )     # uncomment to see the plot


# training predictions
#
pred_obj1 <- prediction(predict(ct1)[,2], data_heart$num[train])
auc1 <- performance(pred_obj1, "auc")
auc1@y.values
#
pred_obj2 <- prediction(predict(ct2)[,2], data_heart$num[train])
auc2 <- performance(pred_obj2, "auc")
auc2@y.values


# test predictions
#
pred_obj1 <- prediction(predict(ct1, newdata = data_heart[test,])[,2],
                        data_heart$num[test])
auc1 <- performance(pred_obj1, "auc")
auc1@y.values
#
pred_obj2 <- prediction(predict(ct2, newdata = data_heart[test,])[,2],
                        data_heart$num[test])
```

```
auc2 <- performance(pred_obj2, "auc")
auc2@y.values
```

The tree `ct2` has a perfect accuracy on the training data, but it perform very poorly in predicting new unseen observations compared to the simpler tree `ct1`.

2.1 Task

Modify the values of the complexity hyperparameters of the two tree above and investigate their impact on the in-sample and out-of-sample predictive performance.

3 Tuning a tree

We can use the package `caret` to tune the complexity hyperparameter of a tree. As an illustrative example, we examine again the the Behavioral Risk Factor Surveillance System (BRFSS) data introduced in the logistic regression lab and available in the dataset `data_heart_disease_BRFSS2015.csv`.

```
data <- read.csv("data_heart_disease_BRFSS2015.csv")

# make sure that binary/categorical variables are correctly encoded as factor
data[,c(1:4,6:14,18:19)] <- lapply( data[,c(1:4,6:14,18:19)], factor )
str(data)
```

Remember that the two target classes in the data are highly imbalanced. Therefore, in this case we use the AU-PR and the F_1 score as metrics to assess the predictive performance of the model. These metrics will be computed specifying function `prSummary` in the argument `summaryFunction` of `trainControl`. The use of AU-PR for assessment allows to evaluate the predictive performance of the model regardless of a specified classification threshold, which makes things easier. The model tuning will be implemented considering the AU-PR, by specifying the argument `metric` of function `train` to `"AUC"`. Hence, the optimal hyperparameter `cp` is the one which maximizes the AU-PR, therfore is the one which leads to joint high precision and high recall.

The associated precision, recall, and F1 score metric will be computed for a MAP classification (as it is default of `caret`). We decide to use the default MAP since we consider AU-PR among the metrics.

Because the function `prSummary` takes as the positive class the one corresponding to the first level of the factor target variable (see `?prSummary`), we reorder and rename (for convenience) the levels of the target variable.

```
levels(data$HeartDiseaseorAttack) <- c("no", "yes")
data$HeartDiseaseorAttack <- factor(data$HeartDiseaseorAttack, levels = c("yes", "no"))
table(data$HeartDiseaseorAttack)
```

The data include a large number of samples. To keep computational complexity low, we run a simple hold-out 75%–25% procedure with 3 replications. To do so, we create the train partitions at the start, calling the function `createDataPartition`. This will produce in output a list of 3 vectors of indexes of those observations which will be used for training, the left-out observations will be automatically determined by the training function and employed as a validation set at each replication. These partitions can be inputted directly in function `trainControl` in the argument `index`. For the same function, we set `classProbs = TRUE`, as the estimated probabilities are needed to compute the PR curve.

We set a grid of values for the `cp` hyperparameter. Note that the values have been chosen after some trial and error and exploratory runs to get a sense of the range of the optimal values of this hyperparameter.

```
# create in-sample sets for training
train_sets <- createDataPartition(data$HeartDiseaseorAttack,
                                  p = 0.75,
                                  times = 3)

# set training settings
```

```r
train_ctrl <- trainControl(index = train_sets,
                           number = 1,
                           method = "repeatedcv",
                           summaryFunction = prSummary, classProbs = TRUE)

# set cp values
tune_grid <- expand.grid(cp = c(1e-05, 1e-04, 2e-03, 1e-03, 3e-03))

# run cross-validation
fit <- train(HeartDiseaseorAttack ~ ., data = data,
             method = "rpart",
             trControl = train_ctrl,
             tuneGrid = tune_grid,
             metric = "AUC")
```

3.1 Task

- Inspect the output `fit$resample`. What does it contain?
- What is the optimal hyperparameter value of the tree classifier?
- Try different grids for the hyperparameter `cp` and examine how it affects the out-of-sample predictive performance.
- What `summaryFunction` you would need to use if instead of using AU-PR you wanted to tune the model in terms of AU-ROC? Check `?defaultSummary`.
- Set aside some test data and re-run the tuning procedure. Evaluate the generalized predictive performance of the selected classification tree model on the test data. How does it perform in terms of detecting subjects with heart disease?