Seminar 8

Classification II: Pruning Trees

Overview

In this tutorial document, we will construct a classification tree with the *caret* package (short for classification and regression training). *Caret* package was developed to create a unified interface for modeling and prediction. It brings a set of functionalities from 27 packages and supports around 150 models including Bayesian classification, support vector machine (SVM) classification, discriminant analysis, regressions, neural networks, and more.

Caret package aims to be the go-to package for your predictive analytics needs. Thus, it not only covers model training, but also data manipulation, and visualization.

If you haven't installed the caret package yet, please install the package to complete the exercises in this tutorial.

> Install.packages("caret")

> Library(caret)

Data

We will use segmentationData that used in the previous exercise with the *rpart* package.

> data(segmentationData) # Load the data

You can find the descriptions of the variables from the *caret* package.

> ?segmentationData

The data has a variable "Case" to separate the training set and testing set. We do not need the *Case* variable for this exercise. Delete the Case variable from the data and rename the data as 'ClassData'

```
# ! Delete one column #
ClassData <- segmentationData[, !(colnames(segmentationData) == 'Case') ]
```

As we did in the last exercise, we have to separate the data into training and testing sets. We randomly select 80% of the data for the training set and assign the remainder to the testing set.

```
set.seed(123)  # Set seed to ensure reproducible results
num_obs <- nrow(ClassData)
train_size <- num_obs * 0.8
train_sample <- sample(num_obs, train_size)

Class_Train <- ClassData[train_sample, ]
Class_Test  <- ClassData[-train_sample, ]

nrow(Class_Train)
nrow(Class_Test)
```

Classification Trees with Caret

Caret package allows us to change tuning parameters (i.e., complexity parameters such as cost complexity) and re-sampling strategies for the tree models.

- ***K-fold Cross Validation***

By partitioning the original data into training and testing sets, we drastically reduce the number of samples which can be used for learning the model, and the results can depend on a particular random choice for the pair of (train, validation) sets.

A solution to this problem is a procedure called cross-validation (CV). While a test set should still be held out for final model evaluation, the training set is split into *k* smaller sets. The following procedure is followed for each of the *k* "folds":

- A model is trained using $k - 1$ of the folds as training data;
- The resulting model is validated on the remaining part of the data (i.e., it is used as a test set to compute a performance measure such as accuracy).

The performance measure reported by *k*-fold cross-validation is then the average of the values computed in the loop. This approach can be computationally expensive, but does not waste too much data (as it is the case when fixing an arbitrary test set), which is a major advantage in problem such as inverse inference where the number of samples is very small.

To prune a fully-grown tree with the resampling approach, the *train* function in the *caret* package is used. More details on this function can be found by typing help(train).

Before training the model, we have to specify how a new sampling approach works as below:

```
## Set Training Parameters ##
# 10-fold validation repeated three times#
cv_control <- trainControl(method='repeatedcv', number=10, repeats=3,
                           summaryFunction = twoClassSummary, classProbs = TRUE)# Class probabilities for ROC#
```

To specify the resampling method, a *trainControl* function is used. The option *method* controls the type of resampling and "repeatedcv" is used to specify repeated K-folder cross validation (and the argument *repeats* controls the number of repetitions). K is controlled by the number argument and defaults to 10.

Finally, to choose different measures of performance, additional arguments are given to *trainControl*. The *summaryFunction* argument is used to "twoClassSummary" which takes the observed and predicted values and estimate some measure of model performance. It will compute measures specific to two-class problems (e.g., Yes or No), such as the ROC curve, the sensitivity and specificity. Since the ROC curve is based on the predicted class probabilities (which are not computed automatically), another option is required. The *classProbs* = TRUE option is used to include these calculations.

Lastly, the function will pick the tuning parameters associated with the best results. Since now we are using custom performance measures, the criterion that should be optimized must also be specified. In the call to train, we can use *metric* = "ROC" to do this.

The final model fit by using the train function would be:

```
## Use caret to fit a model and fine tune the fit ##
Caret_Tree <- train(Class ~ ., data = Class_Train, method ="rpart",
                    trControl=cv_control, metric = "ROC") #Change metric to ROC
```

Here the left hand side of the tilde (~) is the class attribute having two classes (PS and WS) and the right hand side has the all the attributes (.) to classify instances into the two classes. The *method = "rpart"* indicates that we use the *rpart* model to construct a tree (as we did in the last exercise). Then, *trControl = cv_control* calls the resampling method that we defined in the *trainControl* function. Finally, the ROC is our model performance metric (measures).

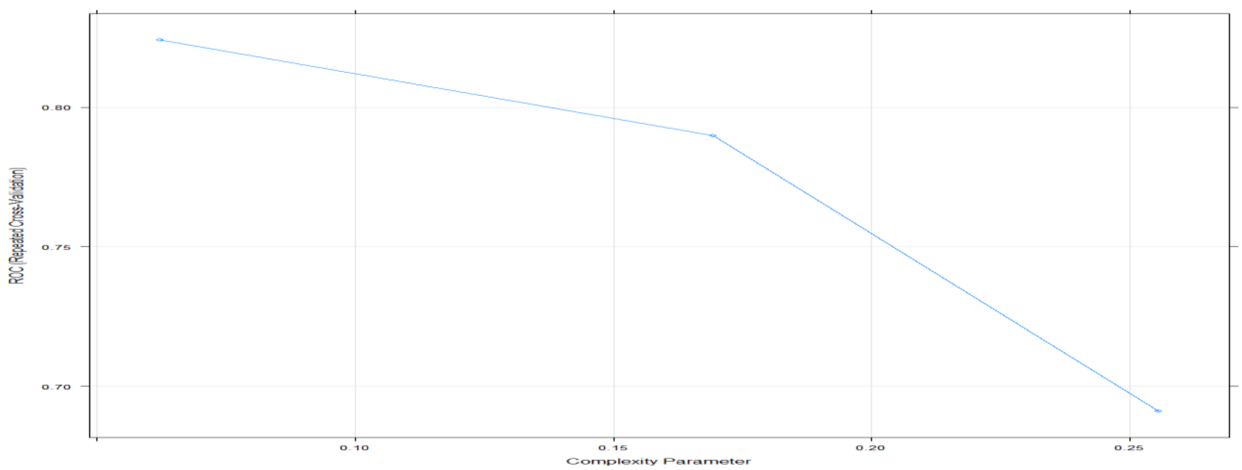
Let's see the detail in the resampling results across the tuning parameters (complexity parameter, CP):

```
> Caret_Tree
CART

1615 samples
  59 predictor
   2 classes: 'PS', 'WS'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 1453, 1453, 1453, 1453, 1455, 1453, ...
Resampling results across tuning parameters:

  cp          ROC        Sens       Spec
  0.06217617  0.8242400  0.8433159  0.6706191
  0.16925734  0.7898310  0.7146409  0.8037911
  0.25561313  0.6910162  0.7293907  0.6526417

ROC was used to select the optimal model using  the largest value.
The final value used for the model was cp = 0.06217617.
```

In this output, the results are the average resampled estimates of performance (ROC, Sensitivity, and Specificity). The note at the bottom tells us the optimal value of complexity parameter (CP =

0.06217617). Based on this value, a final decision tree model is fit to the data using this specification, and this is the model that is used to predict new datasets.

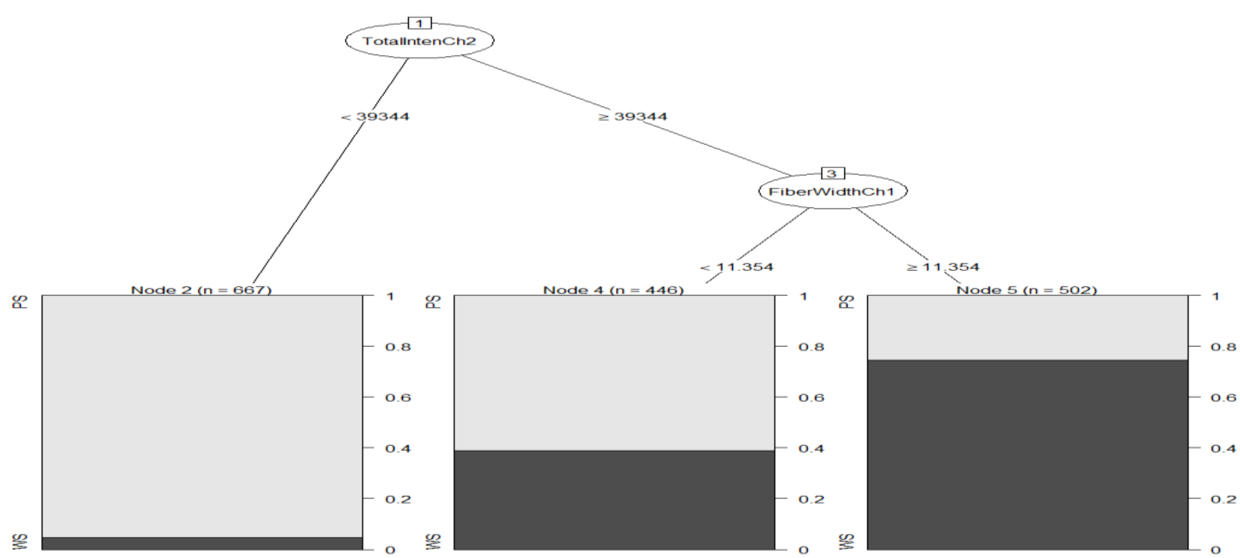
Let's see the performance of models over various levels of tuning parameter.



The resulting plot of our pruned tree is shown in the following figure:

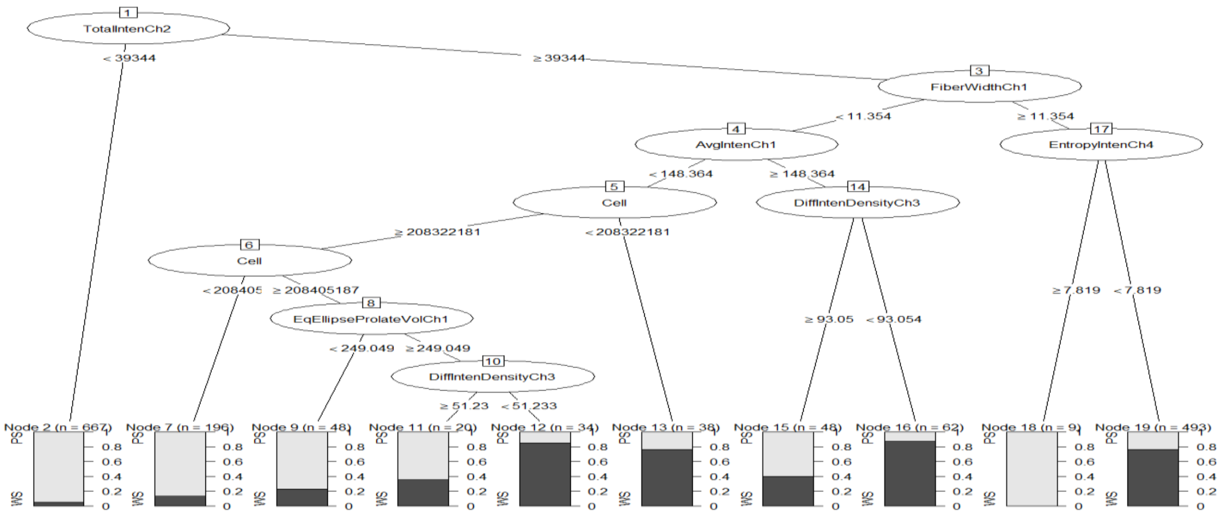
```
library(partykit)
plot(as.party(Caret_Tree$finalModel))
```

Decision Tree after Pruning



The below is the tree before pruning (presented in the last exercise):

Decision Tree before Pruning



As you can see, our pruned tree has only five nodes instead of having unnecessary multiple nodes.

Validate Using Testing Data

Next step is to predict the class membership with test data, and then present the confusion matrix for performance metrics of this model over the test dataset.
The predict() function will return a vector of predicted class values (type="class"); our class attribute have two classes (PS and WS).

```
CaretTree_predict <- predict(Caret_Tree, Class_Test)
```

Let's create a confusion matrix.

Confusion Matrix after Pruning

```
> confusionMatrix(CaretTree_predict, Class_Test$Class)
Confusion Matrix and Statistics

          Reference
Prediction  PS  WS
        PS 220  49
        WS  44  91

               Accuracy : 0.7698
                 95% CI : (0.72[illegible], 0.81)
    No Information Rate : 0.6535
    P-Value [Acc > NIR] : 2.594e-07

                  Kappa : 0.4874
 Mcnemar's Test P-Value : 0.6783

            Sensitivity : 0.8333
            Specificity : 0.6500
         Pos Pred Value : 0.8178
         Neg Pred Value : 0.6741
             Prevalence : 0.6535
         Detection Rate : 0.5446
   Detection Prevalence : 0.6658
      Balanced Accuracy : 0.7417

       'Positive' Class : PS
```

The below is the confusion matrix before pruning (presented in the last exercise):

Confusion Matrix before Pruning

```
> confusionMatrix(rpart_predict, Class_Test$Class)
Confusion Matrix and Statistics

          Reference
Prediction  PS  WS
        PS 203  23
        WS  61 117

               Accuracy : 0.7921
                 95% CI : (0.7492, 0.8306)
    No Information Rate : 0.6535
    P-Value [Acc > NIR] : 7.570e-10

                  Kappa : 0.5684
 Mcnemar's Test P-Value : 5.413e-05

            Sensitivity : 0.7689
            Specificity : 0.8357
         Pos Pred Value : 0.8982
         Neg Pred Value : 0.6573
             Prevalence : 0.6535
         Detection Rate : 0.5025
   Detection Prevalence : 0.5594
      Balanced Accuracy : 0.8023

       'Positive' Class : PS
```

Although the overall performance measures have been slightly decreased after pruning (see the Accuracy and Specificity), we have a shallower tree with fewer nodes which generally avoids the overfitting problem. In addition, the Sensitivity has been improved after pruning.