Decision Trees

Applied Machine Learning for Educational Data Science

true

11/09/2021

Contents

[Updated: Thu, Nov 11, 2021 - 16:17:40]

1. Regression Trees

We will again consider the toy dataset (N=20) we used before to predict a readability score from average word length and number of sentences.

```
readability_sub <- read.csv('https://raw.githubusercontent.com/uo-datasci-specialization/c4-ml-fall-202

readability_sub[,c('mean.wl','sents','target')]
```

```
    mean.wl sents      target
1  4.603659     7 -2.58590836
2  3.830688    23  0.45993224
3  4.180851    17 -1.07470758
```

```
4  4.015544     7 -1.81700402
5  4.686047     6 -1.81491744
6  4.211340    18 -0.94968236
7  4.025000    10 -0.12103065
8  4.443182     4 -2.82200582
9  4.089385     9 -0.74845172
10 4.156757    28  0.73948755
11 4.463277    15 -0.96218937
12 5.478261    10 -2.21514888
13 4.770492    10 -1.21845136
14 4.568966     8 -1.89544351
15 4.735751    19 -0.04101056
16 4.372340    15 -1.83716516
17 4.103448     6 -0.18818586
18 4.042857     6 -0.81739314
19 4.202703     7 -1.86307557
20 3.853535    19 -0.41630158
```

1.1. Basics of a regression tree

1.1.1. Tree Structure

Let's imagine a simple tree model to predict readability scores from average word length.

This model splits the sample into two pieces using a **split point** of 4.5 for the predictor variable. There are 14 observations with average word length is less than 4.5, and 6 observations with average word length is more than 4.5. The top of the tree model is called **root node**, and the R_1 and R_2 in this model are called **terminal nodes**. This model has two terminal nodes. There is a number assigned to each terminal node. These numbers are the average values for the target outcome (readability score) for those observations in that specific node. This can be symbolically shown as

$$\bar{Y_t} = \frac{1}{n_t} \sum_{i \epsilon R_t} Y_i,$$

where n_t is the number of observations in a terminal node, and R_t represents the set of observations that exist in the t_{th} node.

There is also a concept of **depth** of a tree. The root node counted as depth 0 and each split increases the depth of the tree by one. In this case, we can say this tree model has a depth of one.

We can increase the complexity of our tree model by splitting first node into two more nodes using a split point of 4.

Now, our model has a depth of two and a total three terminal nodes, and each terminal node is again assigned a number which is the average of target outcome for those observations in that node.

The tree model can have nodes from splitting another variable. For instance, the model below first splits the observations based on the average word length, and then based on the number of sentences yielding again a tree model with three nodes with a depth of two. The complexity of this tree model is same the complexity of the previous one. The only difference is that we have nodes from two variables instead of one.

A final example is another tree model with increasing complexity and having a depth of three and four nodes. It first splits observations based on whether or not the average word length is less than 4.5, then splits observations based on whether or not the average word length is less than 4, and finally splits observations based on whether or not the number of sentences is less than 11.

1.1.2. Predictions

Given a tree model, the predictions for the new observations are made by first

Suppose you developed a tree model and decided to use this model to make predictions for new observations. Let's assume our model is the below. How do we use this model to make predictions for new observations?

Suppose that there is a new reading passage we want to predict the readability score based on this tree model. This new reading passage has 20 sentences and the average word length is 3.5. To predict the readability score for this reading passage, we have to decide which node this passage would belong in this tree model. You can trace a path starting from the **root node** (top of the tree) and see what this reading passage will end up. As you can see below, this reading passage would be in the first node based on the observed values for these two predictors, and so we predict that the readability score for this new reading passage is equal to 0.022.

If we have another new reading passage with 5 sentences and an average word length of 4.4, then it would end up in the second node and we would predict the readability score as -1.197.

1.1.3. Loss function

When we fit a tree model, the algorithm decides what is the best split that minimizes the sum of squared errors. The sum of squared error from a tree model can be shown as

$$SSE = \sum_{t=1}^{T} \sum_{i \epsilon R_t} (Y_i - \hat{Y}_{R_t})^2,$$

where T is the total number of terminal nodes in the tree model, and $\hat{Y}_{R_t}$ is the prediction for the observations in the t^{th} node (average target outcome for those observations in the t^{th} node).

1.1.4. Growing a tree

Deciding a root node and then growing a tree model from that root node can become computationally exhaustive depending on the size of dataset and number of variables. In the most basic sence, the decision tree algorithm searches all variables designated as predictors in the dataset at all possible split points for these variables, calculates the SSE for all possible splits, and then finds the split that would reduce the amount of prediction error the most. The search continues by growing the tree model sequentially until there is no more split left that would give better predictions.

I will demonstrate the logic of this search process with the toy dataset (N=20) and two predictor variables: average word length and number of sentences. Before we start our search process, we should come up with a baseline SSE to decide whether or not any future split is going to improve our predictions. We can imagine that a model witn no split at all and using the mean of all 20 observations to predict the target outcome is the simplest baseline model. So, if we compute the sum of squared residuals assuming that we predict every observation with the mean of these 20 observations, that could be a baseline measure. As you see below, if we use the mean target score as our prediction for the 20 observations, then the sum of squared residuals would be 18.654. We will try to improve this number over the baseline by growing a tree model.

```
# average outcome

mu <- mean(readability_sub$target)

# SSE for baseline model

sum((readability_sub$target - mu)^2)
```

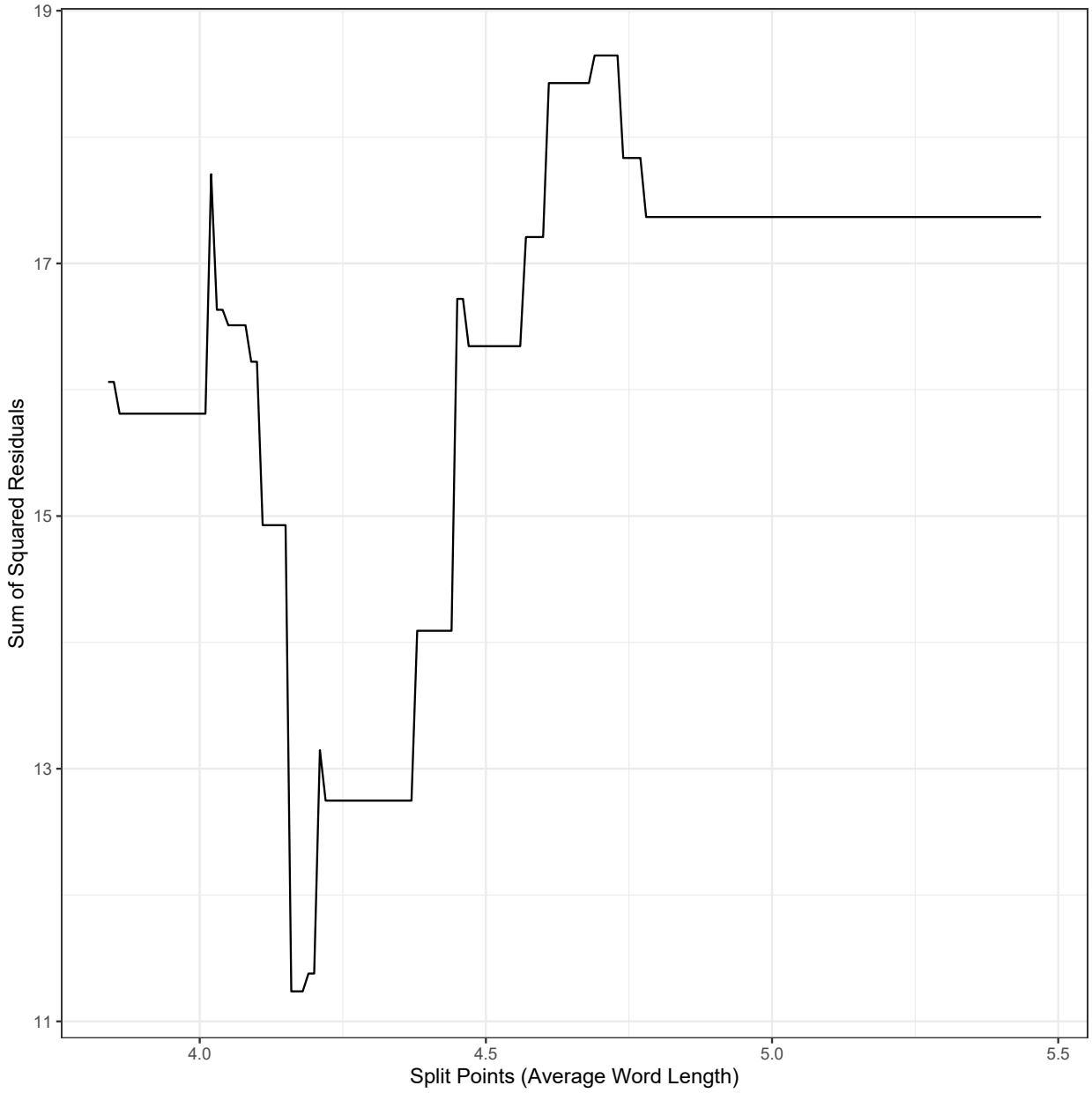
```
[1] 18.65414
```

Find the root node The first step in building the tree model is to find the root node. In this case, we have two candidates for the root node: average word length and number of sentenses. We want to know which variable should be the root node and what value we should use to split to improve the baseline SSE the most. This is how the process would look like.

1) Pick a split point
2) Divide 20 observations into two nodes based on the split point
3) Calculate the average target outcome within each node as predictions for the observations in that node
4) Calculate SSE within each node using the predictions from Step 3, and sum them all across two nodes.
5) Repeat Step 1 - Step 4 for every possible split point, and find the best split point with the minimum SSE
6) Repeat Step 1 - Step 5 for every possible predictor.

The code below implements Step 1 - Step 5 for the predictor **average word length**.

```
splits1 <- seq(3.84,5.47,0.01)

# For every split point, partition the observations into two groups and then
# make predictions based on the partitioning. Save the SSE in an empy object

sse1 <- c()

for(i in 1:length(splits1)){

  gr1 <- which(readability_sub$mean.wl <= splits1[i])
  gr2 <- which(readability_sub$mean.wl > splits1[i])

  pr1 <- mean(readability_sub[gr1,]$target)
  pr2 <- mean(readability_sub[gr2,]$target)

  sse1[i] = sum((readability_sub[gr1,]$target - pr1)^2) +
            sum((readability_sub[gr2,]$target - pr2)^2)
}

ggplot()+
  geom_line(aes(x=splits1,y=sse1))+
  xlab('Split Points (Average Word Length)')+
  ylab('Sum of Squared Residuals')+
  theme_bw()
```




```
splits1[which.min(sse1)]
```

```
[1] 4.16
```

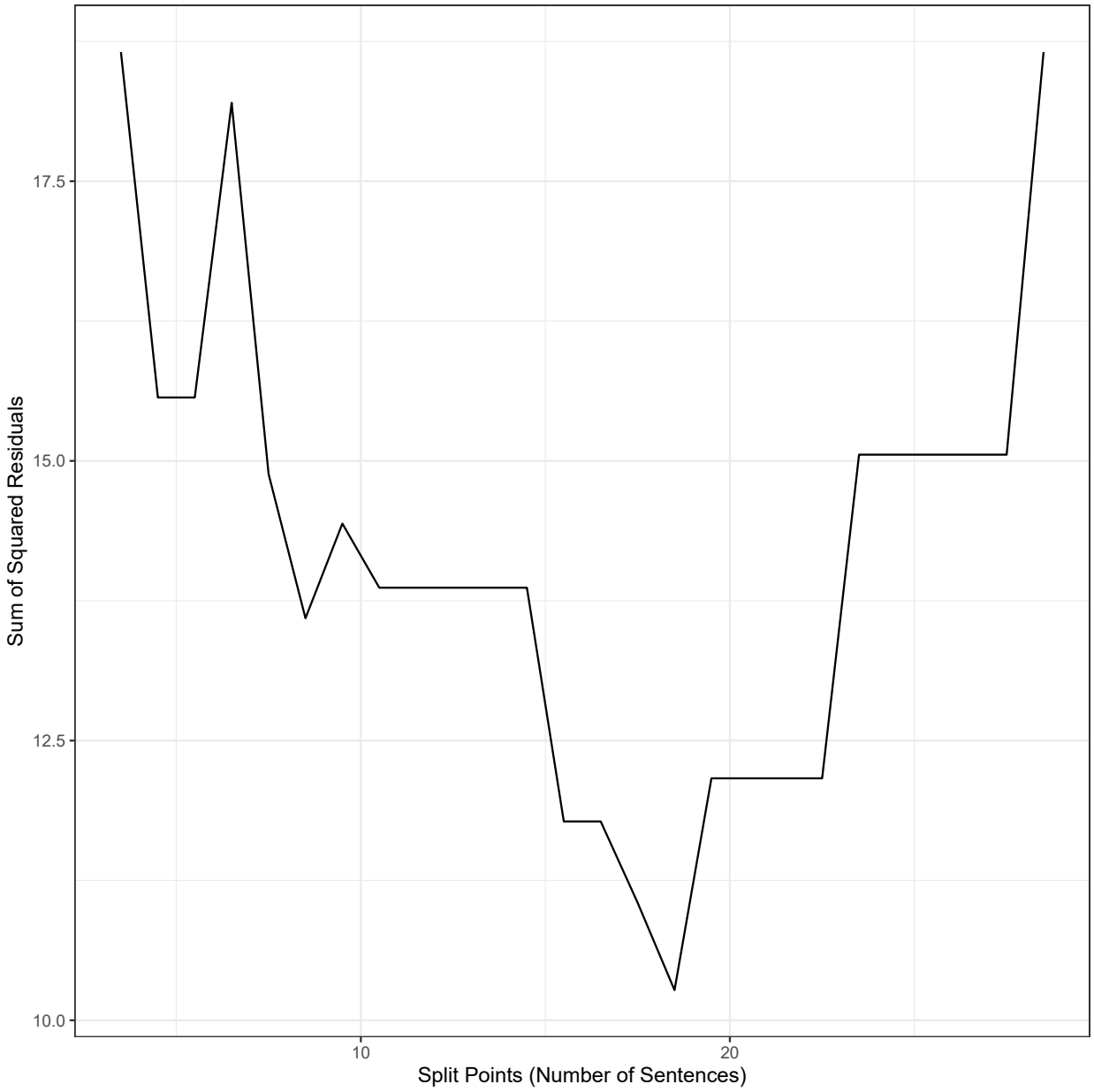
```
min(sse1)
```

```
[1] 11.23762
```

The search process indicates that the best split point for the average word length is 4.16, and if we divide the observations into two nodes based on the average word length using the split point 4.16, SSE would be equal to 11.24, a significant improvement over the baseline model.

We will now repeat the same process for the number of sentences. The code below implements Step 1 - Step 5 for the predictor **number of sentences**.

```r
splits2 <- seq(3.5,28.5,1)

# For every split point, partition the observations into two groups and then
# make predictions based on the partitioning. Save the SSE in an empy object

sse2 <- c()

for(i in 1:length(splits2)){

  gr1 <- which(readability_sub$sents <= splits2[i])
  gr2 <- which(readability_sub$sents > splits2[i])

  pr1 <- mean(readability_sub[gr1,]$target)
  pr2 <- mean(readability_sub[gr2,]$target)

  sse2[i] = sum((readability_sub[gr1,]$target - pr1)^2) +
            sum((readability_sub[gr2,]$target - pr2)^2)
}

ggplot()+
  geom_line(aes(x=splits2,y=sse2))+
  xlab('Split Points (Number of Sentences)')+
  ylab('Sum of Squared Residuals')+
  theme_bw()
```




```
splits2[which.min(sse2)]
```

```
[1] 18.5
```

```
min(sse2)
```

```
[1] 10.26953
```

The search process indicates that the best split point for the number of sentences is 1, and if we divide the observations into two nodes based on the number of sentences using the split point 18.5, SSE would be equal to 10.27, also a significant improvement over the baseline model.

We have two choices for the **root node** (because we assume we only have two predictors):

- Average Word Length, Best split point = 4.16, SSE = 11.24
- Number of Sentences, Best split point = 18.5, SSE = 10.27

Both options provide a significant improvement in our predictions over the baseline (SSE = 18.654), but one is better than the other. Therefore, our final decision to start growing our tree is to pick number of sentences as our **root node** and split it at 8.5. Our tree model starts growing!

Adding depth with terminal nodes Now, we have to decide if we should add another split to either one of these two nodes. There are now four possible split scenarios.

1) For first terminal node on the left, we can split the observations based on average word length.
2) For first terminal node on the left, we can split the observations based on number of sentences.
3) For second terminal node on the right, we can split the observations based on average word length.
4) For second terminal node on the right, we can split the observations based on number of sentences.

For each one of these scenarios, we can implement Step 1 - Step 4, and identify the best split point and what SSE that split yields. The code below implements these steps for all 4 scenarios.

Scenario 1:

```
node1 <- c(1,3,4,5,6,7,8,9,11,12,13,14,16,17,18,19)
node2 <- c(2,10,15,20)

# Splits based on average word length for node 1

splits1 <- seq(4.01,5.47,0.01)

sse1 <- c()

for(i in 1:length(splits1)){

  gr1 <- which(readability_sub[node1,]$mean.wl <= splits1[i])
  gr2 <- which(readability_sub[node1,]$mean.wl > splits1[i])
  gr3 <- node2

  pr1 <- mean(readability_sub[node1[gr1],]$target)
  pr2 <- mean(readability_sub[node1[gr2],]$target)
  pr3 <- mean(readability_sub[node2,]$target)


  sse1[i] = sum((readability_sub[node1[gr1],]$target - pr1)^2) +
            sum((readability_sub[node1[gr2],]$target - pr2)^2) +
            sum((readability_sub[node2,]$target - pr3)^2)
}

splits1[which.min(sse1)]
```

```
[1] 4.19
```

```
min(sse1)
```

```
[1] 6.353205
```

Scenario 2:

```
# Splits based on number of sentences for node 1

splits2 <- seq(3.5,17.5,0.5)

sse2 <- c()

for(i in 1:length(splits2)){

  gr1 <- which(readability_sub[node1,]$sents <= splits2[i])
  gr2 <- which(readability_sub[node1,]$sents > splits2[i])
  gr3 <- node2

  pr1 <- mean(readability_sub[node1[gr1],]$target)
  pr2 <- mean(readability_sub[node1[gr2],]$target)
  pr3 <- mean(readability_sub[node2,]$target)


  sse2[i] = sum((readability_sub[node1[gr1],]$target - pr1)^2) +
            sum((readability_sub[node1[gr2],]$target - pr2)^2) +
            sum((readability_sub[node2,]$target - pr3)^2)
}

splits2[which.min(sse2)]
```

```
[1] 4
```

```
min(sse2)
```

```
[1] 8.212086
```

Scenario 3:

```
# Splits based on average word length for node 2

splits3 <- c(3.84,3.86,4.16)

sse3 <- c()

for(i in 1:length(splits3)){

  gr1 <- which(readability_sub[node2,]$mean.wl <= splits3[i])
  gr2 <- which(readability_sub[node2,]$mean.wl > splits3[i])
  gr3 <- node2

  pr1 <- mean(readability_sub[node2[gr1],]$target)
```

```
  pr2 <- mean(readability_sub[node2[gr2],]$target)
  pr3 <- mean(readability_sub[node1,]$target)


  sse3[i] = sum((readability_sub[node2[gr1],]$target - pr1)^2) +
            sum((readability_sub[node2[gr2],]$target - pr2)^2) +
            sum((readability_sub[node1,]$target - pr3)^2)
}

splits3[which.min(sse3)]
```

```
[1] 3.86
```

```
min(sse3)
```

```
[1] 10.16233
```

Scenario 4:

```
# Splits based on number of sentences for node 2

splits4 <- seq(19.5,23.5)

sse4 <- c()

for(i in 1:length(splits4)){

  gr1 <- which(readability_sub[node2,]$sents <= splits4[i])
  gr2 <- which(readability_sub[node2,]$sents > splits4[i])
  gr3 <- node1

  pr1 <- mean(readability_sub[node2[gr1],]$target)
  pr2 <- mean(readability_sub[node2[gr2],]$target)
  pr3 <- mean(readability_sub[node1,]$target)


  sse4[i] = sum((readability_sub[node2[gr1],]$target - pr1)^2) +
            sum((readability_sub[node2[gr2],]$target - pr2)^2) +
            sum((readability_sub[node1,]$target - pr3)^2)
}

splits4[which.min(sse4)]
```

```
[1] 19.5
```

```
min(sse4)
```

```
[1] 9.583344
```

Based on our search, the following splits provided the least SSEs:

1) 1st terminal node, Split variable: Average Word Length, Split Point: 4.19, SSE = 6.35

2) 1st terminal node, Split variable: Number of Sentences, Split Point: 4, SSE = 8.21

3) 2nd terminal node, Split variable: Average Word Length, Split Point: 3.86, SSE = 10.16

4) 2nd terminal node, Split variable: Number of Sentences, Split Point: 19.5, SSE = 9.58

We decide to continue with Scenario 1 because it yields the minimum SSE. Therefore, our tree model now looks like this.

Termination of Growth The search process continues until a certain type of criteria is met to stop the algorithm. There may be a number of conditions where we may constrain the growth and force the algorithm to stop searching for additional growth in the tree model. Some of these are listed below:

- Minimizing SSE: the algorithm stops when there is no potential split in any of the existing nodes that would reduce the sum of squared errors.
- Minimum number of observations to split: the algorithm does not attempt to split a node unless there is a certain number of observation in the node.
- Minimum number of observations in a node: the algorithm does not consider splits unless there is a certain number of observation in the nodes.
- Maximum depth: The algorithm stops searching when the tree reaches to a certain depth.

1.1.5. Pruning a tree

Overfitting and underfitting also occurs as one develops a tree model. When the depth of a tree becomes unnecessarily large, there is a risk of overfitting (increased variance in model predictions across samples, less generalizable). On the other hand, when the depth of a tree is small, there is a risk of underfitting (increased bias in model predictions, underperforming model).

In order to balance the model variance and model bias, we considered penalizing large coefficients in regression models yielding different types of penalty terms on the regression coefficients. There is a similar approach that can be applied for tree models. In the context of tree models, the loss function with a penalty term can be specified as

$$SSE = \sum_{t=1}^{T} \sum_{i \epsilon R_t} (Y_i - \hat{Y}_{R_t})^2 + \alpha T.$$

The penalty term α is known as the **cost complexity** parameter. The product term αT increases as the number of terminal nodes increases in the model, so this term penalizes increasing complexity of the tree model. By fine-tuning the value of α through cross-validation, we can find a balance between model variance and bias as we did for regression models. The process is called pruning because the terminal nodes from the tree are eliminated in a nested way for increasing levels of α.

1.2. Growing trees with the `rpart()` function

1.2.1. A simple example

We can develop a tree model using the `rpart()` function fromt the `rpart` package. Also, we will use the `fancyRpartPlot` function from the `rattle` package to get a nice plot of a tree model.

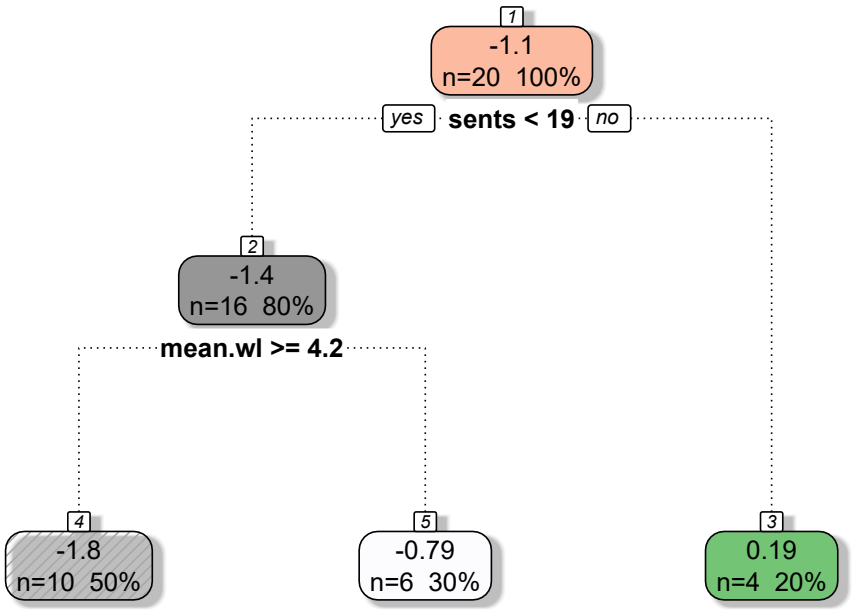
```
#install.packages('rpart')

#install.packages('rattle')

require(rpart)
require(rattle)

#?rpart
#?fancyRpartPlot
```

Let's first replicate our search above to build a tree model predicting the readability scores from average word length and number of sentences for the toy dataset.

```
dt <- rpart(formula = target ~ mean.wl + sents,
            data    = readability_sub,
            method  = "anova",
            control = list(minsplit=5,
                           cp=0,
                           minbucket = 2,
                           maxdepth = 2)
            )
```

- The `formula` argument works the same way as in the regression models. The variable name on the left side of `~` represents the outcome variable. The variable names on the right side of `~` represent the predictor variables.
- The `data` argument provides the name of the dataset to find these variables specified in the formula.
- The `method` argument is set to `anova` to indicate that the outcome is a continuous variable and this is a regression problem.
- The `control` argument is a list object with a number of settings to be used during the tree building process. For more information, check `?rpart.control`. You can accept the default values by not specifying it at all. Here, I specified a few important ones. `minsplit=5` forces algorithm not to attempt to split a node unless there is at least five observations. `minbucket=2` forces algorithm to have at least two observations for any node. `maxdepth = 2` forces algorithm to stop when the depth of a tree reaches to 2. `cp=0` indicates that we do not want to apply any penalty term ($\lambda = 0$) during the model building process.

```
fancyRpartPlot(dt,type=2,sub='')
```




```
  # ?prp
  # check for a lot of settings for modifying this plot
```

As you can see, we got the exact same tree model we built in our own search process in the earlier section. You can also ask more specific information about the model building process by running the `summary()` function.

```
summary(dt)
```

```
Call:
rpart(formula = target ~ mean.wl + sents, data = readability_sub,
    method = "anova", control = list(minsplit = 5, cp = 0, minbucket = 2,
        maxdepth = 2))
```

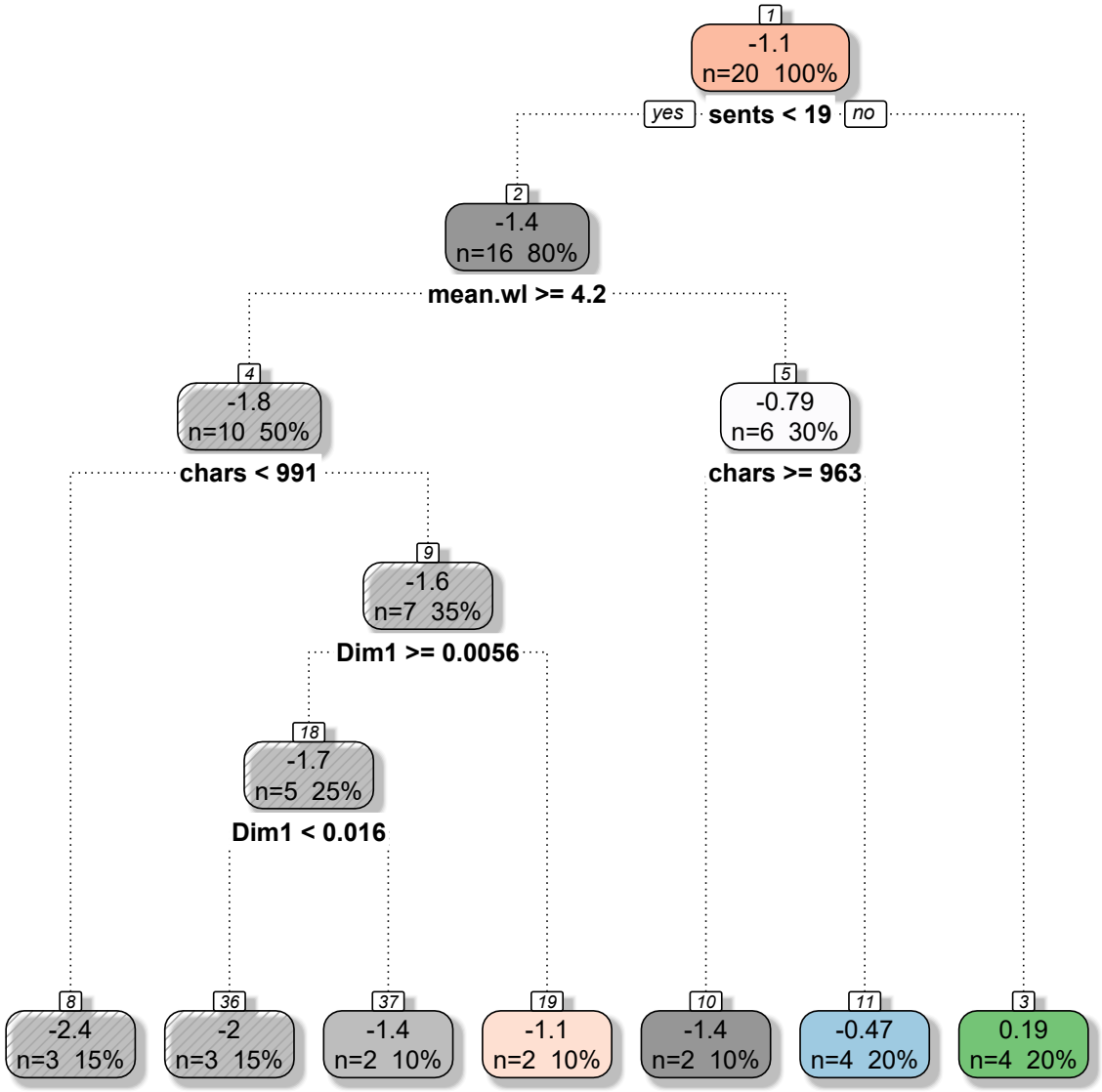
```
  n= 20

        CP nsplit rel error    xerror      xstd
1 0.4494768      0 1.0000000 1.0954180 0.2654693
2 0.2099443      1 0.5505232 1.0846046 0.2539360
3 0.0000000      2 0.3405789 0.5940115 0.1254369

Variable importance
  sents mean.wl
     51      49

Node number 1: 20 observations,    complexity param=0.4494768
  mean=-1.109433, MSE=0.9327068
  left son=2 (16 obs) right son=3 (4 obs)
  Primary splits:
      sents   < 18.5     to the left,  improve=0.4494768, (0 missing)
      mean.wl < 4.168804 to the right, improve=0.3975804, (0 missing)
  Surrogate splits:
      mean.wl < 3.93454  to the right, agree=0.9, adj=0.5, (0 split)

Node number 2: 16 observations,    complexity param=0.2099443
  mean=-1.433173, MSE=0.5921154
  left son=4 (10 obs) right son=5 (6 obs)
  Primary splits:
      mean.wl < 4.191777 to the right, improve=0.4133832, (0 missing)
      sents   < 8.5      to the left,  improve=0.1443139, (0 missing)

Node number 3: 4 observations
  mean=0.1855269, MSE=0.1989219

Node number 4: 10 observations
  mean=-1.816399, MSE=0.3609665

Node number 5: 6 observations
  mean=-0.7944622, MSE=0.3246422
```

1.2.2. Tree pruning by increasing the complexity parameter

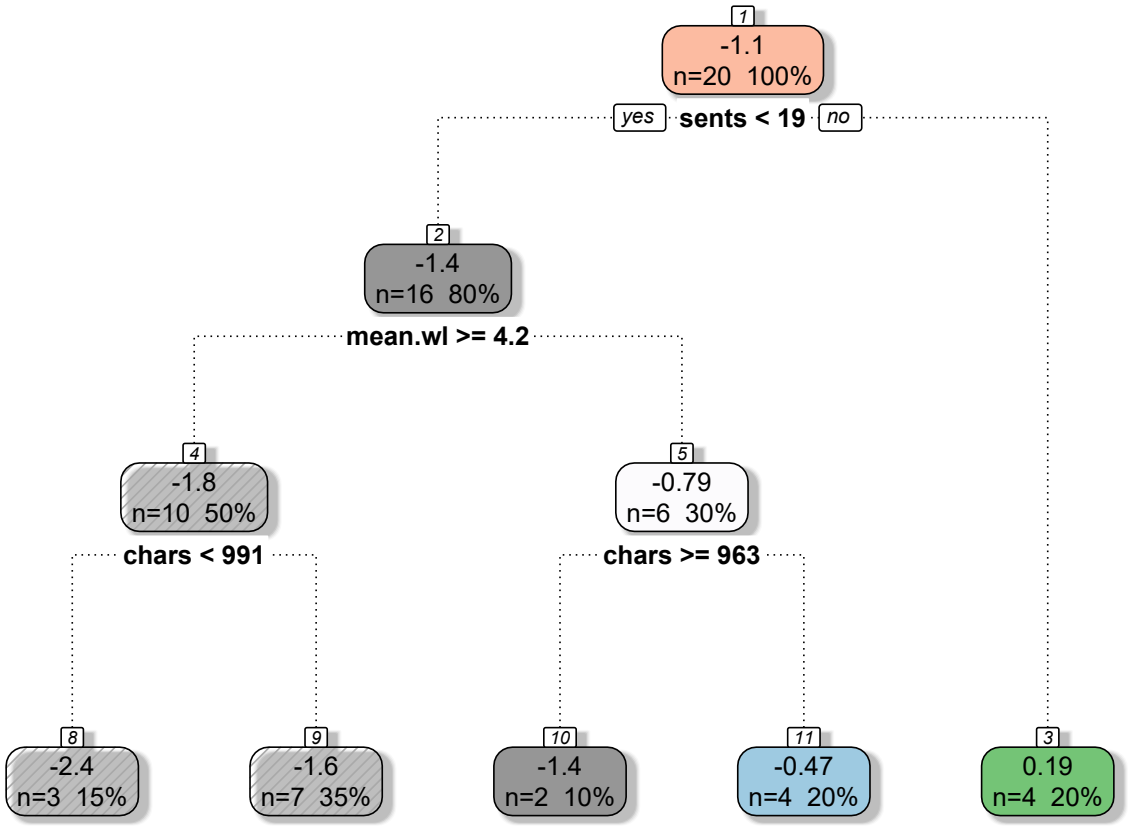
We will now expand the model by adding more predictors to be considered for the tree model. Suppose that we now have 6 different predictors to be considered for the tree model. We will provide the same settings with 6 predictors except the complexity parameter. We will fit three models by setting the complexity parameter at 0, 0.05, and 0.1 to see what happens to the tree model as we increase the complexity parameter.

```
dt <- rpart(formula = target ~ mean.wl + sents + chars + Dim1 + Dim10 + Dim300,
            data    = readability_sub,
            method  = "anova",
            control = list(minsplit=5,
                           cp=0,
                           minbucket = 2,
                           maxdepth = 5)
            )
```

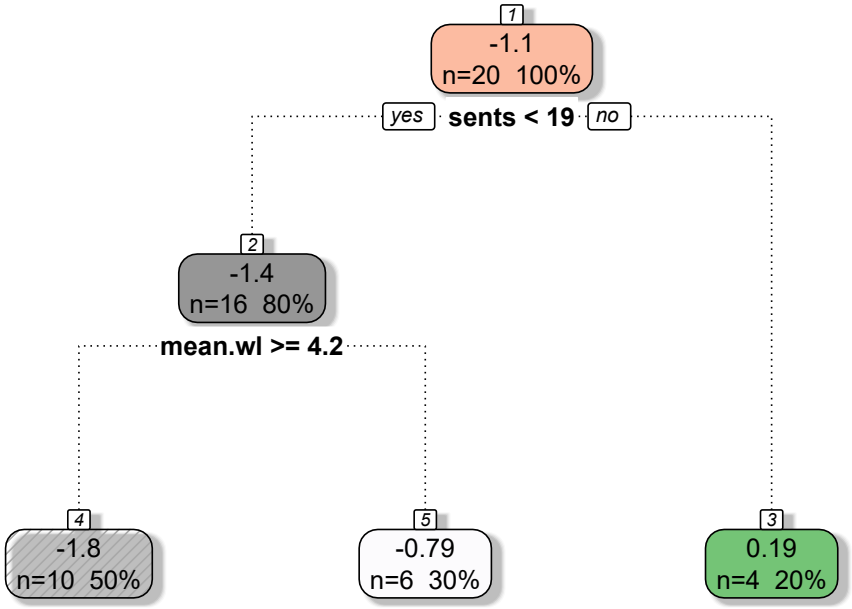
```
fancyRpartPlot(dt,type=2,sub='')
```




```
dt <- rpart(formula = target ~ mean.wl + sents + chars + Dim1 + Dim10 + Dim300,
            data    = readability_sub,
            method  = "anova",
            control = list(minsplit=5,
                           cp=0.05,
                           minbucket = 2,
                           maxdepth = 5)
            )


fancyRpartPlot(dt,type=2,sub='')
```




```
dt <- rpart(formula = target ~ mean.wl + sents + chars + Dim1 + Dim10 + Dim300,
            data    = readability_sub,
            method  = "anova",
            control = list(minsplit=5,
                           cp=0.1,
                           minbucket = 2,
                           maxdepth = 5)
            )


fancyRpartPlot(dt,type=2,sub='')
```



1.3. Training a tree model with `caret::train()`

`rpart` is also available in the caret package that provides user-friendly interface for hyperparameter tuning with cross validation. First, let's check the hyperparameters available to tune for the `rpart` method.

```
require(caret)

getModelInfo()$rpart$parameters
```

```
  parameter   class                label
1        cp numeric Complexity Parameter
```

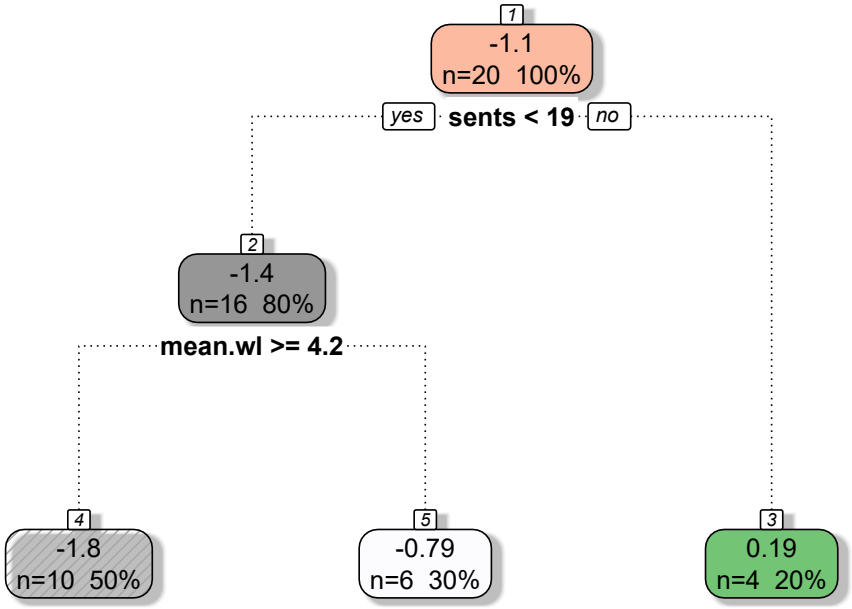
It seems that the `caret::train()` only allows tuning the complexity parameter (α). Now, let's replicate the same examples with the caret package.

```
# Specify the grid for cp
# fix it to 0, so we are not tuning it

grid <- data.frame(cp=0)

# Cross-validation settings
# method=none, no cross validation

cv <- trainControl(method = "none")


fit_rpart <- caret::train(x         = readability_sub[,c('mean.wl','sents')],
                          y         = readability_sub$target,
                          method    = 'rpart',
                          tuneGrid  = grid,
                          trControl = cv,
                          control   = list(minsplit=5,
                                           minbucket = 2,
                                           maxdepth = 2))

fit_rpart$finalModel
```

```
n= 20

node), split, n, deviance, yval
      * denotes terminal node

1) root 20 18.6541400 -1.1094330
  2) sents< 18.5 16  9.4738470 -1.4331730
    4) mean.wl>=4.191777 10  3.6096650 -1.8163990 *
    5) mean.wl< 4.191777 6  1.9478530 -0.7944622 *
  3) sents>=18.5 4  0.7956874  0.1855269 *
```

This provides the same results we obtained before. If you would like to plot the tree model obtained from the `caret::train()` function, you can still use the `fancyRpartPlot()` function.

```
fancyRpartPlot(fit_rpart$finalModel,type=2,sub='')
```



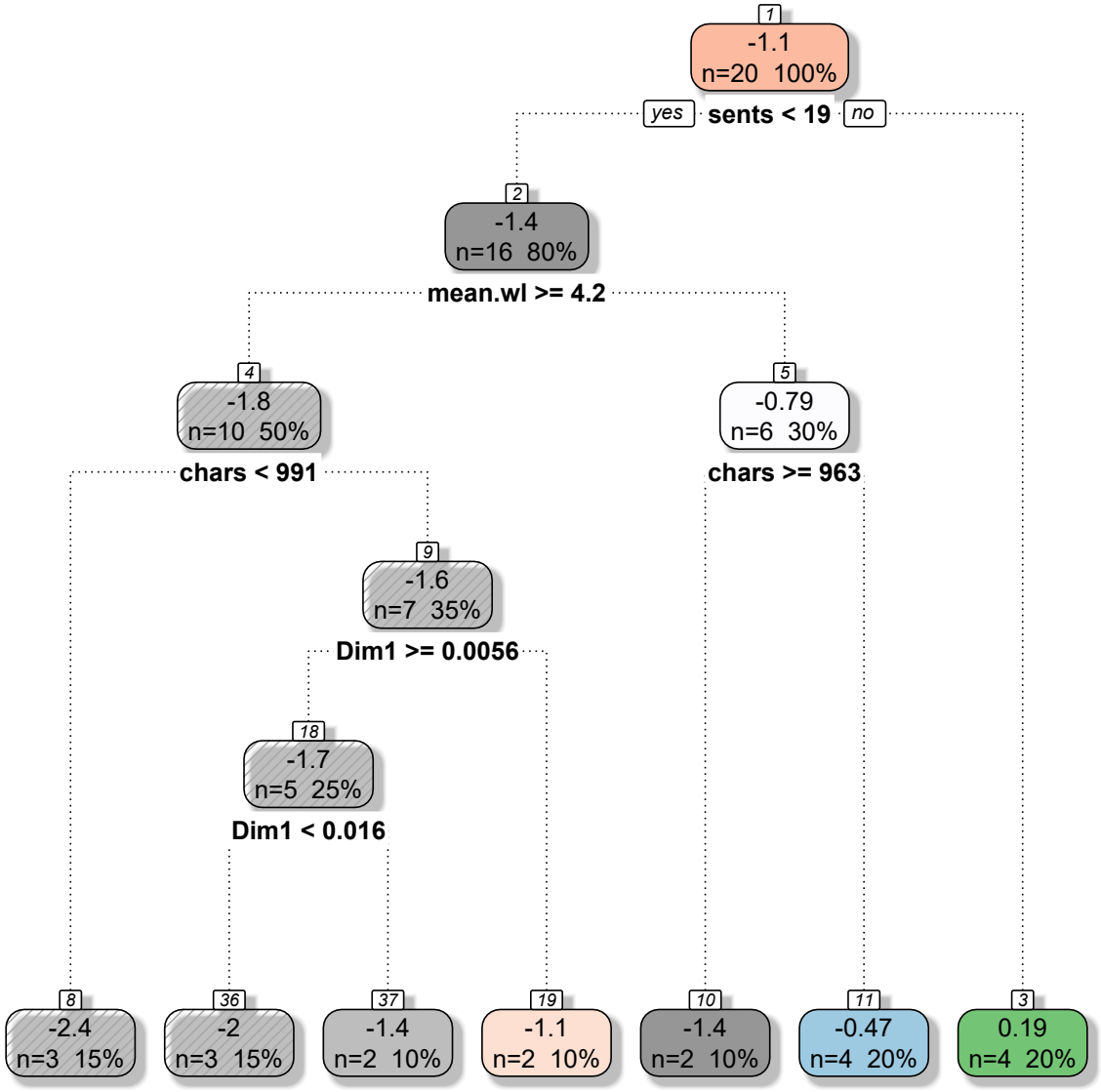
Now, let's replicate the second example.

```
# Specify the grid for cp

grid <- data.frame(cp=0)

# Cross-validation settings

cv <- trainControl(method = "none")

fit_rpart <- caret::train(x         = readability_sub[,c('mean.wl','sents','chars',
                                                         'Dim1','Dim10','Dim300')],
                          y         = readability_sub$target,
                          method    = 'rpart',
```

```
                          tuneGrid  = grid,
                          trControl = cv,
                          control   = list(minsplit=5,
                                           minbucket = 2,
                                           maxdepth = 5))

fit_rpart$finalModel
```

```
n= 20

node), split, n, deviance, yval
      * denotes terminal node

 1) root 20 18.65414000 -1.1094330
   2) sents< 18.5 16  9.47384700 -1.4331730
     4) mean.wl>=4.191777 10  3.60966500 -1.8163990
       8) chars< 990.5 3  0.49925880 -2.4236630 *
       9) chars>=990.5 7  1.52996200 -1.5561430
        18) Dim1>=0.005642436 5  0.86984860 -1.7449730
          36) Dim1< 0.0157772 3  0.08282653 -1.9825860 *
          37) Dim1>=0.0157772 2  0.36357260 -1.3885530 *
        19) Dim1< 0.005642436 2  0.03611839 -1.0840670 *
     5) mean.wl< 4.191777 6  1.94785300 -0.7944622
      10) chars>=963 2  0.27550200 -1.4458560 *
      11) chars< 963 4  0.39941010 -0.4687653 *
   3) sents>=18.5 4  0.79568740  0.1855269 *
```

```
fancyRpartPlot(fit_rpart$finalModel,type=2,sub='')
```



1.4. Predicting the Readability Scores

Now, we will import the whole readability dataset and train a tree model with all observations. Then, we will test the model performance on the test set. Most of the code below is identical to the code we used in earlier classes. The only difference is that we are now using `rpart` method in the caret package to train a tree model. I will also use parallel processing to reduce the model training time.

1. Import the dataset and prepare the dataset for model training

```
# Import the dataset

readability <- read.csv('https://raw.githubusercontent.com/uo-datasci-specialization/c4-ml-fall-2021/ma
```

```r
# Remove the variables with more than 80% missingness

require(finalfit)

missing_    <- ff_glimpse(readability)$Continuous
flag_na     <- which(as.numeric(missing_$missing_percent) > 80)
readability <- readability[,-flag_na]

# Write the recipe

require(recipes)

blueprint_readability <- recipe(x     = readability,
                                vars  = colnames(readability),
                                roles = c(rep('predictor',990),'outcome')) %>%
  step_zv(all_numeric()) %>%
  step_nzv(all_numeric()) %>%
  step_impute_mean(all_numeric()) %>%
  step_normalize(all_numeric_predictors()) %>%
  step_corr(all_numeric(),threshold=0.9)
```

2. Train/Test Split

```r
  set.seed(10152021)  # for reproducibility

  loc      <- sample(1:nrow(readability), round(nrow(readability) * 0.9))
  read_tr  <- readability[loc, ]
  read_te  <- readability[-loc, ]
```

3. Cross-validation settings

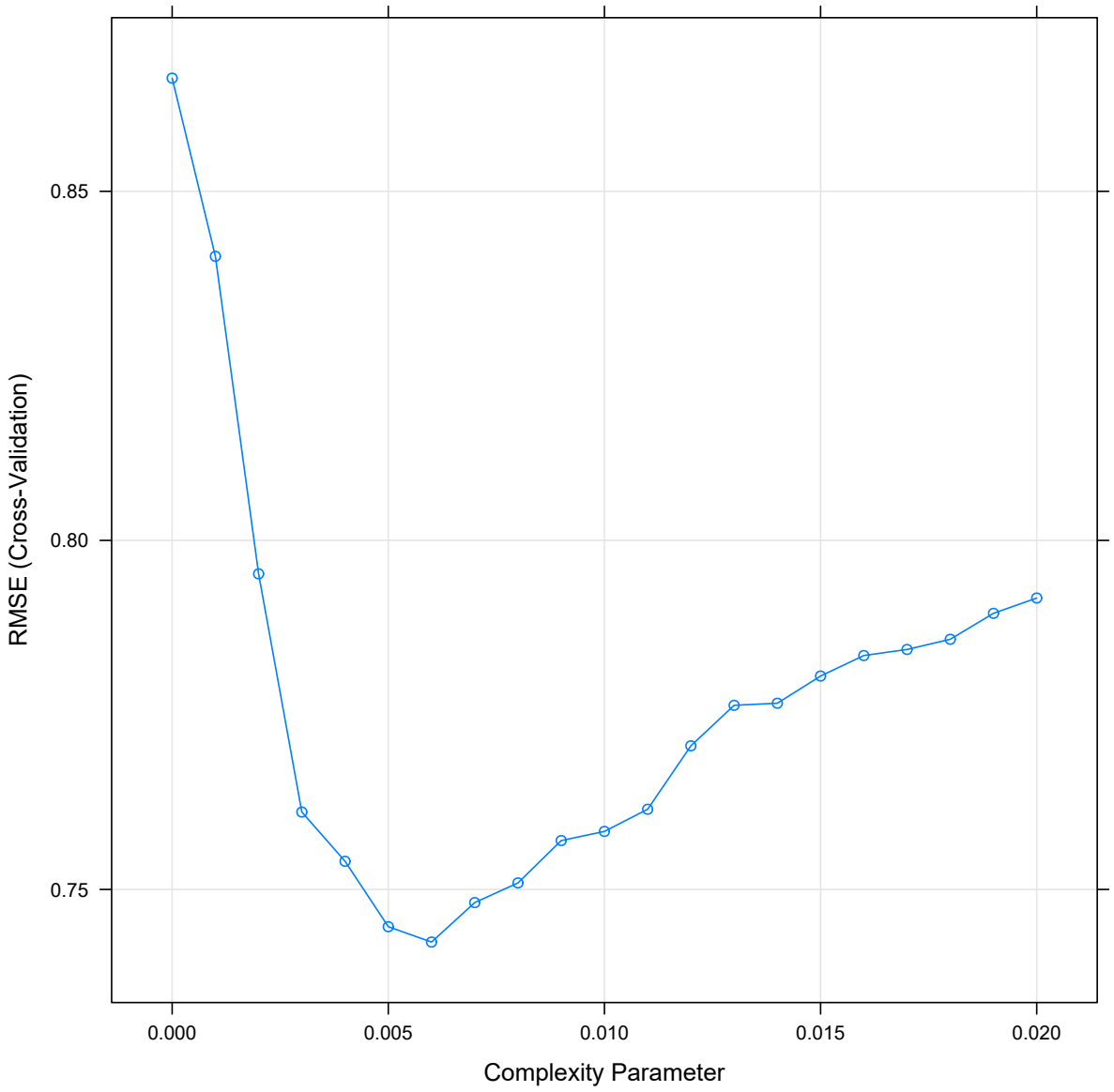
```r
# Create the row indices for 10-folds

    # Randomly shuffle the training data

      read_tr = read_tr[sample(nrow(read_tr)),]

    # Create 10 folds with equal size

      folds = cut(seq(1,nrow(read_tr)),breaks=10,labels=FALSE)

    # Create the list for each fold

      my.indices <- vector('list',10)
      for(i in 1:10){
        my.indices[[i]] <- which(folds!=i)
      }

# Cross-validation settings

  cv <- trainControl(method = "cv",
                     index  = my.indices)
```

4. Tuning grid for the complexity parameter

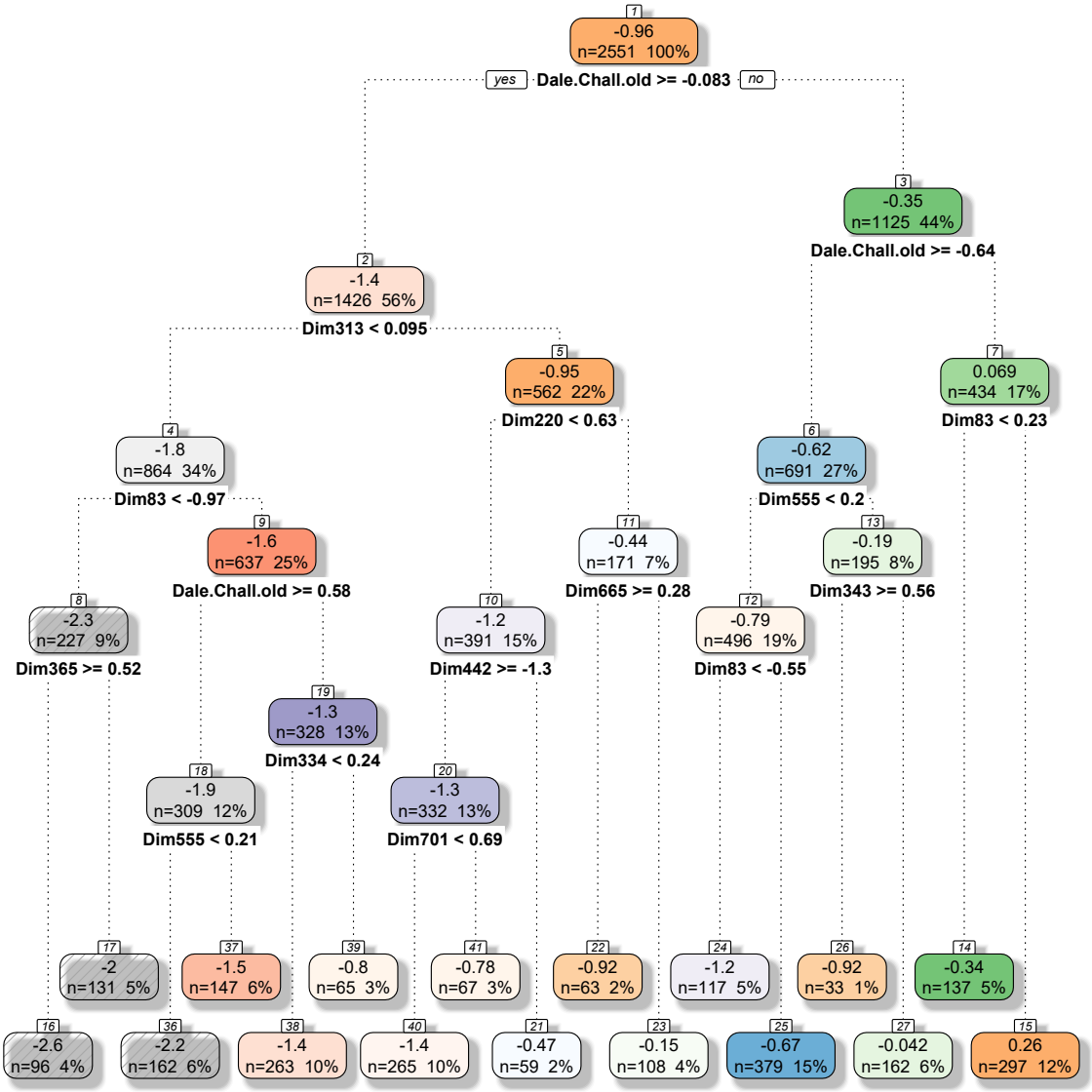
```
grid <- data.frame(cp=seq(0,0.02,.001))
```

5. Train the model

```
caret_rpart <- caret::train(blueprint_readability,
                            data      = read_tr,
                            method    = 'rpart',
                            tuneGrid  = grid,
                            trControl = cv,
                            control   = list(minsplit=20,
                                             minbucket = 2,
                                             maxdepth = 60))


plot(caret_rpart)

caret_rpart$bestTune

fancyRpartPlot(caret_rpart$finalModel,type=2,sub='')
```


RMSE (Cross-Validation)
0.85
0.80
0.75
0.000
0.005
0.010
0.015
0.020
Complexity Parameter

```
     cp
7 0.006
```



6. Evaluate the model performance on the test set

```
predicted_te <- predict(caret_rpart,newdata = read_te)

# R-square

cor(read_te$target,predicted_te)^2
```

```
[1] 0.4962518
```

```
# mean absolute error

mean(abs(read_te$target - predicted_te))
```

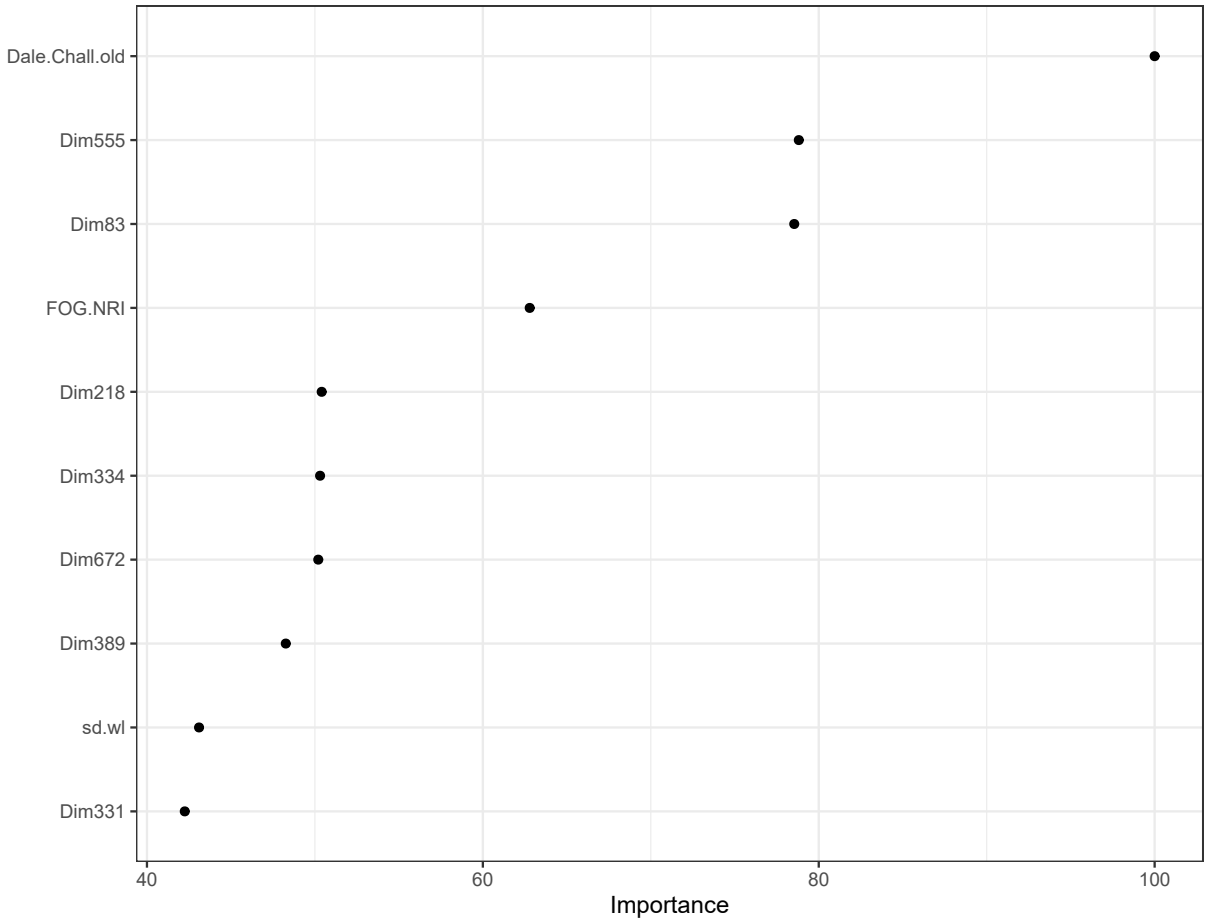
```
[1] 0.5774477
```

```
# root mean squared error

sqrt(mean((read_te$target - predicted_te)^2))
```

```
[1] 0.7294451
```

7. Variable Importance Plot

```
require(vip)

vip(caret_rpart,
    num_features = 10,
    geom = "point") +
  theme_bw()
```



8. Comparison of model performance

	R-square	MAE	RMSE
Linear Regression	0.644	0.522	0.644
Ridge Regression	0.727	0.435	0.536
Lasso Regression	0.725	0.434	0.538
KNN	0.623	0.500	0.629
Decision Tree	0.497	0.577	0.729

2. Classification Trees (Predicting Recidivism)

The classification trees are very similar to the regression trees. The only difference is that the loss function and the metric used while selecting the best split is different while growing the tree model. A more special metric (e.g., gini index or entropy) is used to decide the best split that improves the accuracy of predictions. This document provides a detailed technical discussion of how `rpart` builds a regression tree.

This section, we will implement the decision tree algorithm to the Recidivism dataset using the `caret::train()` function.

1. Import the dataset and prepare the dataset for model training

```
# Import data

recidivism <- read.csv('https://raw.githubusercontent.com/uo-datasci-specialization/c4-ml-fall-2021/mai

# Write the recipe


  # List of variable types

  outcome <- c('Recidivism_Arrest_Year2')

  categorical <- c('Residence_PUMA',
                   'Prison_Offense',
                   'Age_at_Release',
                   'Supervision_Level_First',
                   'Education_Level',
                   'Prison_Years',
                   'Gender',
                   'Race',
                   'Gang_Affiliated',
                   'Prior_Arrest_Episodes_DVCharges',
                   'Prior_Arrest_Episodes_GunCharges',
                   'Prior_Conviction_Episodes_Viol',
                   'Prior_Conviction_Episodes_PPViolationCharges',
                   'Prior_Conviction_Episodes_DomesticViolenceCharges',
                   'Prior_Conviction_Episodes_GunCharges',
                   'Prior_Revocations_Parole',
                   'Prior_Revocations_Probation',
                   'Condition_MH_SA',
                   'Condition_Cog_Ed',
                   'Condition_Other',
                   'Violations_ElectronicMonitoring',
                   'Violations_Instruction',
```

```
                 'Violations_FailToReport',
                 'Violations_MoveWithoutPermission',
                 'Employment_Exempt')

numeric   <- c('Supervision_Risk_Score_First',
               'Dependents',
               'Prior_Arrest_Episodes_Felony',
               'Prior_Arrest_Episodes_Misd',
               'Prior_Arrest_Episodes_Violent',
               'Prior_Arrest_Episodes_Property',
               'Prior_Arrest_Episodes_Drug',
               'Prior_Arrest_Episodes_PPViolationCharges',
               'Prior_Conviction_Episodes_Felony',
               'Prior_Conviction_Episodes_Misd',
               'Prior_Conviction_Episodes_Prop',
               'Prior_Conviction_Episodes_Drug',
               'Delinquency_Reports',
               'Program_Attendances',
               'Program_UnexcusedAbsences',
               'Residence_Changes',
               'Avg_Days_per_DrugTest',
               'Jobs_Per_Year')

props      <- c('DrugTests_THC_Positive',
                'DrugTests_Cocaine_Positive',
                'DrugTests_Meth_Positive',
                'DrugTests_Other_Positive',
                'Percent_Days_Employed')

# Convert all nominal, ordinal, and binary variables to factors
# Leave the rest as is

for(i in categorical){

  recidivism[,i] <- as.factor(recidivism[,i])

}

# For variables that represent proportions, add/substract a small number
# to 0s/1s for logit transformation

for(i in props){
  recidivism[,i] <- ifelse(recidivism[,i]==0,.0001,recidivism[,i])
  recidivism[,i] <- ifelse(recidivism[,i]==1,.9999,recidivism[,i])
}

# Blueprint for processing variables

require(recipes)

blueprint_recidivism <- recipe(x  = recidivism,
                               vars  = c(categorical,numeric,props,outcome),
                               roles = c(rep('predictor',48),'outcome')) %>%
```

```
    step_indicate_na(all_of(categorical),all_of(numeric),all_of(props)) %>%
    step_zv(all_numeric()) %>%
    step_impute_mean(all_of(numeric),all_of(props)) %>%
    step_impute_mode(all_of(categorical)) %>%
    step_logit(all_of(props)) %>%
    step_ns(all_of(numeric),all_of(props),deg_free=3) %>%
    step_normalize(paste0(numeric,'_ns_1'),
                   paste0(numeric,'_ns_2'),
                   paste0(numeric,'_ns_3'),
                   paste0(props,'_ns_1'),
                   paste0(props,'_ns_2'),
                   paste0(props,'_ns_3')) %>%
    step_dummy(all_of(categorical),one_hot=TRUE)  %>%
    step_num2factor(Recidivism_Arrest_Year2,
                    transform = function(x) x + 1,
                    levels=c('No','Yes'))
```

2. Train/Test Split

```
  loc <- which(recidivism$Training_Sample==1)

  recidivism_tr  <- recidivism[loc, ]
  recidivism_te  <- recidivism[-loc, ]
```

3. Cross-validation settings

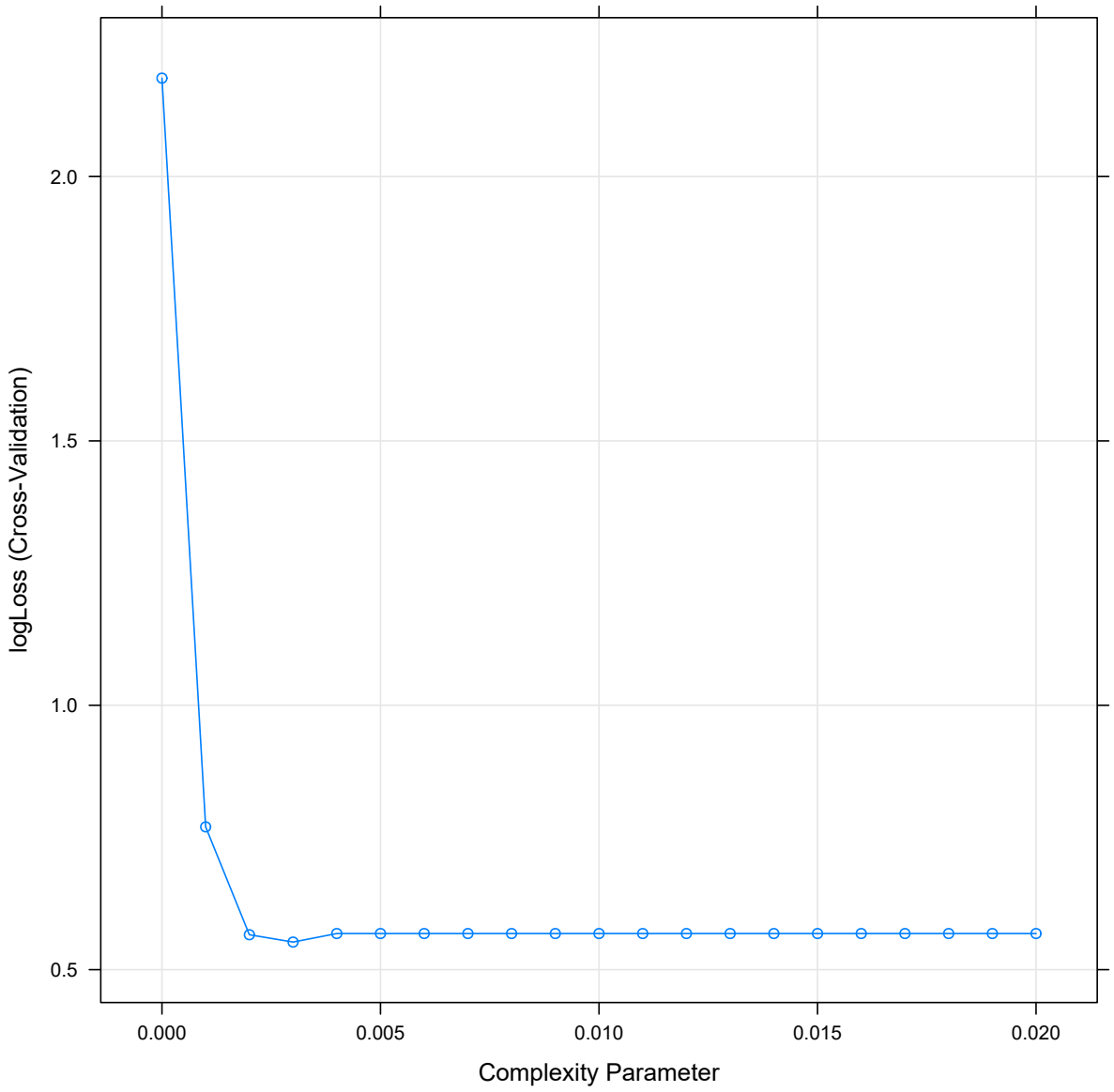
```
# Randomly shuffle the training data

  set.seed(10302021) # for reproducibility

  recidivism_tr = recidivism_tr[sample(nrow(recidivism_tr)),]

# Create 10 folds with equal size

  folds = cut(seq(1,nrow(recidivism_tr)),breaks=10,labels=FALSE)

# Create the list for each fold

  my.indices <- vector('list',10)
  for(i in 1:10){
    my.indices[[i]] <- which(folds!=i)
  }


# Cross-validation settings

  cv <- trainControl(method = "cv",
                     index  = my.indices,
                     classProbs = TRUE,
                     summaryFunction = mnLogLoss)
```

4. Tuning grid for the complexity parameter

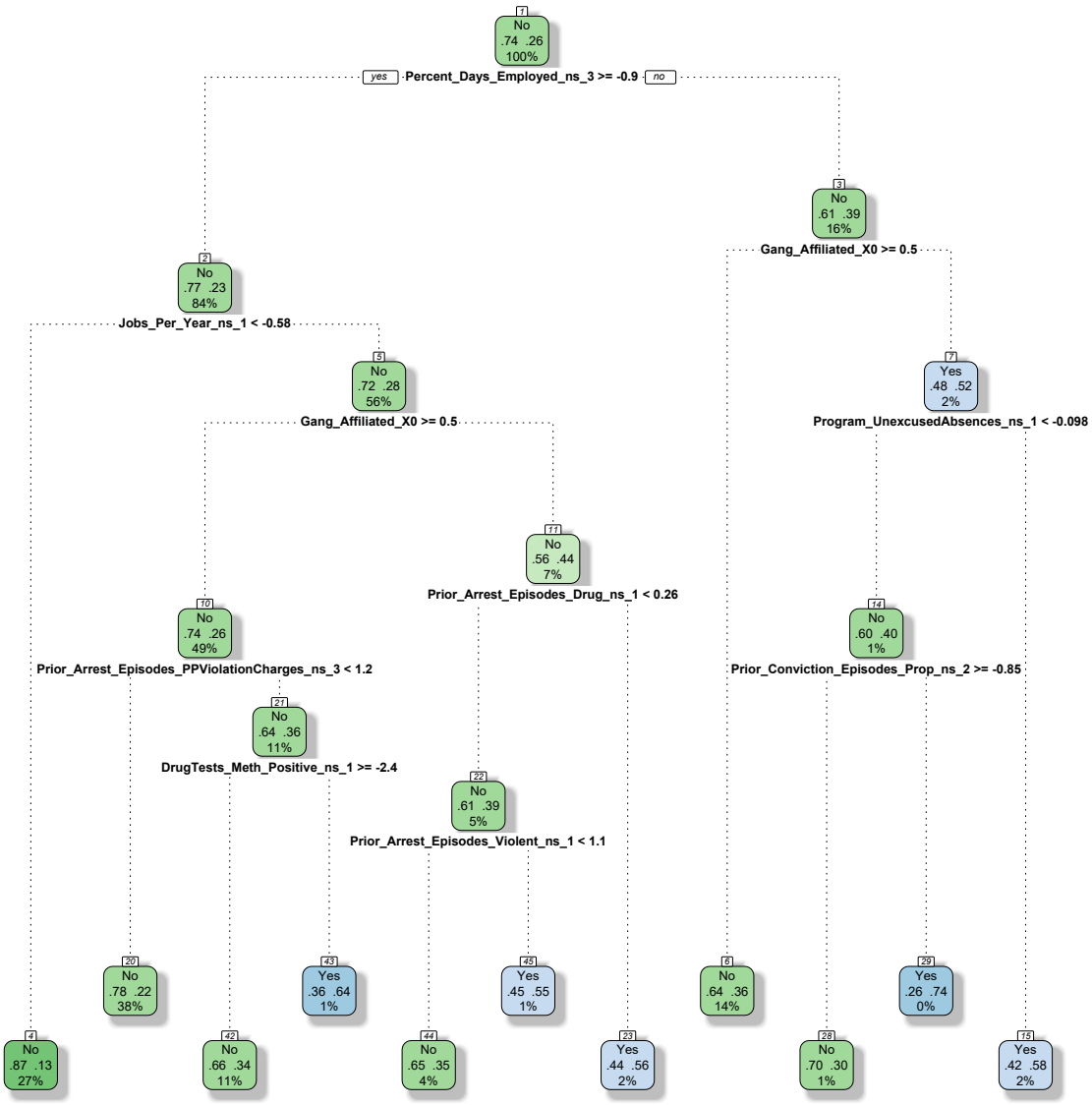
```
grid <- data.frame(cp=seq(0,0.02,.001))
```

5. Train the model

```
caret_rpart2 <- caret::train(blueprint_recidivism,
                             data      = recidivism_tr,
                             method    = 'rpart',
                             tuneGrid  = grid,
                             trControl = cv,
                             metric    = 'logLoss',
                             control   = list(minsplit=20,
                                              minbucket = 2,
                                              maxdepth = 60))


plot(caret_rpart2)

caret_rpart2$bestTune

fancyRpartPlot(caret_rpart2$finalModel,type=2,sub='')
```


logLoss (Cross-Validation)
2.0
1.5
1.0
0.5
0.000
0.005
0.010
0.015
0.020
Complexity Parameter

```
    cp
4 0.003
```



6. Evaluate the model performance on the test set

```
# Predict the probabilities for the observations in the test dataset

predicted_te <- predict(caret_rpart2, recidivism_te, type='prob')

dim(predicted_te)
```

```
[1] 5460    2
```

```
head(predicted_te)
```

```
         No       Yes
```

```
1 0.8652790 0.1347210
2 0.6355353 0.3644647
3 0.7754211 0.2245789
4 0.6355353 0.3644647
5 0.7754211 0.2245789
6 0.6559297 0.3440703
```

```
# Compute the AUC

require(cutpointr)

cut.obj <- cutpointr(x     = predicted_te$Yes,
                     class = recidivism_te$Recidivism_Arrest_Year2)

auc(cut.obj)
```

```
[1] 0.6521839
```

```
# Confusion matrix assuming the threshold is 0.5

pred_class <- ifelse(predicted_te$Yes>.5,1,0)

confusion <- table(recidivism_te$Recidivism_Arrest_Year2,pred_class)

confusion
```

```
   pred_class
       0    1
  0 3989  157
  1 1197  117
```

```
# True Negative Rate

confusion[1,1]/(confusion[1,1]+confusion[1,2])
```

```
[1] 0.9621322
```

```
# False Positive Rate

confusion[1,2]/(confusion[1,1]+confusion[1,2])
```

```
[1] 0.03786782
```

```
# True Positive Rate

confusion[2,2]/(confusion[2,1]+confusion[2,2])
```

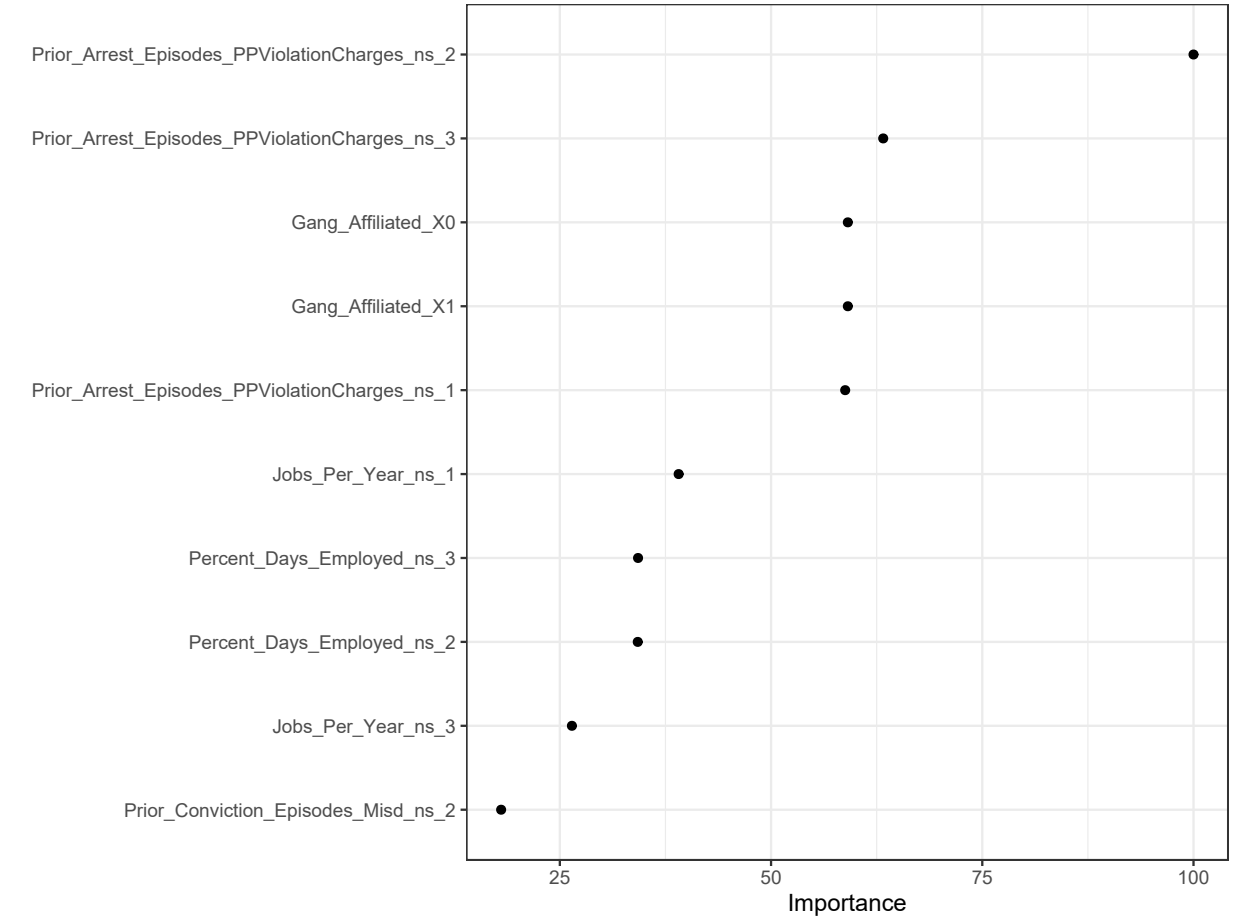
```
[1] 0.0890411
```

```
# Precision

confusion[2,2]/(confusion[1,2]+confusion[2,2])
```

```
[1] 0.4270073
```

7. Variable Importance Plot

```
require(vip)

vip(caret_rpart2,
    num_features = 10,
    geom = "point") +
  theme_bw()
```



8. Comparison of model performance

	-LL	AUC	ACC	TPR	TNR	FPR	PRE
Logistic Regression	0.5096	0.7192	0.755	0.142	0.949	0.051	0.471
Logistic Regression with Ridge Penalty	0.5111	0.7181	0.754	0.123	0.954	0.046	0.461
Logistic Regression with Lasso Penalty	0.5090	0.7200	0.754	0.127	0.952	0.048	0.458

	-LL	AUC	ACC	TPR	TNR	FPR	PRE
Logistic Regression with Elastic Net	0.5091	0.7200	0.753	0.127	0.952	0.048	0.456
Decision Tree	0.5522	0.6521	0.752	0.089	0.962	0.038	0.427