Statistical Learning, Homework #2

Elisa Paolazzi [MAT. 224424]

01/04/2022

R Setup

```r
library(tidyverse)
library(dplyr) # for data manipulation
library(ggplot2) # for nice plots
library(GGally) # for ggpairs()
library(tree) # for cost-complexity trees
library(rsample) # for train-test split
library(caret) # for K-fold CV
library(randomForest) # for Random forest
library(gbm) # for boosted trees
library(cowplot) # for plot grid
library(glmnet) # for Shrinkage methods

# choose a seed
seed = 14
```

Data exploration

The first step of the analysis is to load the data (`prostate.csv`) and explore it:

```r
# load the dataset
prostate_df = read.csv("prostate.csv")
# dataset dimensions
dim(prostate_df)
```

```
## [1] 97  9
```

```r
# show the dataset
knitr::kable(head(prostate_df))
```

lcavol	lweight	age	lbph	svi	lcp	gleason	pgg45	lpsa
-0.5798185	2.769459	50	-1.386294	0	-1.386294	6	0	-0.4307829
-0.9942523	3.319626	58	-1.386294	0	-1.386294	6	0	-0.1625189
-0.5108256	2.691243	74	-1.386294	0	-1.386294	7	20	-0.1625189
-1.2039728	3.282789	58	-1.386294	0	-1.386294	6	0	-0.1625189
0.7514161	3.432373	62	-1.386294	0	-1.386294	6	0	0.3715636
-1.0498221	3.228826	50	-1.386294	0	-1.386294	6	0	0.7654678

As described in the assignment the dataset contains 97 observations (men who were about to receive a radical prostatectomy) for 9 variables. The outcome variable is contained in the `lpsa` column, while the other 8 columns contain explanatory variables, which represent patients' clinical measures.

We can also print a summary in order to better understand the content of the columns, and thus the nature of our predictors:

```r
summary(prostate_df)
```

```
##      lcavol           lweight           age             lbph        
##  Min.   :-1.3471   Min.   :2.375   Min.   :41.00   Min.   :-1.3863  
##  1st Qu.: 0.5128   1st Qu.:3.376   1st Qu.:60.00   1st Qu.:-1.3863  
##  Median : 1.4469   Median :3.623   Median :65.00   Median : 0.3001  
##  Mean   : 1.3500   Mean   :3.629   Mean   :63.87   Mean   : 0.1004  
##  3rd Qu.: 2.1270   3rd Qu.:3.876   3rd Qu.:68.00   3rd Qu.: 1.5581  
##  Max.   : 3.8210   Max.   :4.780   Max.   :79.00   Max.   : 2.3263  
##       svi              lcp             gleason          pgg45       
##  Min.   :0.0000   Min.   :-1.3863   Min.   :6.000   Min.   :  0.00  
##  1st Qu.:0.0000   1st Qu.:-1.3863   1st Qu.:6.000   1st Qu.:  0.00  
##  Median :0.0000   Median :-0.7985   Median :7.000   Median : 15.00  
##  Mean   :0.2165   Mean   :-0.1794   Mean   :6.753   Mean   : 24.38  
##  3rd Qu.:0.0000   3rd Qu.: 1.1787   3rd Qu.:7.000   3rd Qu.: 40.00  
##  Max.   :1.0000   Max.   : 2.9042   Max.   :9.000   Max.   :100.00  
##       lpsa        
##  Min.   :-0.4308  
##  1st Qu.: 1.7317  
##  Median : 2.5915  
##  Mean   : 2.4784  
##  3rd Qu.: 3.0564  
##  Max.   : 5.5829  
```

The outcome variable `lpsa` is a numerical variable and represents the $log(cancervolumeincm3)$. `lcavol`, `lweight`, `age`, `lbph`, `lcp`, `pgg45` are numerical predictors, `svi` is a dummy variable and `gleason` is a qualitative variable. No missing values are detected.

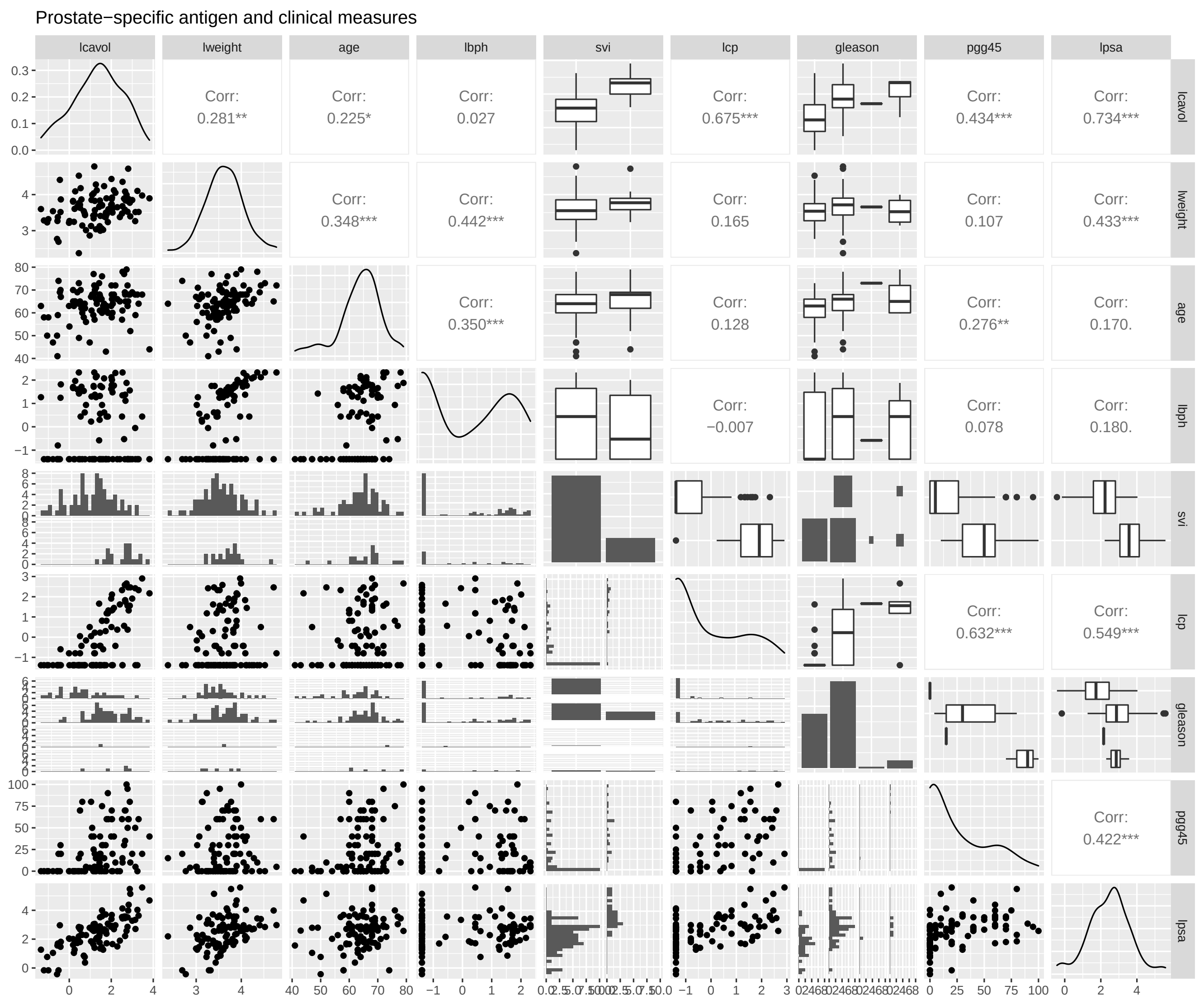
Since `svi` and `gleason` are categorical variables we transform them into factors.:

```r
# convert dummy and categorical columns as factors
prostate_df = prostate_df %>%
  mutate_at(c('svi', 'gleason'), as.factor)
# show a sample of the dataset to verify the transformation
as.tibble(prostate_df)
```

```
## # A tibble: 97 x 9
##    lcavol lweight   age   lbph svi     lcp gleason pgg45   lpsa
##     <dbl>   <dbl> <int>  <dbl> <fct> <dbl> <fct>   <int>  <dbl>
##  1 -0.580    2.77    50 -1.39  0     -1.39 6           0 -0.431
##  2 -0.994    3.32    58 -1.39  0     -1.39 6           0 -0.163
##  3 -0.511    2.69    74 -1.39  0     -1.39 7          20 -0.163
##  4 -1.20     3.28    58 -1.39  0     -1.39 6           0 -0.163
##  5  0.751    3.43    62 -1.39  0     -1.39 6           0  0.372
##  6 -1.05     3.23    50 -1.39  0     -1.39 6           0  0.765
##  7  0.737    3.47    64  0.615 0     -1.39 6           0  0.765
##  8  0.693    3.54    58  1.54  0     -1.39 6           0  0.854
##  9 -0.777    3.54    47 -1.39  0     -1.39 6           0  1.05 
## 10  0.223    3.24    63 -1.39  0     -1.39 6           0  1.05 
## # ... with 87 more rows
```

Considering the predictors it is useful to show their distribution and their relationships. We can do this by means of the `ggpairs()` function, which displays us a revealing plot:

```r
ggpairs(prostate_df, title="Prostate-specific antigen and clinical measures")
```



Observing the plot we can see each variable density plot on the diagonal, the correlation between the continuous variables (in the upper panels), the scatter plots of the continuous variables (in the lower panels), and the histograms and box plots for the combinations between the categorical and the continuous variables. From this visualization we can notice some quite correlated predictors, such as `lcavol` and `lcp` (0.68), and correlations with the outcome variable and predictors, such as `lcavol` (0.73). Other useful insights about the relationships between the variables can be spotted in the visualization.

We can also directly show the correlation matrix between the numerical variables:

```
knitr::kable(cor(dplyr::select(prostate_df, -svi, -gleason)))
```

	lcavol	lweight	age	lbph	lcp	pgg45	lpsa
lcavol	1.0000000	0.2805214	0.2249999	0.0273497	0.6753105	0.4336522	0.7344603
lweight	0.2805214	1.0000000	0.3479691	0.4422644	0.1645371	0.1073538	0.4333194
age	0.2249999	0.3479691	1.0000000	0.3501859	0.1276678	0.2761124	0.1695928
lbph	0.0273497	0.4422644	0.3501859	1.0000000	-0.0069994	0.0784600	0.1798094
lcp	0.6753105	0.1645371	0.1276678	-0.0069994	1.0000000	0.6315282	0.5488132
pgg45	0.4336522	0.1073538	0.2761124	0.0784600	0.6315282	1.0000000	0.4223159
lpsa	0.7344603	0.4333194	0.1695928	0.1798094	0.5488132	0.4223159	1.0000000

As we said, there are some correlations between the variable.

1. Fit a decision tree on the whole data and plot the results. Choose the tree complexity by cross-validation and decide whether you should prune the tree based on the results. Prune the tree if applicable and interpret the fitted model.

Tree-based methods involve stratifying or segmenting the predictor space into a number of simple regions $R1, ..., Rn$. In regression settings, in order to make a prediction for a given observation, we use the mean response value for the training observations in the region to which it belongs. As we will see the main advantage of this methods over other types of regression models are in terms of interpretation and graphical representation.

We first fit a regression tree on the whole data:

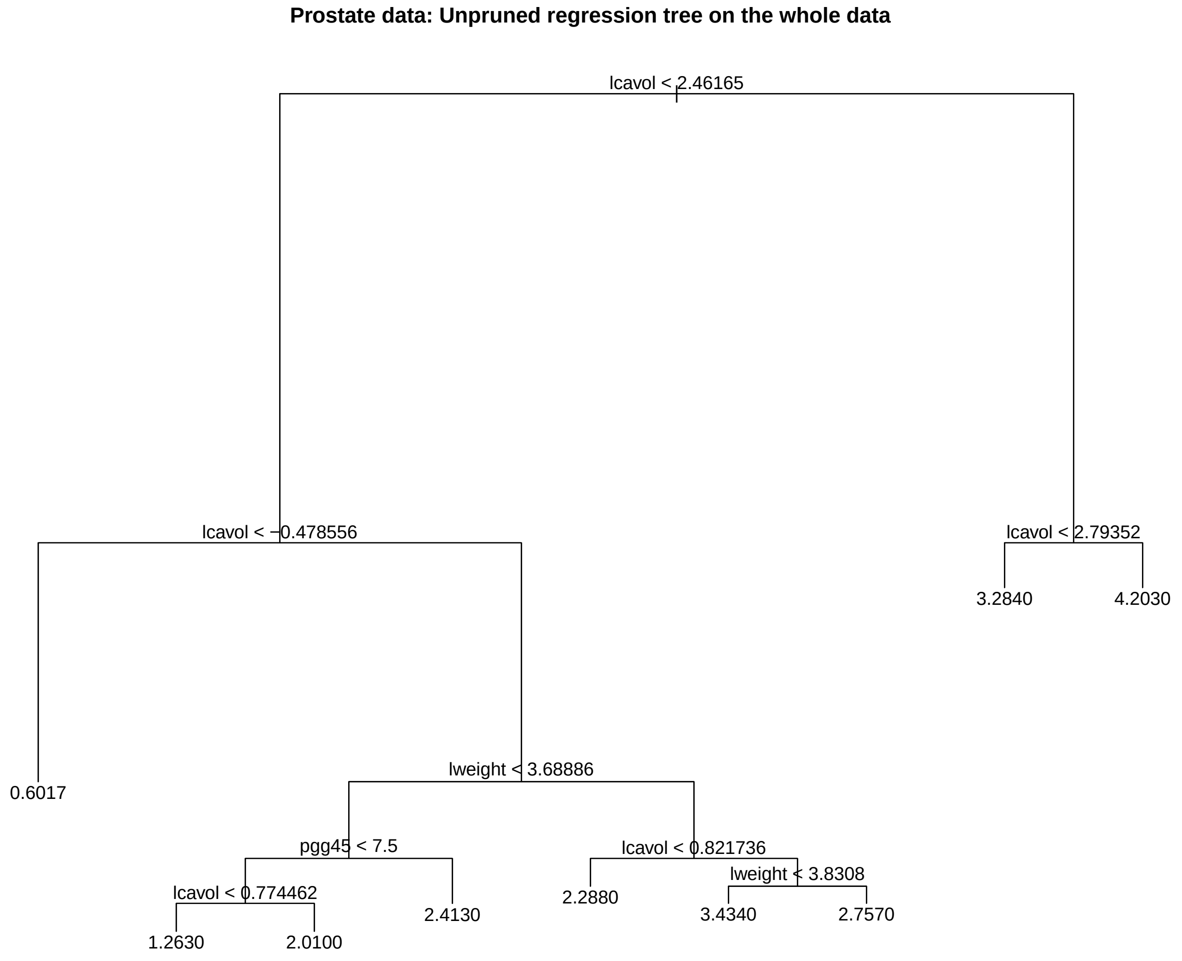
```
# fit a regression tree on the whole dataset
tree_pros = tree(lpsa ~ ., prostate_df)
# tree's summary
summary(tree_pros)
```

```
##
## Regression tree:
## tree(formula = lpsa ~ ., data = prostate_df)
## Variables actually used in tree construction:
## [1] "lcavol"  "lweight" "pgg45"
## Number of terminal nodes:  9
## Residual mean deviance:  0.4119 = 36.24 / 88
## Distribution of residuals:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -1.499000 -0.488000  0.003621  0.000000  0.481200  1.380000
```

The output first shows us the variables that were actually used in the construction of the tree, in our case these are `lcavol`, `lweight` and `pgg45` (3/8). Secondly, it shows the number of terminal nodes (9 in our case) and the residual mean deviance which is calculated by dividing the tree deviance (36.24) by $n-|T0|$ (number of observations - number of terminal nodes, which is $97-9=88$ in our case). Note that the residual mean deviance is a measure of the error remaining in the tree after construction and, for a regression tree, this is related to the mean squared error. Lastly, we can see the distribution of residuals: their values are of course satisfactory since they are close to be centered on 0 and are roughly symmetrical.

We can plot the tree:

```
# plot the branches
plot(tree_pros)
# add branches' labels
text(tree_pros, pretty=0)
# add title
title(main="Prostate data: Unpruned regression tree on the whole data")
```



Observing the graph we can better understand how the tree was built using only three predictors (the predictors have one or more splits based on different thresholds). The most important predictor, which stand at the stump of the tree is `lcavol`, which then creates other subsequent branches. The tree concludes with 9 terminal nodes identifying the partitions $(R1, ..., R9)$ of the feature space. Each terminal node shows the average of the `lpsa` values for the corresponding region, which will then represent the prediction for the observations that will end up within them.

We can also examine in more details the tree's structure by displaying it in text mode:

```r
# display tree details
tree_pros
```

```
## node), split, n, deviance, yval
##       * denotes terminal node
##
##  1) root 97 127.9000 2.4780
##    2) lcavol < 2.46165 76  67.2700 2.1230
##      4) lcavol < -0.478556 9   5.5980 0.6017 *
##      5) lcavol > -0.478556 67  38.0500 2.3270
##       10) lweight < 3.68886 38  21.5100 2.0330
##         20) pgg45 < 7.5 21   9.1450 1.7250
##           40) lcavol < 0.774462 8   2.6660 1.2630 *
##           41) lcavol > 0.774462 13   3.7200 2.0100 *
##         21) pgg45 > 7.5 17   7.9170 2.4130 *
##       11) lweight > 3.68886 29   8.9550 2.7120
##         22) lcavol < 0.821736 10   2.4340 2.2880 *
##         23) lcavol > 0.821736 19   3.7750 2.9360
##           46) lweight < 3.8308 5   0.8768 3.4340 *
##           47) lweight > 3.8308 14   1.2100 2.7570 *
##    3) lcavol > 2.46165 21  16.2500 3.7650
##      6) lcavol < 2.79352 10   2.8850 3.2840 *
##      7) lcavol > 2.79352 11   8.9370 4.2030 *
```

For each node important information are displayed: the splitting criterion, the number of observations in the branch, the deviance and the overall branch prediction; lastly, the asterisk indicates whether the node is a terminal one or not.

Once the tree is fitted and displayed, we can use cross-validation to choose the complexity of the tree and decide whether to prune it or not. However, remember that we are dealing with a tree fitted on the whole dataset.

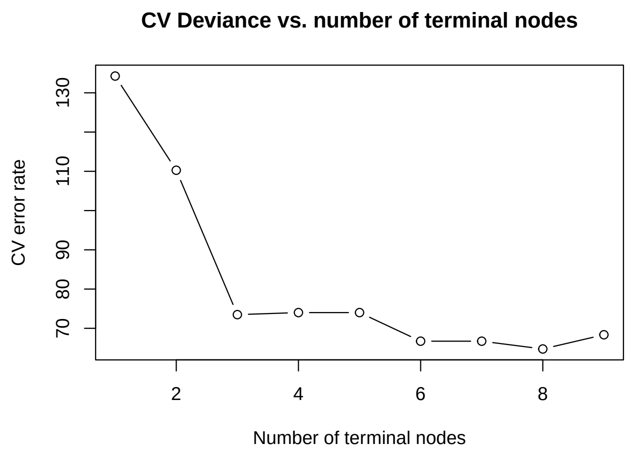
```
# set seed for reproducible results
set.seed(32)

# cross-validation
cv.pros = cv.tree(tree_pros)
cv.pros
```

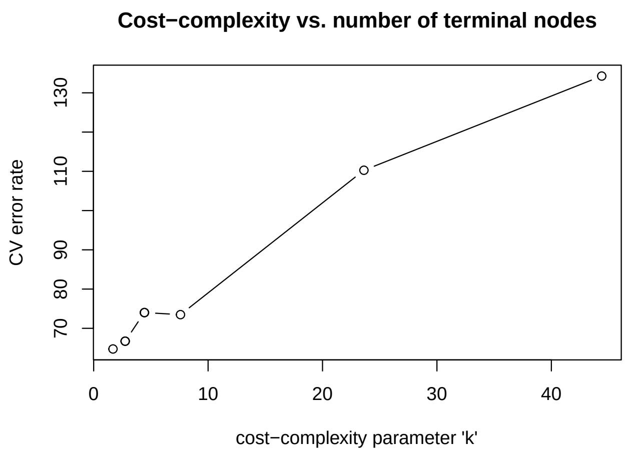
```
## $size
## [1] 9 8 7 6 5 4 3 2 1
##
## $dev
## [1]  68.36065  64.73874  66.72033  66.72033  74.00912  74.00912  73.48853
## [8] 110.27271 134.26677
##
## $k
## [1]      -Inf  1.687786  2.746385  2.758375  4.427103  4.445951  7.587544
## [8] 23.619667 44.401279
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

The cross-validation output shows the number of possible terminal nodes of each tree (from 9 to 1), the corresponding cross-validation error rate and the value of the cost-complexity parameter (which is denoted by k here). As we can already note, at the second position (corresponding to node 8), we record the lowest value of cross-validation error rate (and also the lowest value of k, excluding the one belonging to the maximum size value), which suggests that the tree may be pruned.

Let's visualize the cross-validation error rate as a function of either size and k:

```
op = par(mfrow=c(1, 2))
# plot cross-validation deviance versus the number of terminal nodes
plot(cv.pros$size, cv.pros$dev, type="b", main="CV Deviance vs. number of terminal nodes",
     xlab="Number of terminal nodes",
     ylab="CV error rate")
# plot cost-complexity parameter versus the number of terminal nodes
plot(cv.pros$k, cv.pros$dev, type="b", main="Cost-complexity vs. number of terminal nodes",
     xlab="cost-complexity parameter 'k'",
     ylab="CV error rate")
```





Here we can visually display how the error rate behaves depending on the size and the cost-complexity parameter. As we previously noted, we can clearly see from the plot that the error rate has its minimum with the size of 8.

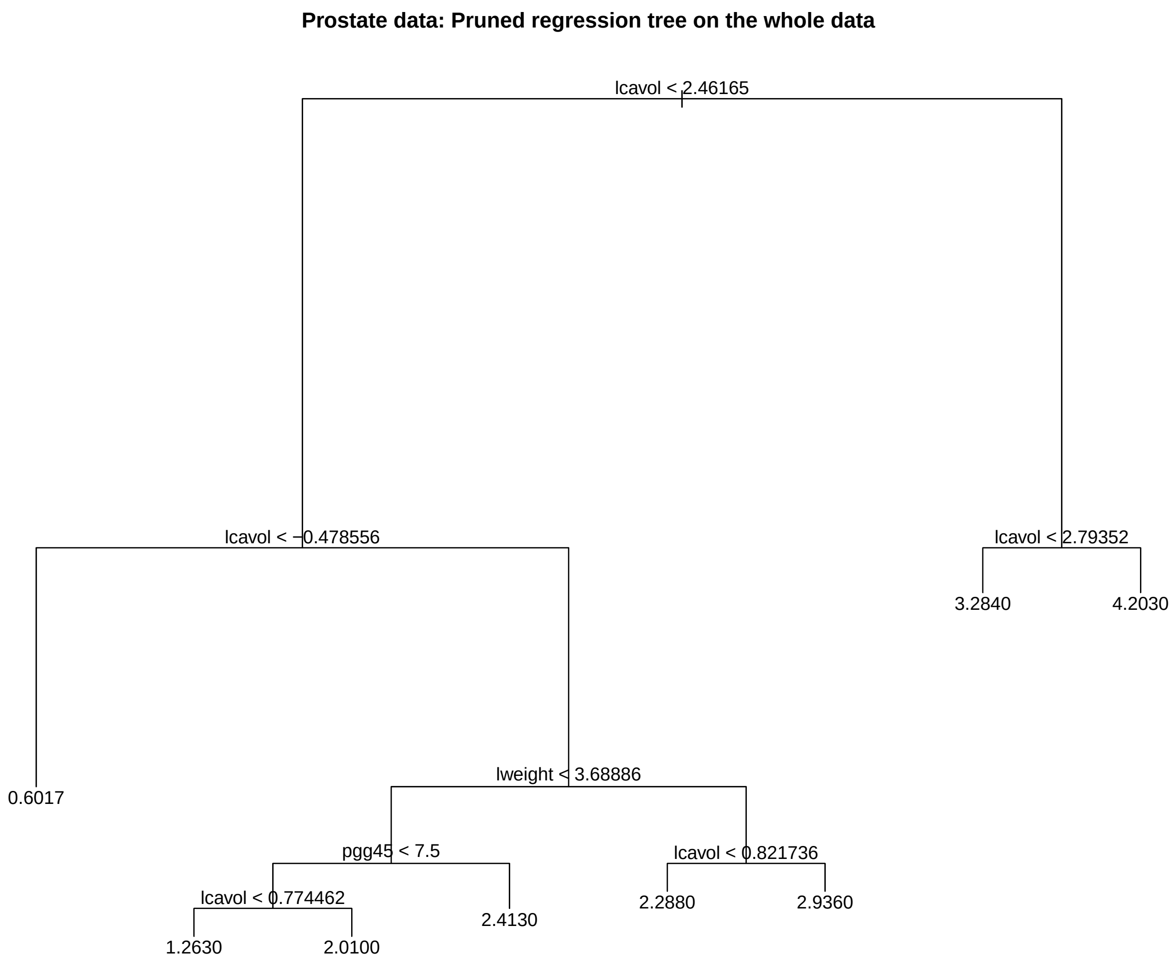
We then can prune the tree using the optimal tree size.

```
# get the optimal size
opt.size = cv.pros$size[which.min(cv.pros$dev)]
# prune the tree using the optimal size value
prune_pros = prune.tree(tree_pros, best=opt.size)
# pruned tree's summary
summary(prune_pros)
```

```
##
## Regression tree:
## snip.tree(tree = tree_pros, nodes = 23L)
## Variables actually used in tree construction:
## [1] "lcavol"  "lweight" "pgg45"
## Number of terminal nodes:  8
## Residual mean deviance:  0.4262 = 37.93 / 89
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1.49900 -0.48800 -0.04795  0.00000  0.50610  1.38000
```

Reading the summary we can notice that the value of the residual mean deviance has slightly increased compared to the previous unpruned tree. This is inevitable since the tree deviance has also increased while $n - |T0|$ has decreased (since n is lower). We can also display the pruned tree:

```
# reset mfrow()
op = par(mfrow=c(1, 1))
# plot the branches
plot(prune_pros)
# add branches' labels
text(prune_pros, pretty=0)
# add title
title(main="Prostate data: Pruned regression tree on the whole data")
```



Compared to the previous tree, we can see that the last split of `lweight` (`lweight` < 3.8308) is missing.

It is now useful to calculate the mean squared error on the whole data, in order to see how the unpruned and pruned trees perform on it:

```
# make prediction on the whole data using the whole data fitted unpruned tree
yhat_unpr = predict(tree_pros, newdata=prostate_df)
# calculate MSE
mean((yhat_unpr - prostate_df$lpsa)^2)
```

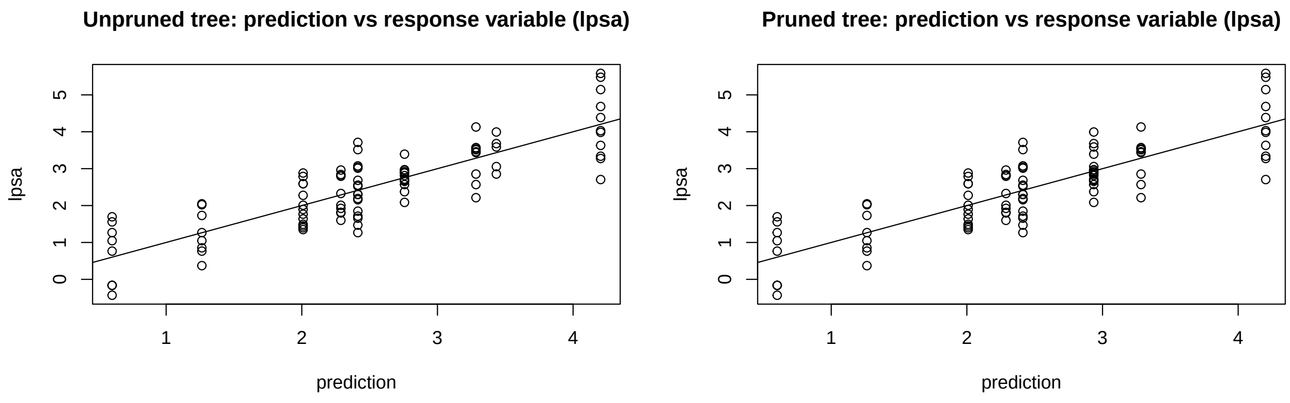
```
## [1] 0.373645
```

```
# make prediction on the whole data using the whole data fitted pruned tree
yhat_pr = predict(prune_pros, newdata=prostate_df)
# calculate MSE
mean((yhat_pr - prostate_df$lpsa)^2)
```

```
## [1] 0.3910449
```

The two models perform roughly the same since the MSE values (which is basically the training error in this case) are very similar. However the unpruned version is the best in this case and its MSE value is one of the lowest that can be achieved with these methods.

```r
op = par(mfrow=c(1, 2))
plot(yhat_unpr, prostate_df$lpsa, main = "Unpruned tree: prediction vs response variable (lpsa)",
     xlab="prediction",
     ylab="lpsa")
abline(0, 1)

plot(yhat_pr, prostate_df$lpsa, main = "Pruned tree: prediction vs response variable (lpsa)",
     xlab="prediction",
     ylab="lpsa")
abline(0, 1)
```



We have achieved great results since the trees are able to capture the complexity of the data. However, this is also due to the fact that we tested the models on the same data they were trained on (the whole dataset), for this reason the models tend to be perfectly adapted to them.

Nevertheless a regression tree performance is not properly evaluated until we estimate the test error. For this purpose we split the observations into a training and a testing partition:

```r
# set the seed for reproducible partitions
set.seed(42)

# split data into training and test sets
split = initial_split(prostate_df, prop=0.7)
x_train = training(split)
x_test = testing(split)
y_test = x_test$lpsa
```

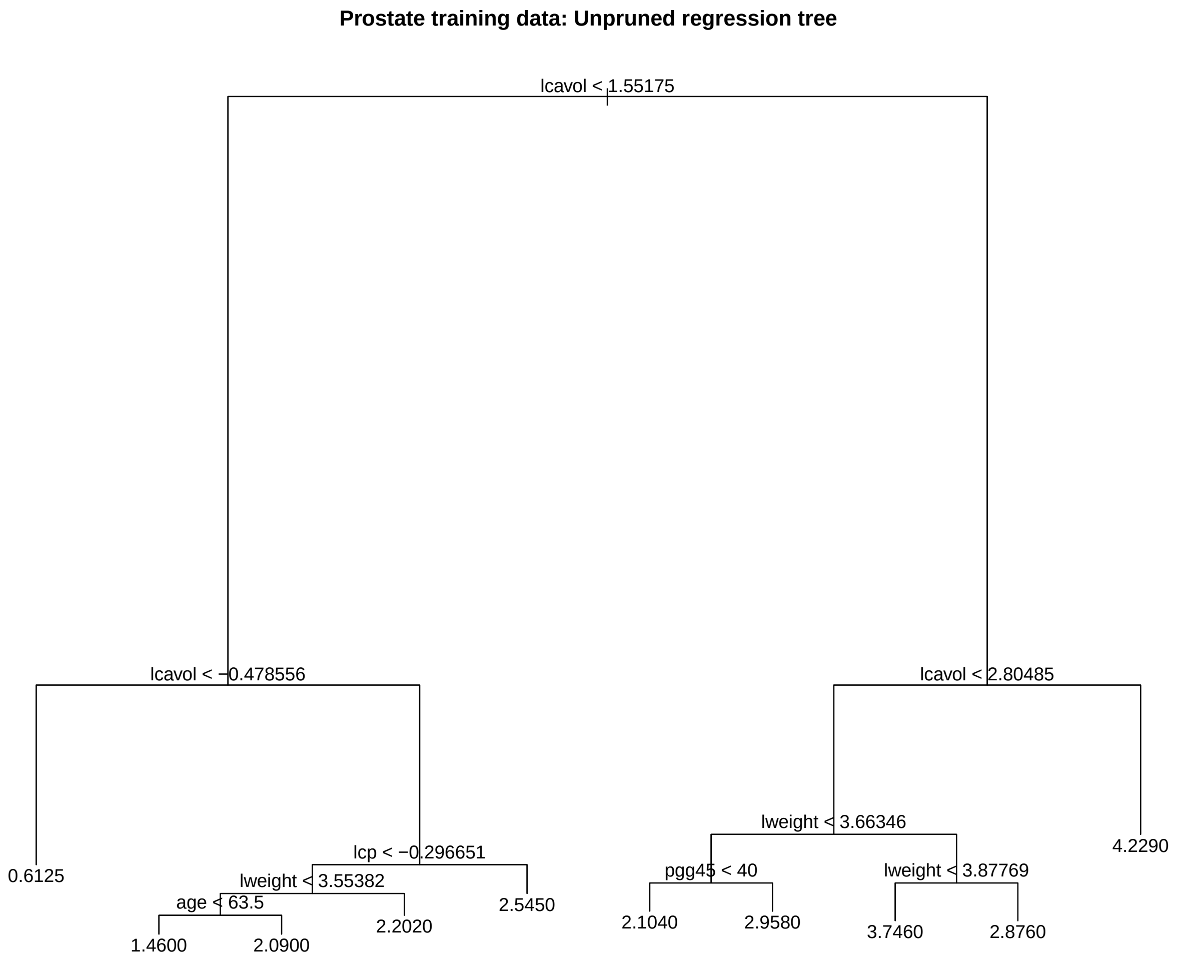
We can now fit the unpruned tree on the training set:

```r
# fit a regression tree on the training set
unpruned_tree = tree(lpsa ~ ., x_train)
# tree's summary
summary(unpruned_tree)
```

```
##
## Regression tree:
## tree(formula = lpsa ~ ., data = x_train)
## Variables actually used in tree construction:
## [1] "lcavol"  "lcp"     "lweight" "age"     "pgg45"
## Number of terminal nodes:  10
## Residual mean deviance:  0.4234 = 24.14 / 57
## Distribution of residuals:
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -1.524000 -0.407000  0.005824  0.000000  0.471500  1.354000
```

This time 5 predictors are actually used in the tree construction and the terminal nodes are 10. Let's visualize the tree:

```r
# reset mfrow()
op = par(mfrow=c(1, 1))

plot(unpruned_tree)
text(unpruned_tree, pretty=0)
title(main="Prostate training data: Unpruned regression tree")
```



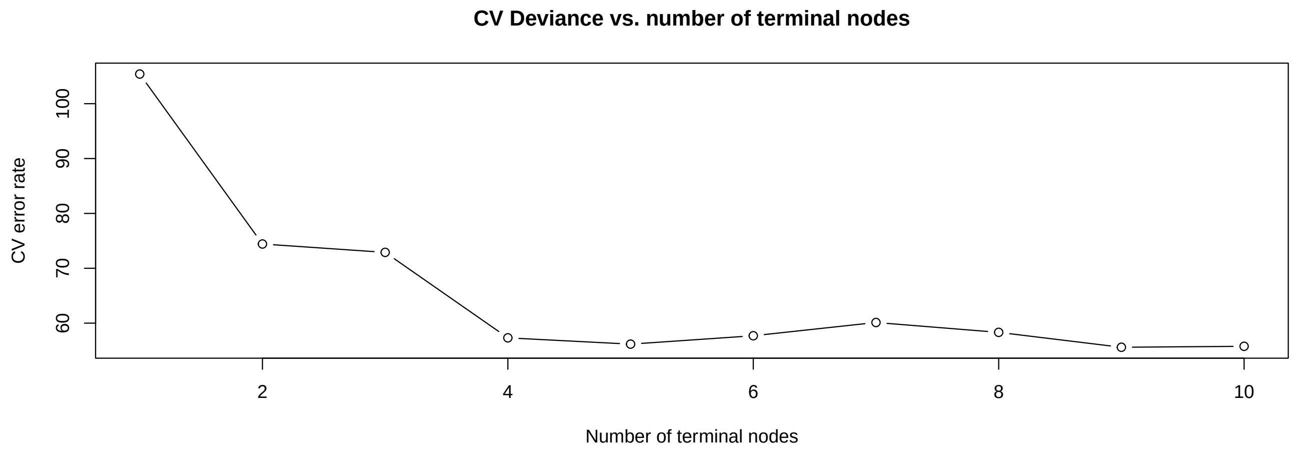
Let's perform a cross-validation in order to decide whether to prune the tree or not:

```r
# set seed for reproducible results
set.seed(8)
# cross-validation
cv.pros_tr = cv.tree(unpruned_tree)
cv.pros_tr
```

```
## $size
##  [1] 10  9  8  7  6  5  4  3  2  1
##
## $dev
##  [1]  55.78079  55.60460  58.32142  60.11856  57.70056  56.17331  57.31861
##  [8]  72.89717  74.42613 105.40015
##
## $k
##  [1]      -Inf  1.222737  1.410028  1.824375  1.864553  2.443329  3.150330
##  [8]  9.666743 11.632975 38.115990
```

```
##
## $method
## [1] "deviance"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

```
plot(cv.pros_tr$size, cv.pros_tr$dev, type="b",
     main="CV Deviance vs. number of terminal nodes",
     xlab="Number of terminal nodes",
     ylab="CV error rate")
```



The optimal size is 9 (according also with the cost-complexity parameter), thus we prune the tree using it:

```
# get the optimal size
opt.size = cv.pros_tr$size[which.min(cv.pros_tr$dev)]
# prune the tree
pruned_tree = prune.tree(unpruned_tree, best=opt.size)
summary(pruned_tree)
```

```
##
## Regression tree:
## snip.tree(tree = unpruned_tree, nodes = 20L)
## Variables actually used in tree construction:
## [1] "lcavol"  "lcp"     "lweight" "pgg45"
## Number of terminal nodes:  9
## Residual mean deviance:  0.4372 = 25.36 / 58
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1.52400 -0.39470  0.05497  0.00000  0.44500  1.35400
```

Compared to the previous full tree, here the `age` branches (which was one of the terminal nodes) was pruned and thus this predictor is not included anymore in this tree construction.

We can finally test the trees on the test set and compare the mean squared error:

```
# make prediction on the test set using the training set fitted unpruned tree
yhat_unpr = predict(unpruned_tree, newdata=x_test)
# calculate MSE
mean((yhat_unpr - y_test)^2)
```

```
## [1] 0.5264935
```

```r
# calculate sd
sqrt(mean((yhat_unpr - y_test)^2))
```

```
## [1] 0.7255987
```

```r
# make prediction on the test set using the train set fitted pruned tree
yhat_pr = predict(pruned_tree, newdata=x_test)
# calculate MSE
mean((yhat_pr - y_test)^2)
```

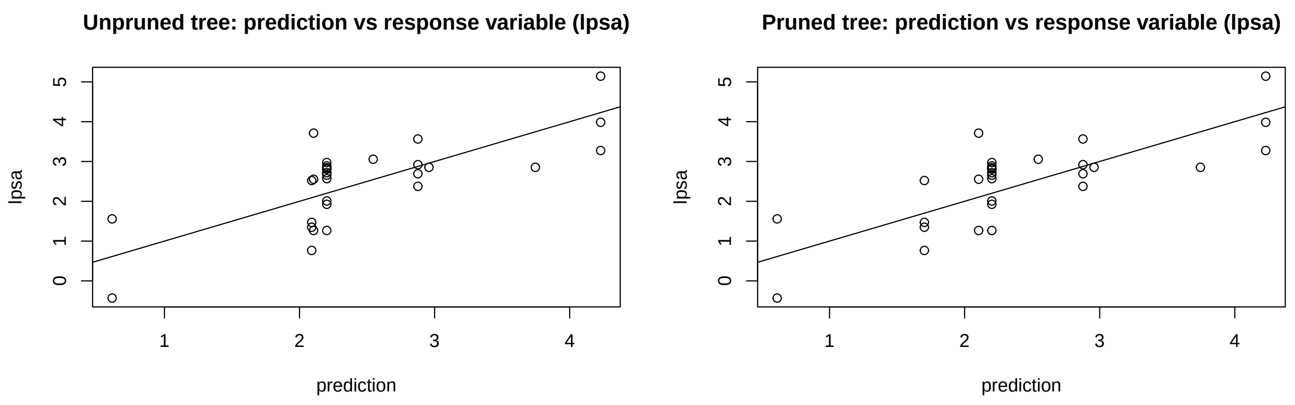
```
## [1] 0.4882392
```

```r
# calculate sd
sqrt(mean((yhat_pr - y_test)^2))
```

```
## [1] 0.6987412
```

And plot the results:

```r
op = par(mfrow=c(1, 2))
plot(yhat_unpr, x_test$lpsa,
     main = "Unpruned tree: prediction vs response variable (lpsa)",
     xlab="prediction",
     ylab="lpsa")
abline(0, 1)

plot(yhat_pr, x_test$lpsa,
     main = "Pruned tree: prediction vs response variable (lpsa)",
     xlab="prediction",
     ylab="lpsa")
abline(0, 1)
```



In this case the pruned model performs better than the unpruned one. Moreover, the pruning procedure succeeds in slightly reducing the variance (as the standard deviation is also lower in the pruned tree) and thus also the overfitting.

However, if we try to repeat the test-train split the results tend to vary quite a bit: this is due to the high variance that is typical of tree models. Indeed, pruning the tree is often not enough to reduce variance.

2. Consider now a random forest and let m be the number of variables to consider at each split. Set the range for m from 1 to the number of explanatory variables, say $nvar$, and define a k-fold cross-validation schema for the selection of this tuning parameter, with k of your choice. Prepare a matrix with $nvar$ rows and 2 columns and fill the first column with the average cross-validation error corresponding to each choice of m and the second column with the OOB error (from the full dataset). Are the CV and OOB error different? Do they reach the minimum at the same value of m? Interpret the optimal model (either using the CV or the OOB error).

Random Forests are a useful method that is able to reduce variance at the cost of losing interpretability (since we have a *forest of trees*). For this purpose a Random Forest builds a number of decision trees on bootstrapped training samples. However, each time a split in a tree is considered and a random sample of m predictors is chosen as split candidates from the full set of p predictors. When we make a prediction, the result will be obtained by averaging the results of all trees.

First, we define the `nvar` vector, which will contain the number of variables that Random Forest models will consider at each split (from 1 to 8, the number of predictors) and the 10 folds for the k-fold cross validation (using the full dataset):

```r
# number of predictors
n_pred = ncol(prostate_df) - 1

# create nvar vector
nvar = seq(1,n_pred)

# set seed for reproducible folds
set.seed(4)
# get 10 set of index for training sets composition
flds = createFolds(prostate_df$lpsa, k = 10, list = T, returnTrain = T)
```

We now initialize the results matrix containing the cross-validation error and the OOB error columns:

```r
# initialize empty matrix
results_matrix = matrix(nrow=n_pred, ncol=2)
```

We can now calculate the average cross-validation and the OOB errors corresponding to each choice of m:

```r
# iterate over m values
for (m in nvar) {

  # CV error
  # create an empty list to store each fold MSE
  list_mse = c()

  # iterate over the folds
  for (i in flds) {
    # set seed for reproducible randomForest() results
    set.seed(seed)

    # define index based on the i fold
    train_idx = i
    # split train and test sets according to the indexes
    train = prostate_df[train_idx,]
    test = prostate_df[-train_idx,]
    y_ts = test$lpsa

    # fit a random forest on k - 1 folds and test on the remaining fold using
    # m number of variables
    rf = randomForest(lpsa ~ ., data=train, mtry=m, xtest = dplyr::select(test, -lpsa),
                      ytest = y_ts, importance=TRUE)
    # extract prediction
    temp_yhat = rf$test$predicted
    # compute MSE
    temp_mse = mean((temp_yhat - y_ts)^2)
    # add MSE to the mse_list
    list_mse = append(list_mse, temp_mse)
  }
  # average k-folds cross validation errors
  cv_error = mean(list_mse)
  # place the value in the matrix
  results_matrix[m,1] = cv_error

  # OOB error
  # set seed for reproducible randomForest() results
  set.seed(seed)

  # fit a random forest on the whole dataset using m number of variables
  rf = randomForest(lpsa ~ ., data=prostate_df, mtry=m, importance=TRUE)
  # extract OOB value
  oob = rf$mse[length(rf$mse)]
  # place the value in the matrix
  results_matrix[m,2] = oob
}

results_matrix
```

```
##           [,1]      [,2]
## [1,] 0.6401257 0.6660063
## [2,] 0.5911416 0.6242139
## [3,] 0.5916627 0.5982064
## [4,] 0.5918543 0.6132826
## [5,] 0.5938599 0.6166758
## [6,] 0.6025337 0.6016992
## [7,] 0.6134017 0.6168778
## [8,] 0.6341732 0.6068489
```

We can also convert the matrix to a dataframe in order to sort the results:

```
# convert matrix to df
results_df = as.data.frame(results_matrix, row.names=nvar)
colnames(results_df) = c("CV_error", "OOB_error")

# order results based on CV error
knitr::kable(results_df[order(results_df$CV_error),])
```

	CV_error	OOB_error
2	0.5911416	0.6242139
3	0.5916627	0.5982064
4	0.5918543	0.6132826
5	0.5938599	0.6166758
6	0.6025337	0.6016992
7	0.6134017	0.6168778
8	0.6341732	0.6068489
1	0.6401257	0.6660063

```
# order results based on OOB error
knitr::kable(results_df[order(results_df$OOB_error),])
```

	CV_error	OOB_error
3	0.5916627	0.5982064
6	0.6025337	0.6016992
8	0.6341732	0.6068489
4	0.5918543	0.6132826
5	0.5938599	0.6166758
7	0.6134017	0.6168778
2	0.5911416	0.6242139
1	0.6401257	0.6660063

As we can notice cross-validation and OOB errors do not reach the minimum at the same value of m: for cross-validation error the minimum is reached by 2, while for the OOB error the minimum is 3.

It should be noted that they can be very similar since OOB error could be seen as an estimate of cross-validation error (the OOB error converges to the leave-one-out cross validation error estimate). Moreover, OOB has clear computational advantages over CV error since no refitting is needed: we can obtain an estimate on unseen data just form our fitted model.

For these reasons both cross-validation and OOB errors are considered good approaches to select an optimal model.

However, we can try to see how the two models (for CV $m = 2$ and for OBB $m = 3$) perform on the previous train-test split:

```
# add "m" column to the df
results_df = results_df %>%
  mutate(m=nvar)
# extract optimal CV m
opt.m_cv = results_df$m[which.min(results_df$CV_error)]
# extract optimal OOB m
opt.m_oob = results_df$m[which.min(results_df$OOB_error)]

# set seed for reproducible results
set.seed(14)
# fit the best CV Random Forest
best_CV_rf = randomForest(lpsa ~ ., data=x_train, mtry=opt.m_cv,
                          xtest = dplyr::select(x_test, -lpsa),
                          ytest = y_test, importance=TRUE)
best_CV_rf
```

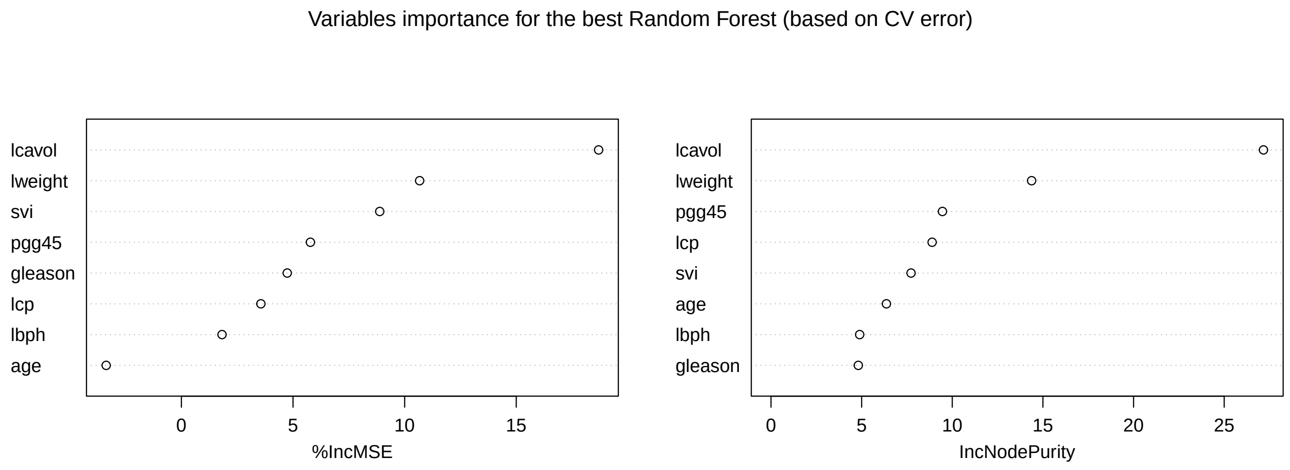
```
##
## Call:
##  randomForest(formula = lpsa ~ ., data = x_train, mtry = opt.m_cv,      xtest = dplyr::select(x_test, -l
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 2
##
##           Mean of squared residuals: 0.7914181
##                     % Var explained: 44.46
##                        Test set MSE: 0.41
##                     % Var explained: 62.91
```

```
# fit the best OBB Random Forest
best_OOB_rf = randomForest(lpsa ~ ., data=x_train, mtry=opt.m_oob,
                           xtest = dplyr::select(x_test, -lpsa),
                           ytest = y_test, importance=TRUE)
best_OOB_rf
```

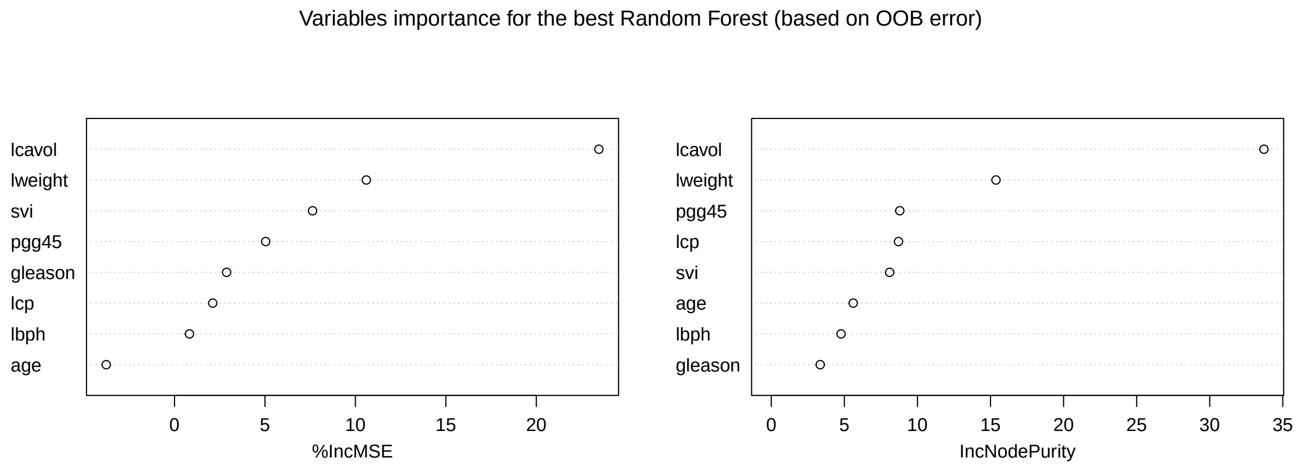
```
##
## Call:
##  randomForest(formula = lpsa ~ ., data = x_train, mtry = opt.m_oob,      xtest = dplyr::select(x_test, 
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 3
##
##           Mean of squared residuals: 0.751261
##                     % Var explained: 47.28
##                       Test set MSE: 0.4
##                     % Var explained: 63.44
```

As we can see, the test set MSE values are essentially the same, which suggests that the performances of the models are very similar in this case. Besides that we can check the importance of each predictors in the two models:

```
op = par(mfrow=c(2, 1))
varImpPlot(best_CV_rf,
           main="Variables importance for the best Random Forest (based on CV error)")
```



```
varImpPlot(best_OOB_rf,
           main="Variables importance for the best Random Forest (based on OOB error)")
```



The left-side plot shows us the mean decrease in accuracy of the predictions when that variable is removed from the model (`%IncMSE`), while the right-side plot x-axes represent a measure of the total decrease in node impurity resulting from splits over that variable. As we can notice the two models have the same most important variables, which are represented by `lcavol` and `lweight`.

However, the chosen numbers of m (2 and 3) were selected using the whole dataset (as required by the assignment), and having fitted and tested the model on our partitions using such m-numbers does not make much sense, as the models may be overfitted.

For this reason, we can repeat the parameter tuning performing CV and OOB on the training set only:

```r
# initialize empty matrix
results_matrix_tr = matrix(nrow=n_pred, ncol=2)

# set seed for reproducible folds
set.seed(4)
# recreate 10 set of index for training set composition (using only training set)
flds_tr = createFolds(x_train$lpsa, k = 10, list = T, returnTrain = T)

# iterate over m values
for (m in nvar) {
  # create an empty list to store each folds MSE
  list_mse = c()

  # iterate over the folds
  for (i in flds_tr) {
    # set seed for reproducible randomForest() results
    set.seed(seed)

    # define index based on the i fold
    train_idx = i
    # split train and test sets according to the indexes
    train = x_train[train_idx,]
    test = x_train[-train_idx,]
    y_ts = test$lpsa

    # fit a random forest on k - 1 folds and test on the remaining fold using
    # m number of variables
    rf = randomForest(lpsa ~ ., data=train, mtry=m, xtest = dplyr::select(test, -lpsa),
                      ytest = y_ts, importance=TRUE)
    # extract prediction
    temp_yhat = rf$test$predicted
    # compute MSE
    temp_mse = mean((temp_yhat - y_ts)^2)
    # add MSE to the mse_list
    list_mse = append(list_mse, temp_mse)
  }
  # average k-folds cross validation errors
  cv_error = mean(list_mse)
  # place the value in the matrix
  results_matrix_tr[m,1] = cv_error

  # set seed for reproducible randomForest() results
  set.seed(seed)

  # fit a random forest on the whole dataset using m number of variables
  rf = randomForest(lpsa ~ ., data=x_train, mtry=m, importance=TRUE)
  # extract OOB value
  oob = rf$mse[length(rf$mse)]
  # place the value in the matrix
  results_matrix_tr[m,2] = oob
}

# convert matrix to df
results_df_tr = as.data.frame(results_matrix_tr, row.names=nvar)
colnames(results_df_tr) = c("CV_error", "OOB_error")

# order results based on CV error
knitr::kable(results_df_tr[order(results_df_tr$CV_error),])
```

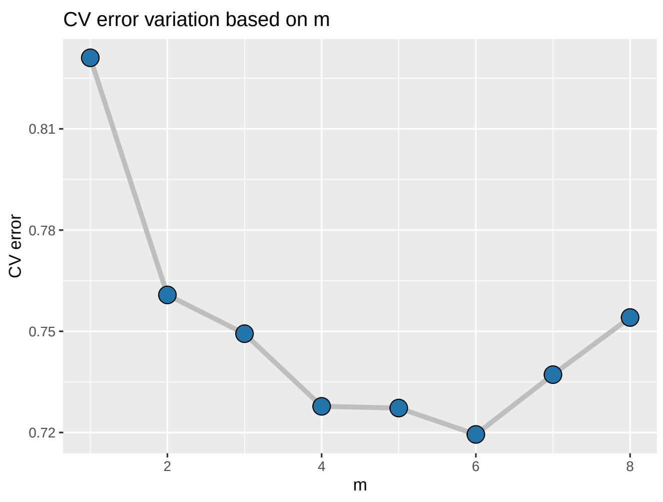
	CV_error	OOB_error
6	0.7194310	0.7359901
5	0.7272455	0.7349767
4	0.7277554	0.7337962
7	0.7371515	0.7436633
3	0.7492856	0.7692272
8	0.7540882	0.7519761
2	0.7607810	0.7914181
1	0.8310701	0.8389603

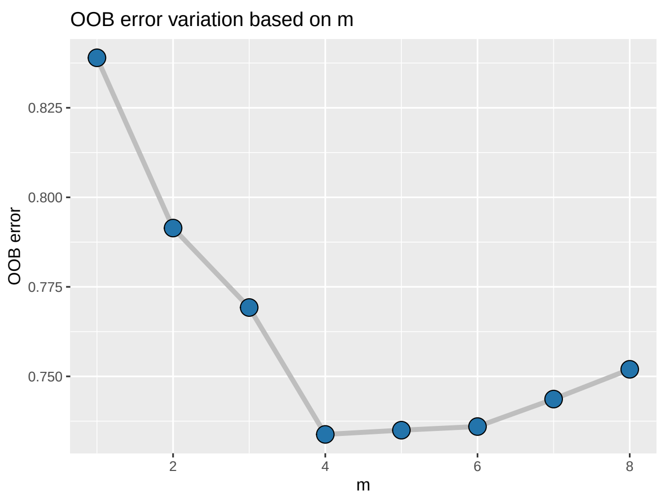
```r
# order results based on OOB error
knitr::kable(results_df_tr[order(results_df_tr$OOB_error),])
```

	CV_error	OOB_error
4	0.7277554	0.7337962
5	0.7272455	0.7349767
6	0.7194310	0.7359901
7	0.7371515	0.7436633
8	0.7540882	0.7519761
3	0.7492856	0.7692272
2	0.7607810	0.7914181
1	0.8310701	0.8389603

We can also produce a plot in order to better observe the trend of errors according to m:

```r
# add m column to df
results_df_tr = results_df_tr %>%
  mutate(m=nvar)

# CV error plot
mse_plot = results_df_tr %>%
 ggplot(aes(x=m, y=CV_error)) +
 geom_line(color="grey", size=1.5) +
 geom_point(shape=21, color="black", fill="#2374AB", size=5) +
 labs(title = "CV error variation based on m",
 x = "m",
 y = "CV error")

# OOB error plot
oob_plot = results_df_tr %>%
 ggplot(aes(x=m, y=OOB_error)) +
 geom_line(color="grey", size=1.5) +
 geom_point(shape=21, color="black", fill="#2374AB", size=5) +
 labs(title = "OOB error variation based on m",
 x = "m",
 y = "OOB error")

# plot side by side
plot_grid(mse_plot, oob_plot,
 labels = "", ncol = 2, nrow = 1)
```





As expected, the CV and OOB error values are higher now. In particular, the best m values is 6 for CV error and 4 for OBB error.

As a final step we can use these m values (this time selected using the training set) to fit the models and test them on the test set:

```
# extract optimal number of m based on CV
opt.m_cv_tr = results_df_tr$m[which.min(results_df_tr$CV_error)]
# extract optimal number of m based on OOB
opt.m_oob_tr = results_df_tr$m[which.min(results_df_tr$OOB_error)]

# set seed for reproducible results
set.seed(33)
# fit the best CV Random Forest
best_CV_rf = randomForest(lpsa ~ ., data=x_train, mtry=opt.m_cv_tr,
                          xtest = dplyr::select(x_test, -lpsa),
                          ytest = y_test, importance=TRUE)
best_CV_rf
```

```
##
## Call:
##  randomForest(formula = lpsa ~ ., data = x_train, mtry = opt.m_cv_tr,      xtest = dplyr::select(x_test
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 6
##
##           Mean of squared residuals: 0.7421963
##                     % Var explained: 47.91
##                        Test set MSE: 0.42
##                     % Var explained: 62.22
```

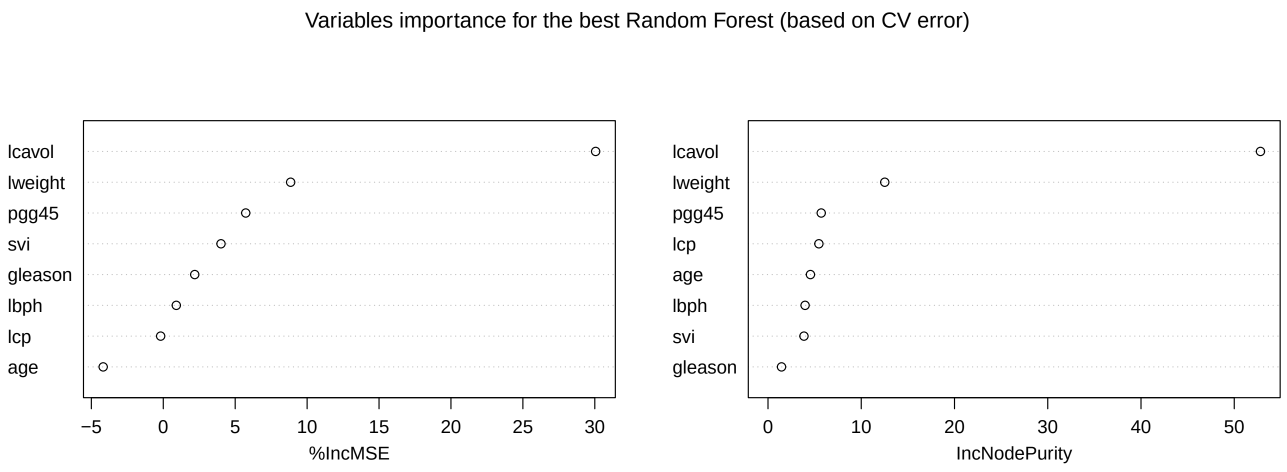
```
# fit the best OBB Random Forest
best_OOB_rf = randomForest(lpsa ~ ., data=x_train, mtry=opt.m_oob_tr,
                           xtest = dplyr::select(x_test, -lpsa),
                           ytest = y_test, importance=TRUE)
best_OOB_rf
```

```
##
## Call:
##  randomForest(formula = lpsa ~ ., data = x_train, mtry = opt.m_oob_tr,      xtest = dplyr::select(x_tes
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 4
##
```

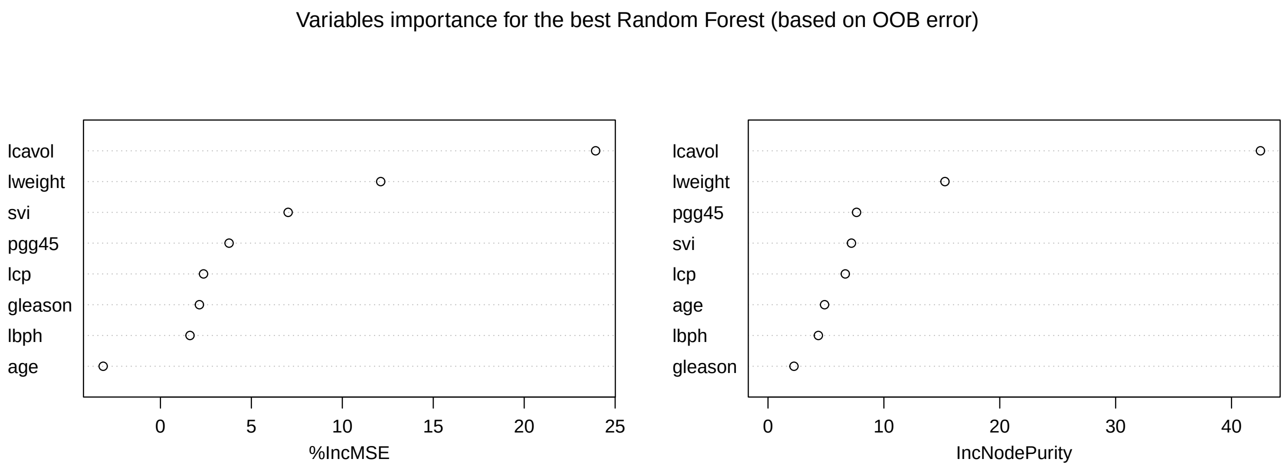
```
##            Mean of squared residuals: 0.7223264
##                      % Var explained: 49.31
##                         Test set MSE: 0.4
##                      % Var explained: 63.27
```

And we can also visualize again the variables importance:

```r
op = par(mfrow=c(2, 1))
varImpPlot(best_CV_rf,
           main="Variables importance for the best Random Forest (based on CV error)")
```



```r
varImpPlot(best_OOB_rf,
           main="Variables importance for the best Random Forest (based on OOB error)")
```



Once again the performances are very similar. Moreover, they exceeds the performances of the previous regression trees (both the prune and unpruned versions fitted and tested using the same train-test partitions). Even in this case, by repeating the split test-train, both the optimal number of m and the performance may vary slightly. Besides that, also the variables importance order can slightly change. However the top 2-3 most important variables (especially `lcavol` and `lweight` as 1st and 2nd) seem to be always represented by the same set of variables.

Moreover, the models using the optimal values of m selected via the OOB error seem to be generally slightly more accurate, presumably due to the fact that the OOB approaches the LOOCV error. However, we cannot select the optimal model using only a few train-test splitting; indeed, cross-validation will be required for this purpose.

3. Fit boosted regression trees making a selection of the number of boosting iterations (n.trees) by CV. Interpret your selected optimal model.

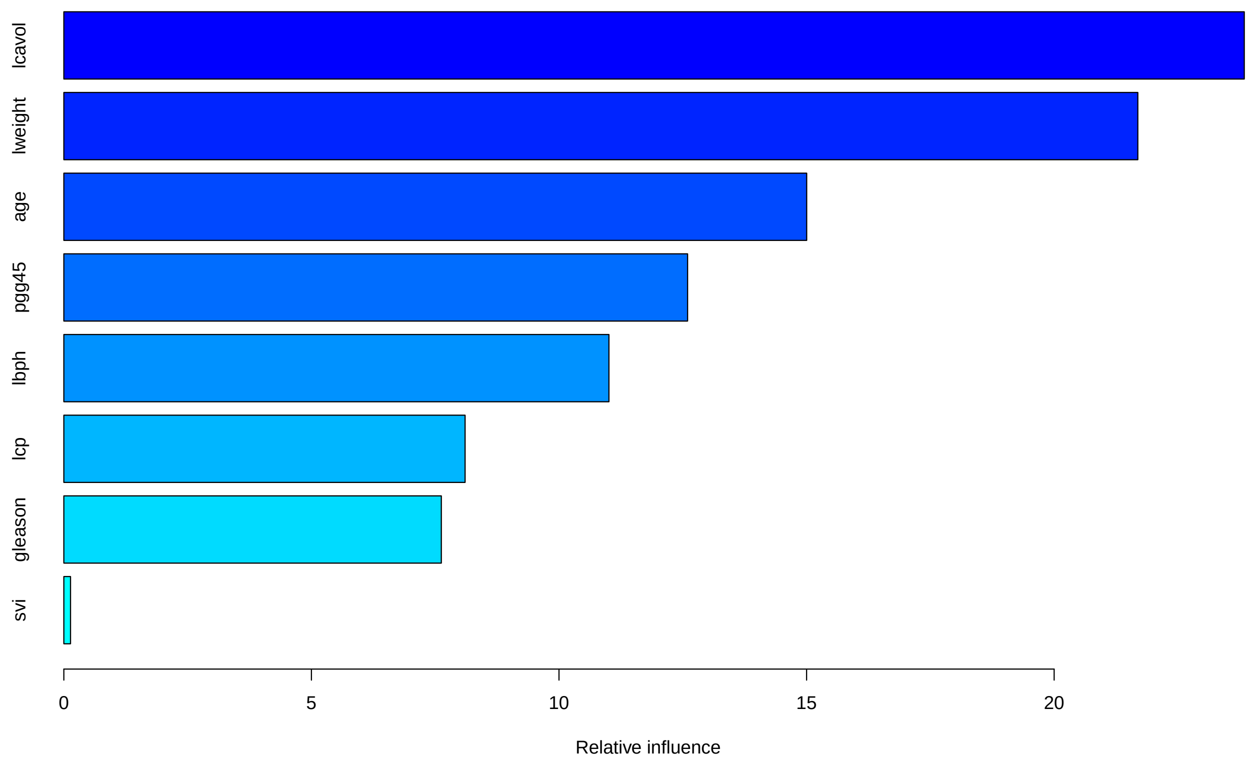
Another technique to reduce tees variance is boosting. In particular this approach is based on the idea of aggregating sequentially grown trees in order to reduce variance, thus each tree is grown based on the information from previously grown trees. Moreover, the advantage compared to Random Forest is that, using boosting, some kind of variable selection is done.

For the purpose of the assignment we will use Gradient Boosting algorithm, which is implemented in R by the `gbm` library.

We will select the optimal model by tuning the parameter M (`n.trees` in the model), which represents the number of trees of boosting iterations. This parameter is connected to the bias-variance trade-off since too large M (the default for `gbm` is 5000) tends to overfit the data.

For the selection of number of boosting iterations (`n.trees`) we decide to use k-fold cross-validation on the training set, which is already implemented in the `gbm()` function:

```r
# reset mfrow()
op = par(mfrow=c(1, 1))

# set seed for reproducible results
set.seed(1)
# fit a boosted regression trees using 10-fold CV
boosted_reg = gbm(lpsa ~ ., data=x_train, distribution="gaussian", n.trees=5000,
                  interaction.depth=4, cv.folds=10)
# show summary
knitr::kable(summary(boosted_reg))
```



	var	rel.inf
lcavol	lcavol	23.8399287
lweight	lweight	21.6901967
age	age	15.0021829
pgg45	pgg45	12.5969057
lbph	lbph	11.0089672
lcp	lcp	8.1035477
gleason	gleason	7.6235181

	var	rel.inf
svi	svi	0.1347531

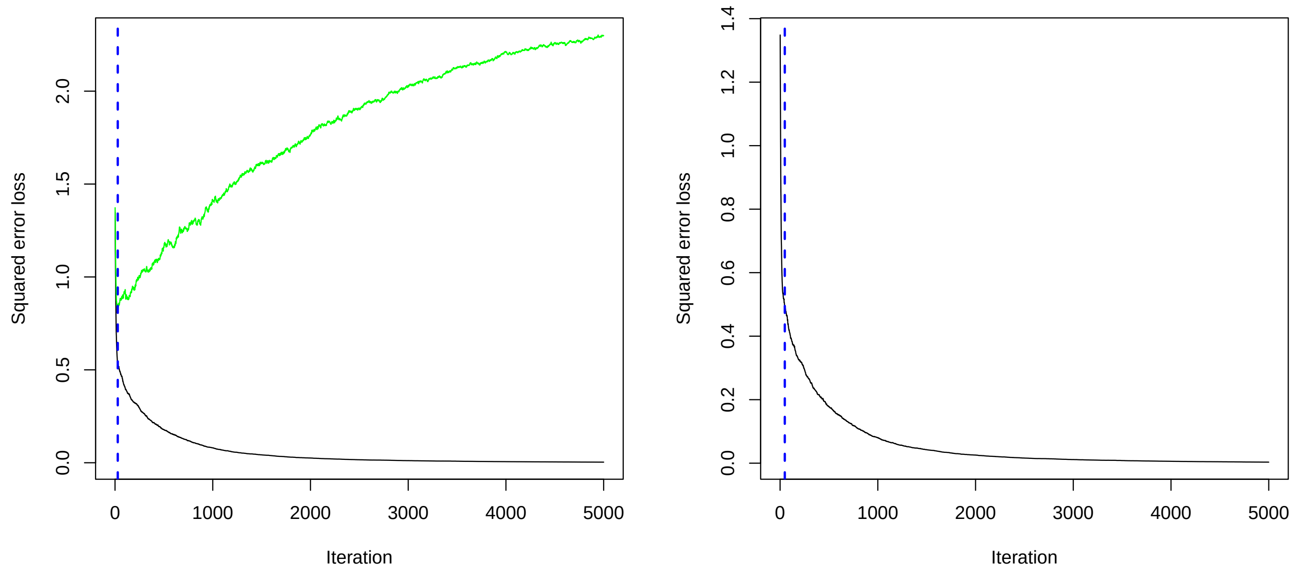
The function show us the relative influence of each predictor on the output: in this case the top two variables are `lcavol` and `lweight`, which are the same as of the previous Random Forest models.

Using the `gbm.perf()` function allow us to find the optimal number of boosting iterations, both by evaluating the cross-validation error and the OOB:

```
op = par(mfrow=c(1,2))
# optimal number of boosting iterations based CV
cv_opt_iter = gbm.perf(boosted_reg, method = "cv")
cv_opt_iter
```

```
## [1] 28
```

```
# optimal number of boosting iterations based OOB
oob_opt_iter = gbm.perf(boosted_reg, method = "OOB")
```



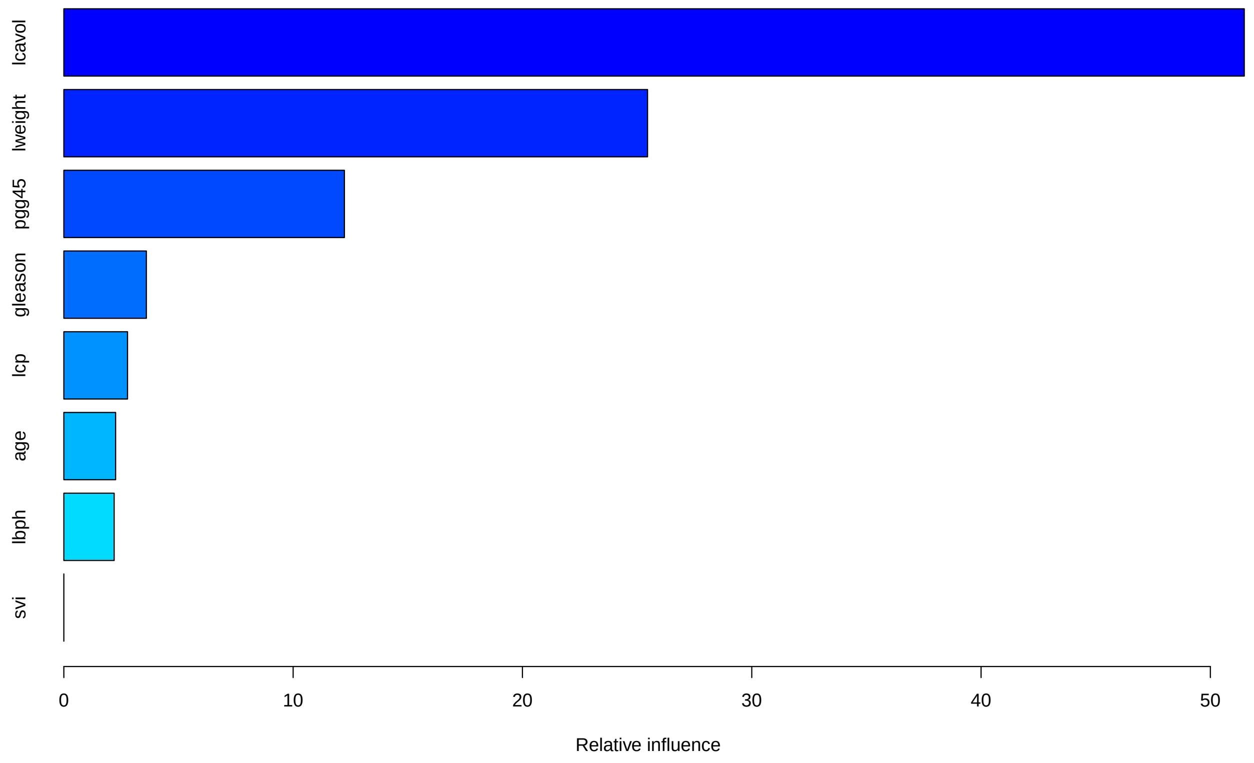
```
oob_opt_iter
```

```
## [1] 48
## attr(,"smoother")
## Call:
## loess(formula = object$oobag.improve ~ x, enp.target = min(max(4,
##     length(x)/10), 50))
##
## Number of Observations: 5000
## Equivalent Number of Parameters: 39.99
## Residual Standard Error: 0.002683
```

As we can notice also by the output plots (squared error loss vs number of iterations), the optimal number of trees for cross-validation is 28, while for OOB is 48. In particular the green line indicates the test error (validation in our case), the black line represents the training error and the blue dotted line points the optimum number of iterations. In the CV based plot we can see how the intersection between the two error lines define the optimal number of iterations.

We can finally see how the models perform on our test set with both CV and OOB based number of iterations:

```r
# reset mfrow()
op = par(mfrow=c(1, 1))

# set seed for reproducible results
set.seed(14)
# fit a boosted regression with the optimal number of iteration based on CV
boosted_reg_cv = gbm(lpsa ~ ., data=x_train, distribution="gaussian",
                     n.trees=cv_opt_iter, cv.folds=0)
# show summary
knitr::kable(summary(boosted_reg_cv))
```

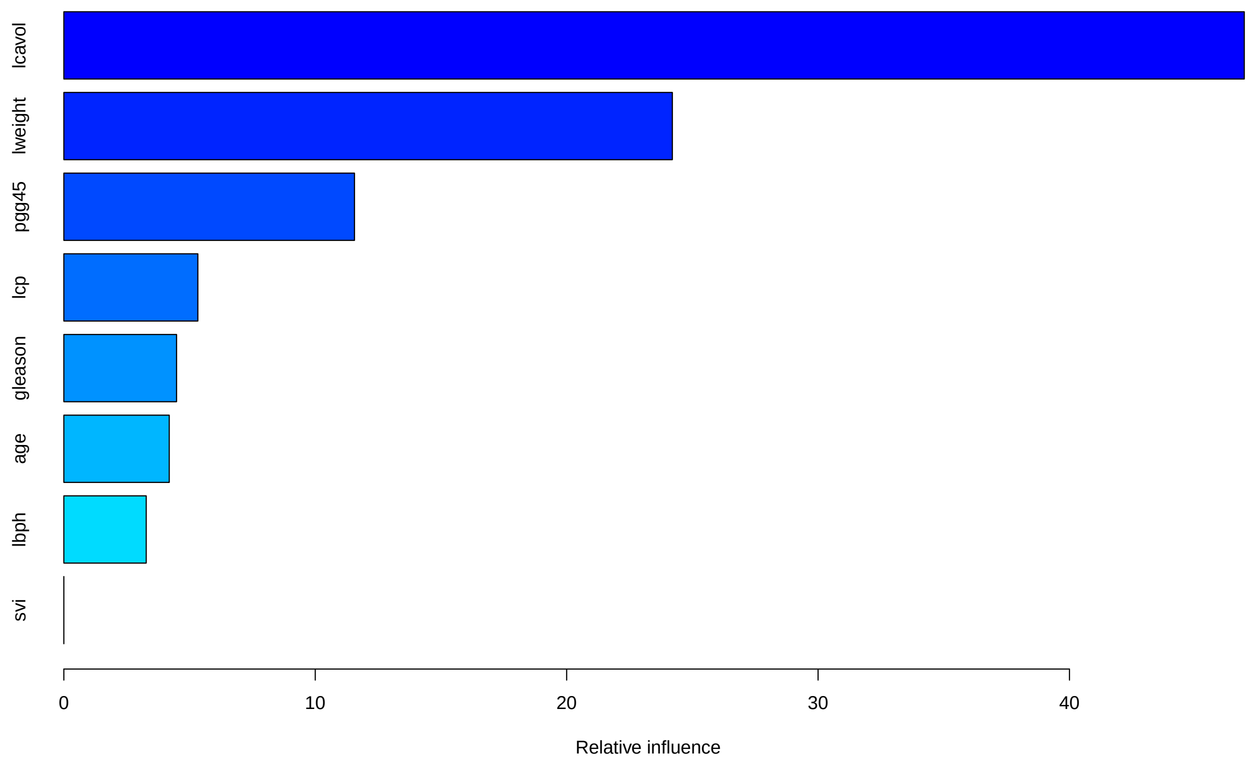


	var	rel.inf
lcavol	lcavol	51.479127
lweight	lweight	25.455589
pgg45	pgg45	12.236196
gleason	gleason	3.599570
lcp	lcp	2.777248
age	age	2.257570
lbph	lbph	2.194701
svi	svi	0.000000

```r
# make prediction on test using the model
yhat = predict(boosted_reg_cv, newdata=x_test, n.trees=cv_opt_iter)
# calculate MSE
boosted_reg_cv_mse = mean((yhat - y_test)^2)
boosted_reg_cv_mse
```

```
## [1] 0.5867136
```

```r
set.seed(14)
# fit a boosted regression with the optimal number of iteration based on OOB
boosted_reg_OOB = gbm(lpsa ~ ., data=x_train, distribution="gaussian",
                      n.trees=oob_opt_iter, cv.folds=0)
# show summary
knitr::kable(summary(boosted_reg_OOB))
```



	var	rel.inf
lcavol	lcavol	46.952936
lweight	lweight	24.203904
pgg45	pgg45	11.560458
lcp	lcp	5.330332
gleason	gleason	4.482983
age	age	4.192648
lbph	lbph	3.276740
svi	svi	0.000000

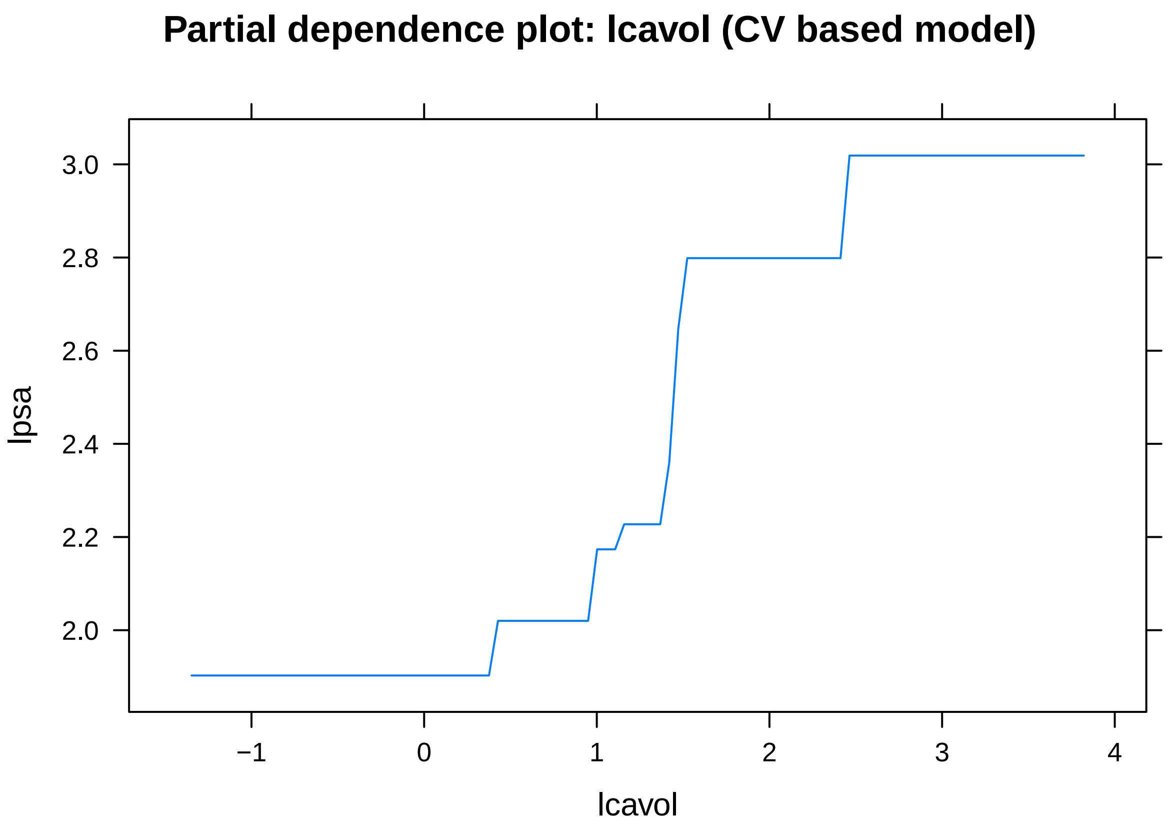
```r
# make prediction on test using the model
yhat = predict(boosted_reg_OOB, newdata=x_test, n.trees=oob_opt_iter)
# calculate MSE
boosted_reg_OOB_mse = mean((yhat - y_test)^2)
boosted_reg_OOB_mse
```

```
## [1] 0.5709084
```

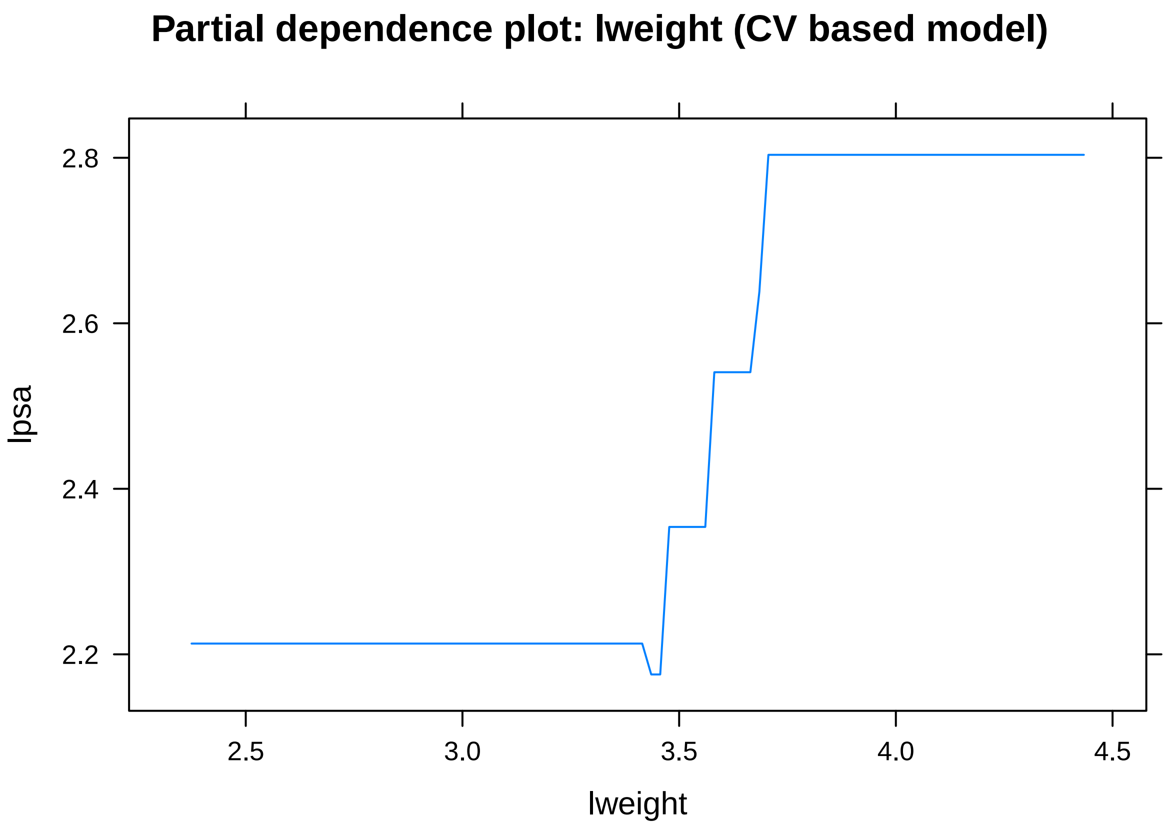
Again, the top two variables in terms of relative influence are still `lcavol` and `lweight` for both models.

We can also evaluate the marginal effect of these two variables by producing a partial dependence plot:

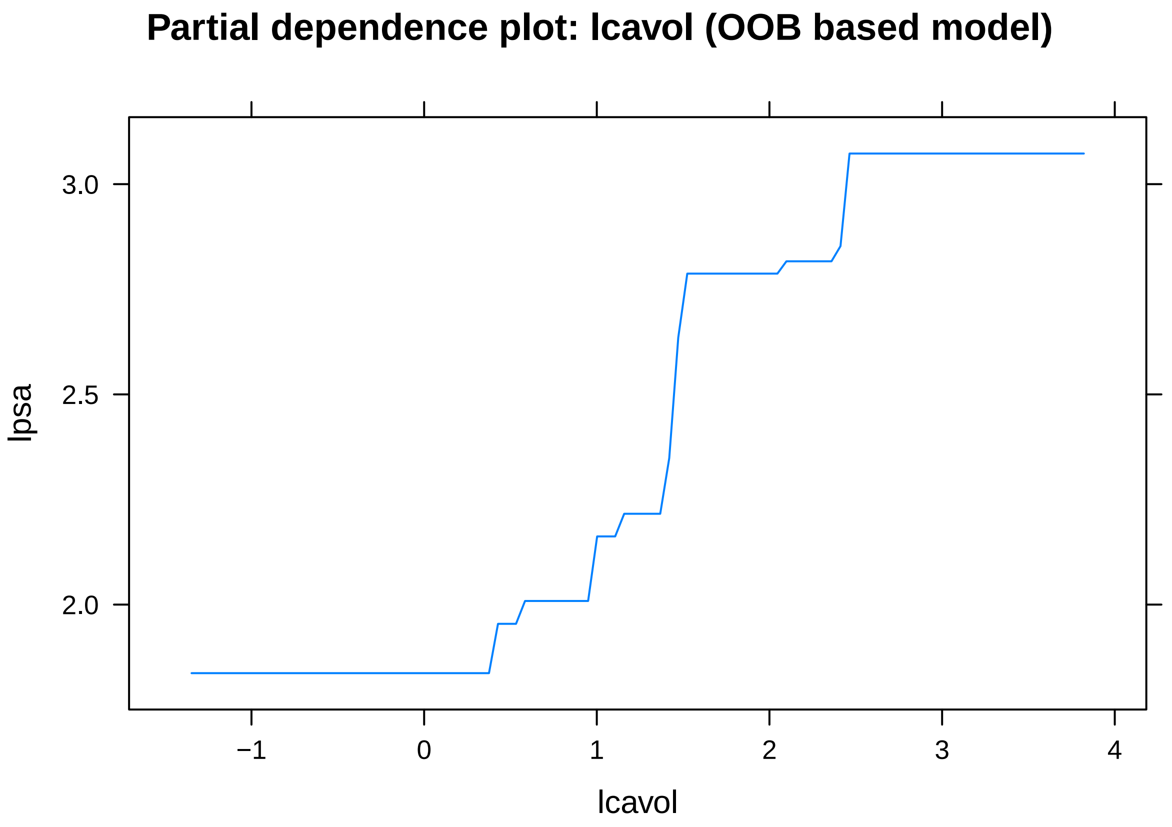
```r
# partial dependence plot CV based model
plot(boosted_reg_cv, i="lcavol",
     main ="Partial dependence plot: lcavol (CV based model)",
     ylab="lpsa")
```



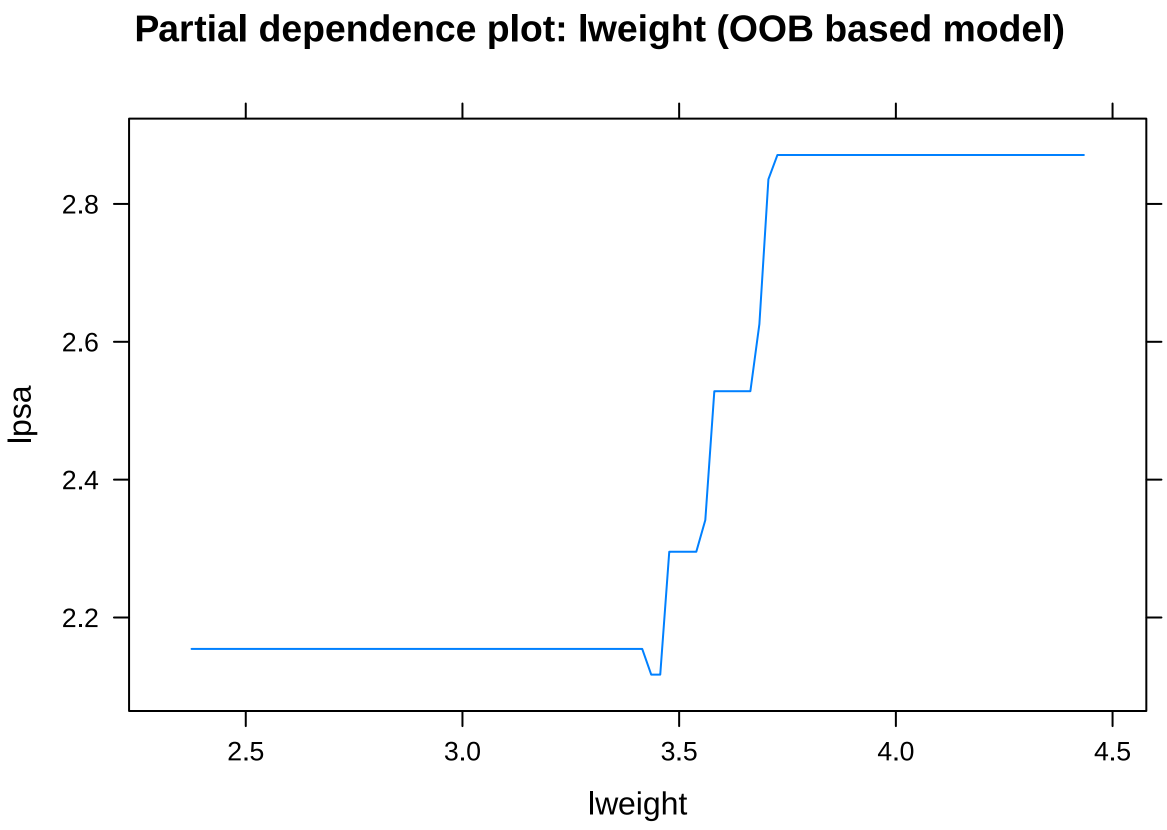
```r
plot(boosted_reg_cv, i="lweight",
     main ="Partial dependence plot: lweight (CV based model)",
     ylab="lpsa")
```



```
# partial dependence plot OOB based model
plot(boosted_reg_OOB, i="lcavol",
     main ="Partial dependence plot: lcavol (OOB based model)",
     ylab="lpsa")
```



```
plot(boosted_reg_OOB, i="lweight",
     main ="Partial dependence plot: lweight (OOB based model)",
     ylab="lpsa")
```



These plots represent the effect that the chosen variable has on the response variable. Since the performances of OOB and CV based boosted regression are similar, these plots, and thus the effect of the most important variables on the response variable are quite similar too.

In both cases it the values of `lpsa` are not increasing until `lcavol` reaches about 0.5, after that the response variable increases with `lcavol`; lastly, reaching a value above 2.5 for `lcavol` the value of `lpsa` stabilize at just above 3.

In the `lweight` partial dependence plots of both the models the value of `lpsa` is stable around 2.2 until `lweight` reaches about 3.4, after that we can notice a little descending peaks, but then the response variable value will increase with `lweight` until about 3.35. At this point for both models the value of `lpsa` stabilizes at about 2.8.

As in the previous cases, the OBB and CV based models performance seems to be quite the same, but once again, repeating the train-test split may lead to different results. However, the top two variables seem to be always `lcavol` and `lweight`.

4. Compare the performance of the three methods (cost-complexity decision trees, random forests and boosting) using cross-validation. Make sure that the model complexity is re-optimized at each choice of the training set (either using another CV or using the OOB error).

The previous train-test splits were useful in order to show, interpret and have a first approach with the outputs of the various models' functions, as well as to get an initial idea of their performance. However, to properly evaluate the performance of the models and to eventually select the optimal model, it is essential to use a cross-validation.

We can finally properly compare methods using 10-folds cross validation. This choice stems from the fact that leave-one-out cross-validation proves to be rather expensive and time consuming in this case (especially for Random Forest and boosted models).

We start by initializing the dataframe that will contain the results and folds we are going to use for cross-validation.

```r
# initialize final results dataframe
final_results = data.frame(model = c("Unpruned decision tree", "Pruned decision tree",
                                     "Random Forest (CV error based m)",
                                     "Random Forest (OOB error based m)",
                                     "Boosted regression (CV based)",
                                     "Boosted regression (OOB based)"),
                           error=0, type="CV MSE")

# set seed for reproducible folds
set.seed(14)
# recreate 10 set of index for training set composition
flds_LOOCV = createFolds(prostate_df$lpsa, k = dim(prostate_df)[1], list = T,
                         returnTrain = T)

# set seed for reproducible folds
set.seed(14)
# recreate 10 set of index for training set composition
flds_10_cv = createFolds(prostate_df$lpsa, k = 10, list = T, returnTrain = T)
```

First, we evaluate the performance of cost-complexity decision trees re-optimizing the tree complexity using cross-validation:

```r
# initialize folds' MSE lists
list_unpr_mse = c()
list_pr_mse = c()

# iterate over folds
for (f in flds_10_cv) {
  # set seed for reproducible randomForest() results
  set.seed(seed)
  # define index based on the i fold
  train_idx = f
  # split train and test sets according to the indexes
  train = prostate_df[train_idx,]
  test = prostate_df[-train_idx,]
  y_ts = test$lpsa

  # fit a regression tree on the training set
  unpruned_tree = tree(lpsa ~ ., train)
  # perform CV to find the optimal number of terminal nodes
  cv.pros_tr = cv.tree(unpruned_tree)
  # get the optimal size
  opt.size = cv.pros_tr$size[which.min(cv.pros_tr$dev)]
  # prune the tree
  pruned_tree = prune.tree(unpruned_tree, best=opt.size)

  # test the cost-complexity decision trees models
  # unpruned tree
  yhat_unpr = predict(unpruned_tree, newdata=test)
  # calculate MSE
  unpr_mse = mean((yhat_unpr - y_ts)^2)
  # pruned tree
  yhat_pr = predict(pruned_tree, newdata=test)
  # calculate MSE
  pr_mse = mean((yhat_pr - y_ts)^2)

  # add MSEs to the lists
  list_unpr_mse = append(list_unpr_mse, unpr_mse)
  list_pr_mse = append(list_pr_mse, pr_mse)
}
```

```r
# add CV MSE to the final results df
final_results[1,]$error = mean(list_unpr_mse)
final_results[2,]$error = mean(list_pr_mse)

knitr::kable(final_results[1:2,])
```

model	error	type
Unpruned decision tree	0.7292767	CV MSE
Pruned decision tree	0.7471775	CV MSE

The performances of the unpruned and the pruned tree are very similar.

Similarly, we can evaluate the performances of Random Forests. In particular, we will report both the performances of models re-optimized via CV error and those optimized via OOB.

```r
# initialize folds' MSE lists
list_CV_rf_mse = c()
list_OOB_rf_mse = c()

# iterate over folds
for (f in flds_10_cv) {
  # set seed for reproducible randomForest() results
  set.seed(seed)
  # define index based on the i fold
  train_idx = f
  # split train and test sets according to the indexes
  train = prostate_df[train_idx,]
  test = prostate_df[-train_idx,]
  y_ts = test$lpsa

  # create a CV/OBB errors df
  temp_res_m = data.frame(nvar=nvar, CV_error = 0, OOB_error = 0)

  # iterate over m values
  for (m in nvar) {
    # CV optimization
    set.seed(seed)
    # recreate 10 set of index for training-validation sets composition
    flds_val = createFolds(train$lpsa, k = 10, list = T, returnTrain = T)
    # create an empty list to store each folds MSE
    list_rf_mse = c()

    for (i in flds_val) {
      # set seed for reproducible randomForest() results
      set.seed(seed)
      # define index based on the i fold
      train_idx_rf = i
      # split train and test sets according to the indexes
      train_rf = train[train_idx_rf,]
      test_rf = train[-train_idx_rf,]
      y_ts_rf = test_rf$lpsa

      # fit a random forest on k - 1 folds and test on the remaining fold using
      # m number of variables
      rf = randomForest(lpsa ~ ., data=train_rf, mtry=m,
                        xtest = dplyr::select(test_rf, -lpsa),
                        ytest = y_ts_rf, importance=TRUE)

      # extract prediction
      temp_yhat = rf$test$predicted
      # compute MSE
      temp_mse = mean((temp_yhat - y_ts_rf)^2)
      # add MSE to the mse_list
      list_rf_mse = append(list_rf_mse, temp_mse)
    }

    # average k-folds cross validation errors
    cv_error = mean(list_rf_mse)
    # add to the df
    temp_res_m[temp_res_m$nvar==m,]$CV_error = cv_error

    # OOB optimization
    # set seed for reproducible randomForest() results
    set.seed(seed)
    # fit a random forest on the whole dataset using m number of variables
    rf = randomForest(lpsa ~ ., data=train, mtry=m, importance=TRUE)
    # extract OOB value
    oob = rf$mse[length(rf$mse)]
    # add to the df
    temp_res_m[temp_res_m$nvar==m,]$OOB_error = oob

  }

  # extract optimal value of m based on CV error
  opt_m_cv = temp_res_m[temp_res_m$CV_error == min(temp_res_m$CV_error),]$nvar
  # extract optimal value of m based on OOB error
  opt_m_oob = temp_res_m[temp_res_m$OOB_error == min(temp_res_m$OOB_error),]$nvar

  # set seed for reproducible results
  set.seed(seed)
  # fit the best CV Random Forest
  best_CV_rf = randomForest(lpsa ~ ., data=train, mtry=opt_m_cv,
                            xtest = dplyr::select(test, -lpsa),
                            ytest = y_ts, importance=TRUE)
  # extract prediction
  temp_yhat = best_CV_rf$test$predicted
  # compute MSE
  temp_mse = mean((temp_yhat - y_ts)^2)
  # add MSE to the mse list
  list_CV_rf_mse = append(list_CV_rf_mse, temp_mse)

  # fit the best OOB Random Forest
  best_OOB_rf = randomForest(lpsa ~ ., data=train, mtry=opt_m_oob,
                             xtest = dplyr::select(test, -lpsa),
                             ytest = y_ts, importance=TRUE)
  # extract prediction
  temp_yhat = best_OOB_rf$test$predicted
  # compute MSE
  temp_mse = mean((temp_yhat - y_ts)^2)
  # add MSE to the mse list
  list_OOB_rf_mse = append(list_CV_rf_mse, temp_mse)
}

# add cross-validation folds' average MSE for the two optimized Random Forest
final_results[3,]$error = mean(list_CV_rf_mse)
final_results[4,]$error = mean(list_OOB_rf_mse)

knitr::kable(final_results[1:4,])
```

model	error	type
Unpruned decision tree	0.7292767	CV MSE
Pruned decision tree	0.7471775	CV MSE
Random Forest (CV error based m)	0.5838541	CV MSE
Random Forest (OOB error based m)	0.5801757	CV MSE

Again, the performances of the two models (Random Forest optimized via CV and OOB errors) seem to be essentially the same.

Finally, we can do the same procedure for boosting:

```r
# initialize folds' MSE lists
boosted_CV_mse_list = c()
boosted_OOB_mse_list = c()

# iterate over folds
for (f in flds_10_cv) {
  # set seed for reproducible results
  set.seed(seed)
  # define index based on the i fold
  train_idx = f
  # split train and test sets according to the indexes
  train = prostate_df[train_idx,]
  test = prostate_df[-train_idx,]
  y_ts = test$lpsa

  # fit a boosted regression trees on the training set using 10-fold CV
  boosted_reg = gbm(lpsa ~ ., data=train, distribution="gaussian", n.trees=5000,
                    interaction.depth=4, cv.folds=10, )

  # find optimal number of boosting iterations based CV
  cv_opt_iter = gbm.perf(boosted_reg, method = "cv", plot.it=F)
  # find optimal number of boosting iterations based OOB
  oob_opt_iter = gbm.perf(boosted_reg, method = "OOB", plot.it=F)

  # fit a boosted regression with the optimal number of iteration based on CV
  boosted_reg_cv = gbm(lpsa ~ ., data=train, distribution="gaussian",
                       n.trees=cv_opt_iter, cv.folds=0)
  # make prediction on test
  yhat_cv = predict(boosted_reg_cv, newdata=test, n.trees=cv_opt_iter)
  # calculate MSE
  boosted_reg_cv_mse = mean((yhat_cv - y_ts)^2)
  # add to the MSE list
  boosted_CV_mse_list = append(boosted_CV_mse_list, boosted_reg_cv_mse)

  # fit a boosted regression with the optimal number of iteration based on OOB
  boosted_reg_oob = gbm(lpsa ~ ., data=train, distribution="gaussian",
                        n.trees=oob_opt_iter, cv.folds=0)
  # make prediction on test
  yhat_oob = predict(boosted_reg_oob, newdata=test, n.trees=oob_opt_iter)
  # calculate MSE
  boosted_reg_oob_mse = mean((yhat_oob - y_ts)^2)
  # add to the MSE list
  boosted_OOB_mse_list = append(boosted_OOB_mse_list, boosted_reg_oob_mse)
}

# add cross-validation folds' average MSE for the two optimized boosted models
final_results[5,]$error = mean(boosted_CV_mse_list)
final_results[6,]$error = mean(boosted_OOB_mse_list)
```

Note that we have not tuned the shrinkage parameter. however, we know that this parameter is correlated with the M (`n.trees`) parameter, which we have tuned instead. Hence our strategy was to select a small number for the shrinkage parameter (we used the default, so 0.001) and let M grow until there is no significant reduction in loss (reaching the optimal number of iteration based both on CV and OOB). Learning will be slower, as will the procedure, but this will not be a big problem since we are not dealing with a large dataset.

Once the models have been evaluated using cross-validation, we can finally compare them. We now sort the results dataframe to see which models have recorded the lowest MSE.

```
knitr::kable(final_results[order(final_results$error),])
```

	model	error	type
4	Random Forest (OOB error based m)	0.5801757	CV MSE
3	Random Forest (CV error based m)	0.5838541	CV MSE
6	Boosted regression (OOB based)	0.6211329	CV MSE
5	Boosted regression (CV based)	0.6708198	CV MSE
1	Unpruned decision tree	0.7292767	CV MSE
2	Pruned decision tree	0.7471775	CV MSE

5. Draw some general conclusions about the analysis and the different methods that you have considered. As we can see from the results dataframe, the Random Forest models (both the one optimized via CV and the one optimized via OOB) seem to have the highest performance, achieving the lowest cross-validation MSE. This is followed by the boosted models; here in particular the difference between the CV-optimized model and the OOB-optimized model seems to be more pronounced, in fact the difference between MSEs is greater than for Random Forests. Finally, we have the cost-complexity decision trees, for which the complete tree performs better than the pruned tree. As we said before, models optimized by OOB error tend to perform better; this is probably due to the fact that it approaches the leave-one-out cross-validation error.

As we know, cost-complexity decision trees suffer from high variance and therefore tend to overfit more than models such as Random Forest and boosted models. The latter models have in fact quite similar performances in our case. However, the boosted models seem to tend more to overfitting: this could be due to the fact that some noise in the data affect them more than Random Forests.

Furthermore, we have understood how some of the variables, such as `lcavol` and `lweight`, play an important role in the prediction of `lpsa`. In the previous points we can observe how these variables were:

- at the stump of the trees (in the case of cost-complexity decision trees),
- at the top of the lists of variable importance (in the case of Random Forests),
- at the top of the lists of relative influence (in the case of boosted models).

Note that in the data exploration, looking at the output of `ggpairs()` and the correlation matrix, we already had a first intuition about the importance that the different variables would have in the models.

Out of curiosity, I lastly checked if Shrinkage models could perform better than tree-based models. Applying the same scheme and fold as the cross-validation used for the previous models, a properly tuned LASSO model performs better than Random Forest, achieving a cross-validation MSE of 0.55, while Ridge regression perform as well as to Random Forest:

```
# initialize MSE lists
list_MSE_lasso = c()
list_MSE_ridge = c()


# iterate over folds
for (f in flds_10_cv) {
  # define index based on the i fold
  train_idx = f
  x_tr_full = prostate_df[train_idx, ]
  x_ts_full = prostate_df[-train_idx, ]
  # glmnet() require
  # matrix creation
  x_tr_only_pred = model.matrix(lpsa ~ ., data=x_tr_full)[, -1]
  x_ts_only_pred = model.matrix(lpsa ~ ., data=x_ts_full)[, -1]
  # we store the y values
  y_tr = x_tr_full$lpsa
  y_ts = x_ts_full$lpsa

  # Lasso
  # fit lasso model on training
  lasso_mod = glmnet(x_tr_only_pred, y_tr, alpha=1)

  set.seed(seed)
  # CV for lamnda tuning
  cv.out = cv.glmnet(x_tr_only_pred, y_tr, alpha=1, lambda=lasso_mod$lambda)
  # extract best lambda
  bestlam = cv.out$lambda.min

  # make predition on test set using optimal lambda
  lasso_pred = predict(lasso_mod, s=bestlam, newx=x_ts_only_pred)
  # calculate MSE
  lasso_mse = mean((lasso_pred - y_ts)^2)
  # add to MSE list
  list_MSE_lasso = append(list_MSE_lasso, lasso_mse)

  # extract coefficient
  lasso_coef = predict(lasso_mod, type="coefficients", s=bestlam)
  # uncomment the line below to see which coefficients are zero and which not
  # print(lasso_coef)

  # Ridge
  grid <- 10^seq(10, -2, length=100) # choice of the text book (from 10^-2 to 10^10)
  # to cover the entire range of scenarios (from the null model to least squares fit)
  # fit ridge regression model on training data
  # fit ridge regression
  ridge_mod = glmnet(x_tr_only_pred, y_tr, alpha=0, lambda=grid)
  # prediction on lambda = 0
  lsr_rr_pred = predict(ridge_mod, s=0, newx=x_ts_only_pred, exact=TRUE, x=x_tr_only_pred, y=y_tr)
  # calculate MSE
  lsr_rr_mse = mean((lsr_rr_pred - y_ts)^2)
  # add to the MSE list
  list_MSE_ridge = append(list_MSE_ridge, lsr_rr_mse)

}

# Lasso MSE
mean(list_MSE_lasso)
```

```
## [1] 0.5539088
```

```
# Rodge MSE
mean(list_MSE_ridge)
```

```
## [1] 0.5825092
```

Considering this result leads us to think that LASSO might be able to better capture the complexity of the data.