Bagging, random forest, boosting, and cubist

Chapter 8-Part II Regression Trees and Rule-Based Models

Bagging

- *Bootstrap aggregation*, or *bagging*, is a general-purpose procedure for reducing the variance of a statistical learning method; we introduce it here because it is particularly useful and frequently used in the context of decision trees.
- Recall that given a set of n independent observations $Z_1, \ldots, Z_n$, each with variance σ^2, the variance of the mean $\bar{Z}$ of the observations is given by σ^2/n.

$$Var(Z_1) = \sigma^2, \quad Var(Z_2) = \sigma^2, \quad \ldots$$

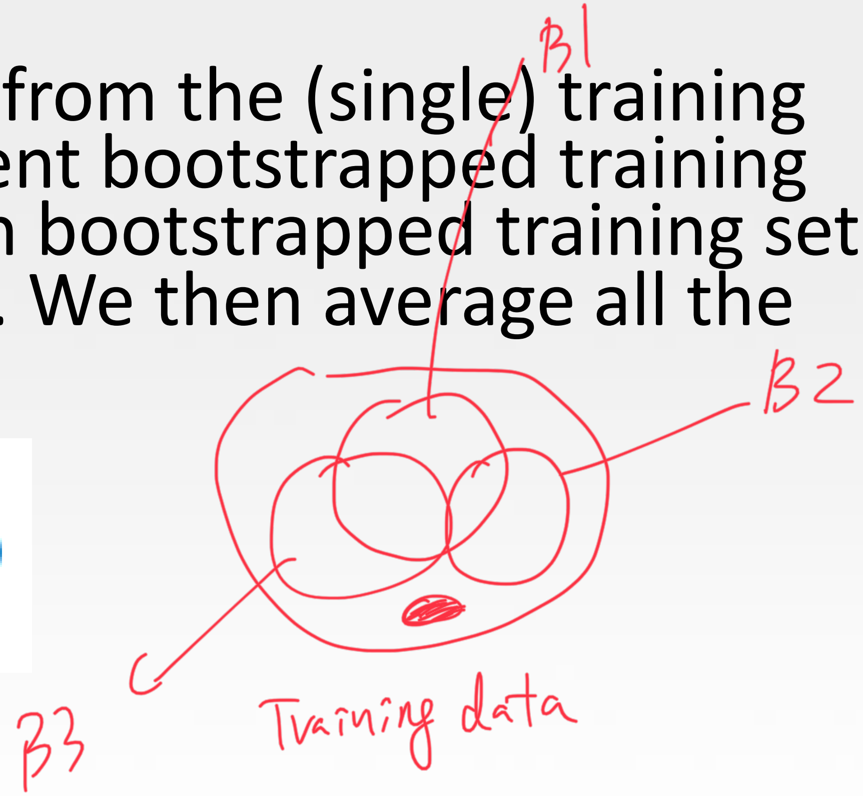
$$\bar{Z} = \frac{1}{n}\sum_{i=1}^{n} X_i \Rightarrow Var(\bar{Z}) = Var\left(\frac{1}{n}\sum_{i=1}^{n} Z_i\right) = \frac{1}{n^2} Var\left(\sum_{i=1}^{n} Z_i\right)$$

$$= \frac{1}{n^2}\left(Var(Z_1) + Var(Z_2) + \ldots Var(Z_n)\right) = \frac{1}{n^2}\cdot n\sigma^2 = \boxed{\frac{\sigma^2}{n}}$$

Bagging

- In other words, *averaging a set of observations reduces variance*. Of course, this is not practical because we generally do not have access to multiple training sets.
- We can bootstrap, by taking repeated samples from the (single) training data set. In this approach we generate *B* different bootstrapped training data sets. We then train our method on the *b*th bootstrapped training set in order to get $\widehat{f}^{*b}(x)$ the prediction at a point *x*. We then average all the predictions to obtain

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^{*b}(x)$$

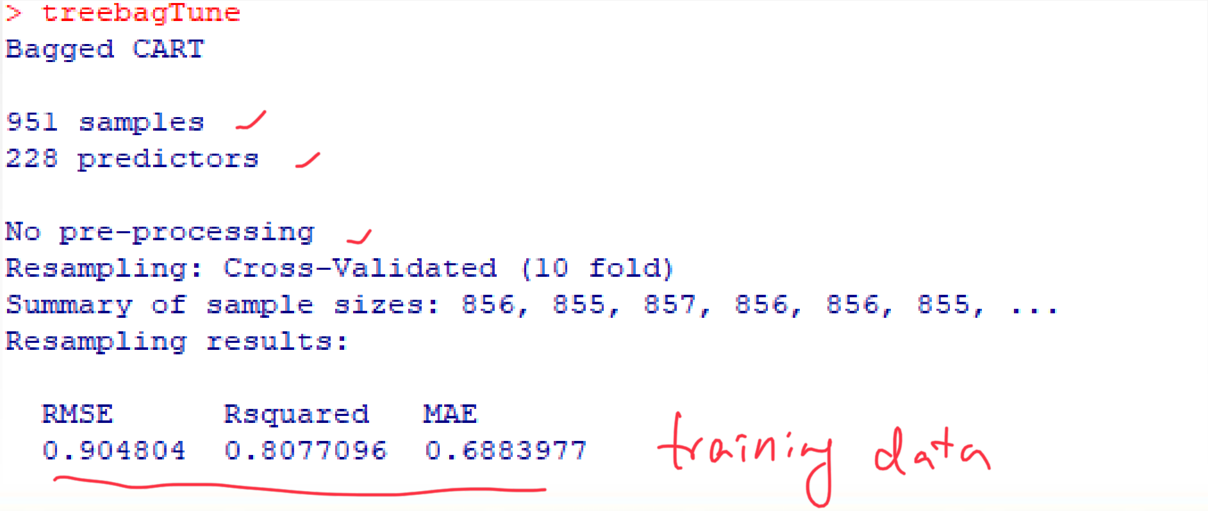


- This is called bagging.

Bagged trees

```
set.seed(100)

treebagTune <- train(x = solTrainXtrans, y = solTrainY,
                     method = "treebag",
                     nbagg = 50,
                     trControl = ctrl)

treebagTune
```

B = 50



```
> treebagTune
Bagged CART

951 samples
228 predictors

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 856, 855, 857, 856, 856, 855, ...
Resampling results:

  RMSE      Rsquared   MAE
  0.904804  0.8077096  0.6883977
```

training data

Out-of-bag (OOB) error

- The key to bagging is that trees are repeatedly fit to bootstrapped subsets of the observations. On average, each bagged tree makes use of around two-thirds of the observations.

$2/3$

OOB error

- The remaining one-third of the observations not used to fit a given bagged tree are referred to as the *out-of-bag* (OOB) observations.
- We can predict the response for the *i*th observation using each of the trees in which that observation was OOB. This will yield around *B*/3 predictions for the *i*th observation, which we average to obtain a single OOB prediction.
- An OOB error can be computed by averaging the *n* single OOB predictions.
- OOB error is virtually equivalent to LOOCV if *B* is large.
- Bagging improves prediction accuracy at the expense of interpretability!

Boosting

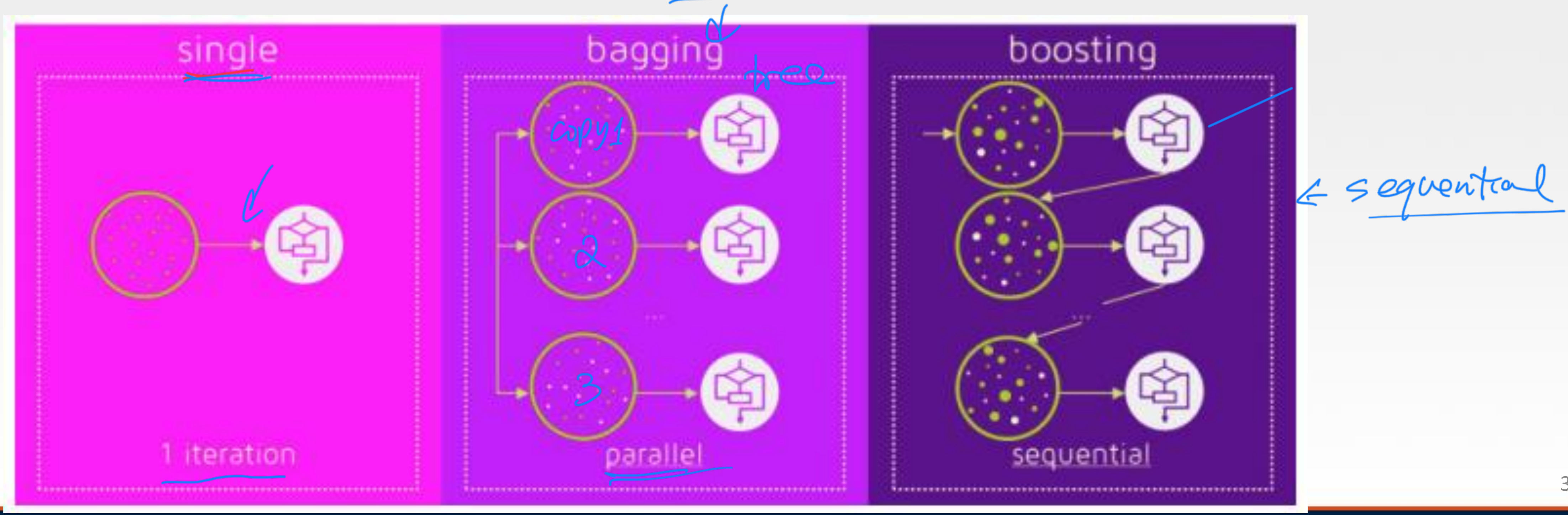
- Like bagging, boosting is a general approach that can be applied to many statistical learning methods for regression or classification.
- Recall that bagging involves creating multiple copies of the original training data set using the bootstrap, fitting a separate decision tree to each copy, and then combining all of the trees in order to create a single predictive model.
- Each tree is built on a bootstrap data set, independent of the other trees.

Boosting

- Boosting works in a similar way, except that the trees are grown sequentially and slowly: each tree is grown using information from previously grown trees.
- Boosting does not involve bootstrap sampling; instead each tree is fit on a modified version of the original data set.

Bagging vs Boosting

- No clear winner; usually depends on the data
- Bagging is computationally more efficient than boosting (note that bagging can train the B models in parallel, boosting cannot)



What is the idea behind this procedure?

- To avoid potential overfitting of a single tree to the data, the boosting approach *learns slowly*.
- Given the current model, we fit a decision tree to the residuals from the model. We then add this new decision tree into the fitted function in order to update the residuals.
- Each of these trees can be rather small, with just a few terminal nodes, determined by the parameter d in the algorithm.
- By fitting small trees to the residuals, we slowly improve the model in areas where it does not perform well. The shrinkage parameter λ slows the process down even further, allowing more and different shaped trees to attack the residuals.

Tuning parameters of Boosting

- There are three tuning parameters that we must carefully consider
  - The number of trees K
  - The shrinkage parameter λ
  - The number *d* splits in each tree

The number of trees K

- Unlike bagging and random forests, boosting can overfit if *K* is too large, although this overfitting tends to occur slowly if at all.
- We use cross-validation to select *K.*

The shrinkage parameter λ

- The shrinkage parameter λ should be a small positive number.
- This controls the rate at which boosting learns.
- Typical values are 0.01 or 0.001, and the right choice can depend on the problem.
- Very small values can require using a very large value of *K* in order to achieve good performance.

The number *d* splits in each tree

- The value of *d* controls the complexity of the boosted ensemble.
- Often *d* = 1 works well, in which case each tree is a stump, consisting of a single split.
- More generally *d* is the interaction depth, and controls the interaction order of the boosted model, since *d* splits can involve at most *d* variables.

Small values of d = $\underline{1, 2, 5, 10}$

Boosting

d

d={1, 3, 5, 7}

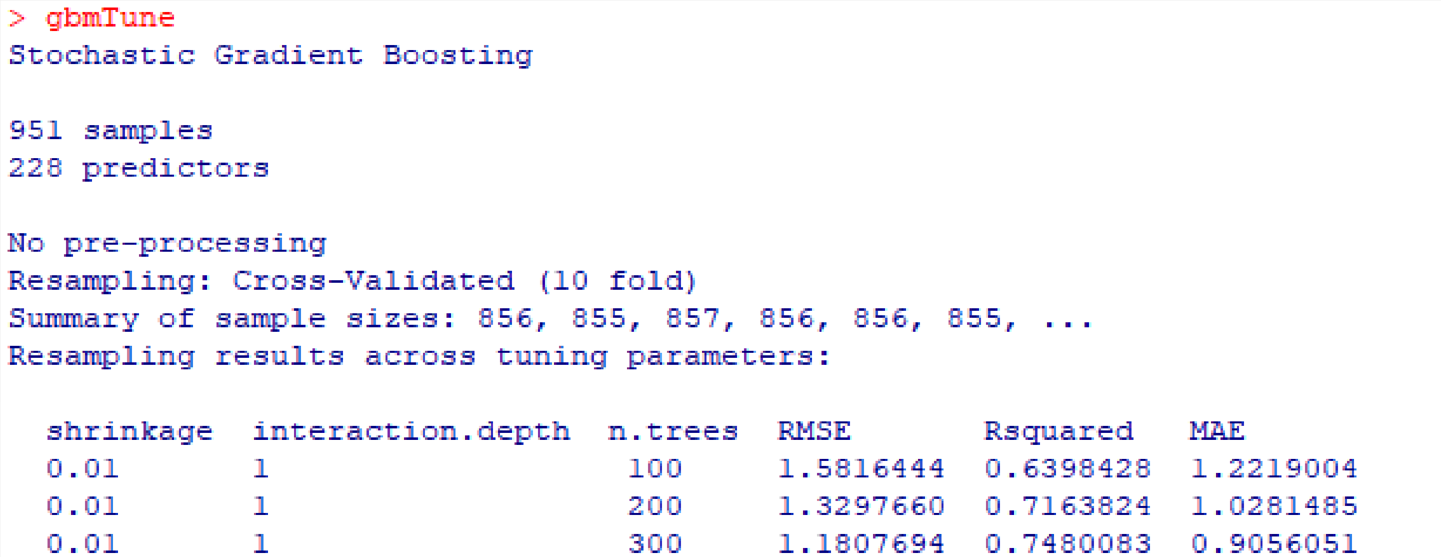
```
gbmGrid = expand.grid( interaction.depth = seq( 1, 7, by=2 ),
                       n.trees = seq( 100, 1000, by=100 ),
                       shrinkage = c(0.01, 0.1),
                       n.minobsinnode = 10 )
set.seed(100) #takes 463.26 seconds to run in my computer
gbmTune <- train(x = solTrainXtrans, y = solTrainY,
                 method = "gbm",
                 tuneGrid = gbmGrid,
                 trControl = ctrl,
                 verbose = FALSE)
gbmTune

plot(gbmTune, auto.key = list(columns = 4, lines = TRUE))

gbmImp <- varImp(gbmTune, scale = FALSE)
gbmImp
```

$K = \{100, 200, \ldots 1000\}$

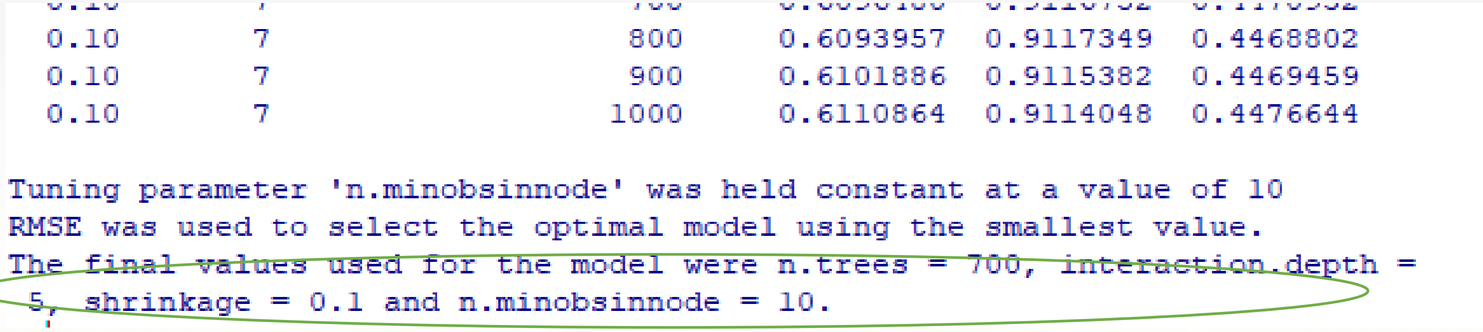
λ

$\lambda = \{0.01, 0.1\}$

Boosting



```
> gbmTune
Stochastic Gradient Boosting

951 samples
228 predictors

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 856, 855, 857, 856, 856, 855, ...
Resampling results across tuning parameters:

  shrinkage  interaction.depth  n.trees  RMSE       Rsquared   MAE
  0.01       1                   100     1.5816444  0.6398428  1.2219004
  0.01       1                   200     1.3297660  0.7163824  1.0281485
  0.01       1                   300     1.1807694  0.7480083  0.9056051
```

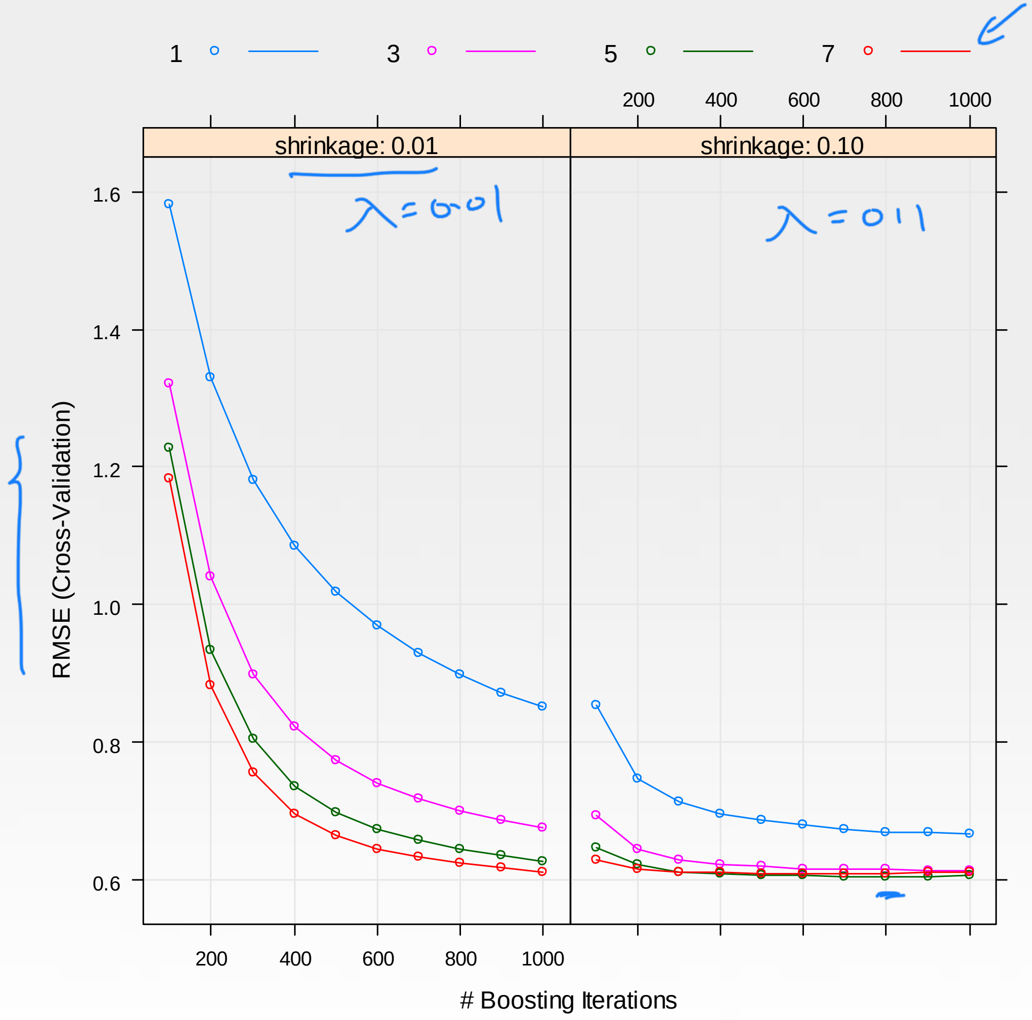


```
  0.10       7                   800     0.6093957  0.9117349  0.4468802
  0.10       7                   900     0.6101886  0.9115382  0.4469459
  0.10       7                  1000     0.6110864  0.9114048  0.4476644

Tuning parameter 'n.minobsinnode' was held constant at a value of 10
RMSE was used to select the optimal model using the smallest value.
The final values used for the model were n.trees = 700, interaction.depth =
 5, shrinkage = 0.1 and n.minobsinnode = 10.
```

$k = 700$, $d = 5$, $\lambda = 0.1$

Cross-validated RMSE profiles for the boosted tree model

d = 1, 3, 5. 7



Random forests

bagging

- Random forests provide an improvement over bagged trees by way of a small tweak that *decorrelates* the trees. This reduces the variance when we average the trees. $Var(T_1+T_2) = Var(T_1)+Var(T_2)+2cov(T_1,T_2)$
- As in bagging, we build a number of decision trees on bootstrapped >0 training samples. that are correlated.
- But when building these decision trees, each time a split in a tree is considered, a random selection of *m* predictors is chosen as split candidates from the full set of *p* predictors. The split is allowed to use only one of those *m* predictors.
- *It usually takes very long time to run and get the output.*

Random forests

- A fresh selection of *m* predictors is taken at each split, and typically we choose

$$m \approx \sqrt{p}$$

p = 110 predictor

m = 10.

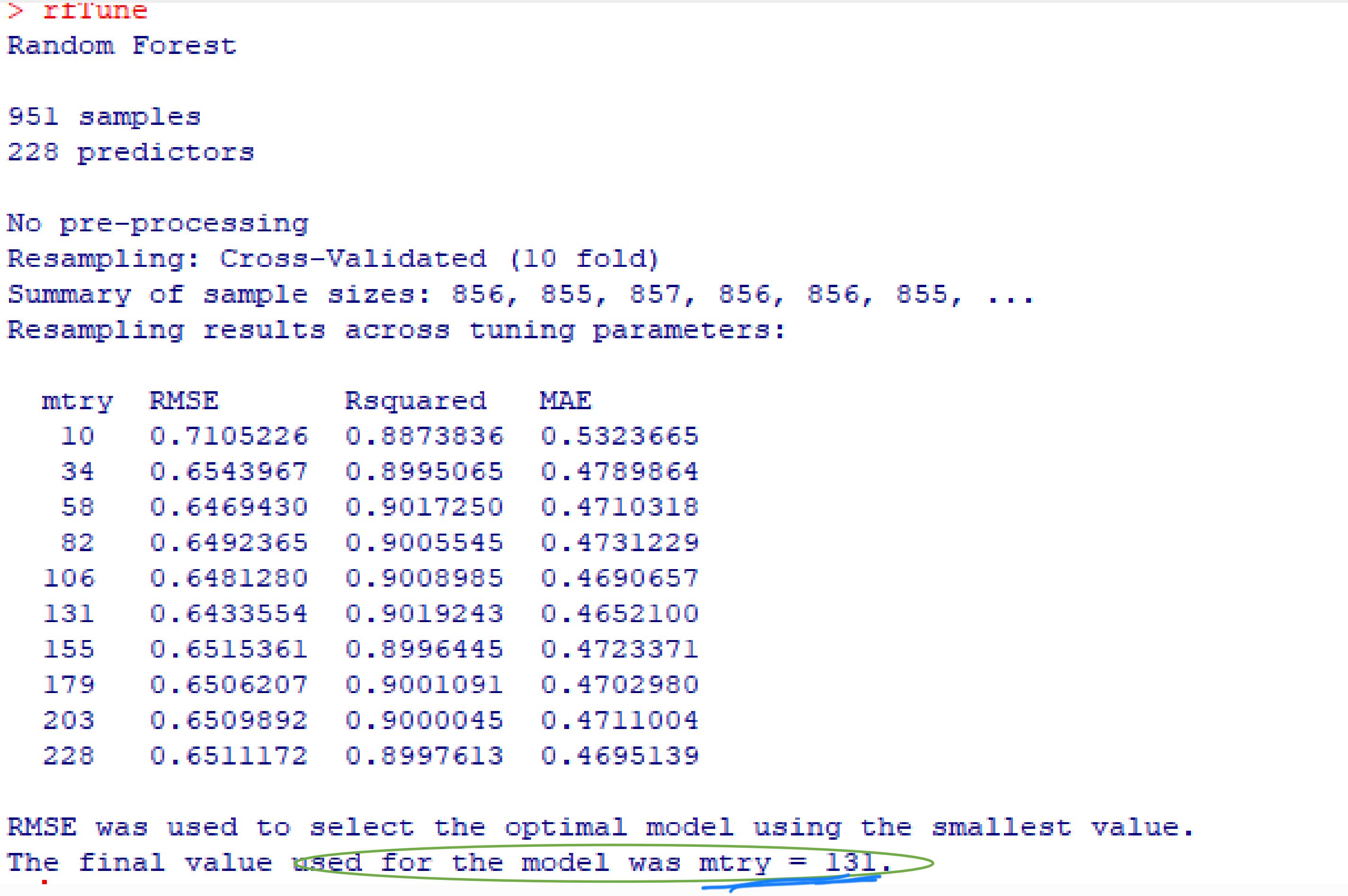
- Using a small value of *m* in building a random forest will typically be helpful when we have a large number of correlated predictors.

Random forests (takes time to run)

```
mtryGrid <- data.frame(mtry = floor(seq(10, ncol(solTrainXtrans), length = 10)))

### Tune the model using cross-validation
set.seed(100)
rfTune <- train(x = solTrainXtrans, y = solTrainY,
                method = "rf",
                tuneGrid = mtryGrid,
                ntree = 200,
                importance = TRUE,
                trControl = ctrl)
rfTune

plot(rfTune)
```

random forest

Random forests



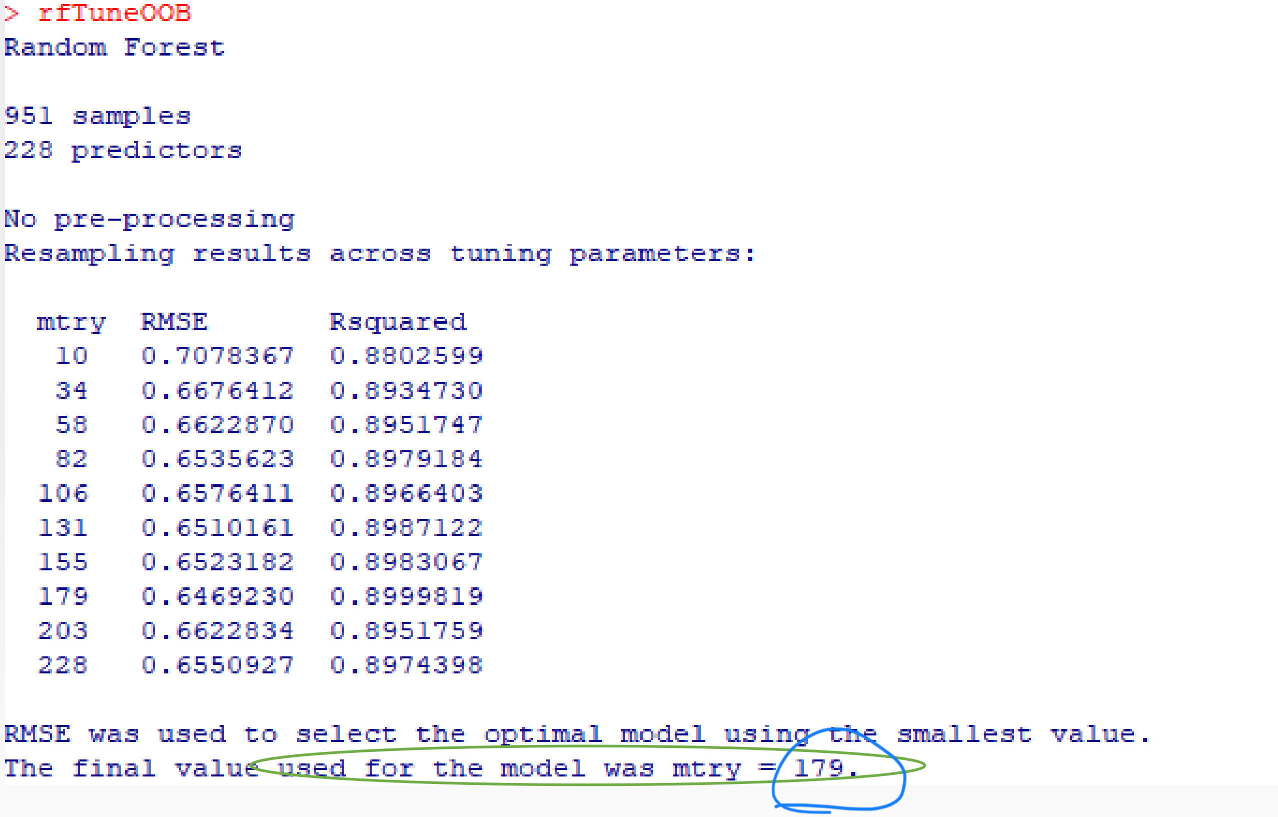
```
> rfTune
Random Forest

951 samples
228 predictors

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 856, 855, 857, 856, 856, 855, ...
Resampling results across tuning parameters:

  mtry  RMSE       Rsquared   MAE
   10   0.7105226  0.8873836  0.5323665
   34   0.6543967  0.8995065  0.4789864
   58   0.6469430  0.9017250  0.4710318
   82   0.6492365  0.9005545  0.4731229
  106   0.6481280  0.9008985  0.4690657
  131   0.6433554  0.9019243  0.4652100
  155   0.6515361  0.8996445  0.4723371
  179   0.6506207  0.9001091  0.4702980
  203   0.6509892  0.9000045  0.4711004
  228   0.6511172  0.8997613  0.4695139

RMSE was used to select the optimal model using the smallest value.
The final value used for the model was mtry = 131.
```

Random forests with OOB

```
### Tune the model using the OOB estimates
ctrlOOB <- trainControl(method = "oob")
set.seed(100)
rfTuneOOB <- train(x = solTrainXtrans, y = solTrainY,
            method = "rf",
            tuneGrid = mtryGrid,
            ntree = 200,
            importance = TRUE,
            trControl = ctrlOOB)
rfTuneOOB

rfImp <- varImp(rfTuneOOB, scale = FALSE)
rfImp
plot(rfImp, 20)
```

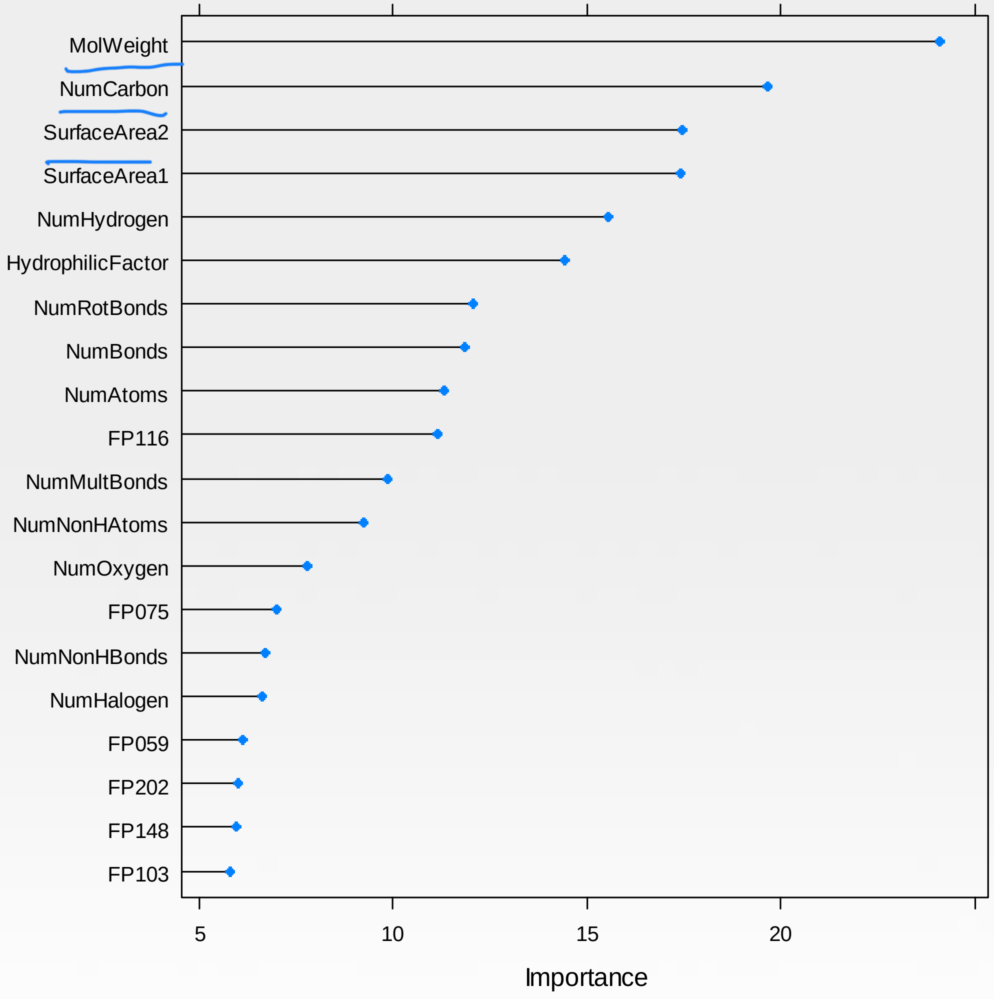
← Variable importance

Random forests with OOB



```
> rfTuneOOB
Random Forest

951 samples
228 predictors

No pre-processing
Resampling results across tuning parameters:

  mtry  RMSE       Rsquared 
   10   0.7078367  0.8802599
   34   0.6676412  0.8934730
   58   0.6622870  0.8951747
   82   0.6535623  0.8979184
  106   0.6576411  0.8966403
  131   0.6510161  0.8987122
  155   0.6523182  0.8983067
  179   0.6469230  0.8999819
  203   0.6622834  0.8951759
  228   0.6550927  0.8974398

RMSE was used to select the optimal model using the smallest value.
The final value used for the model was mtry = 179.
```

Variable importance



Summary

- Decision trees are simple and interpretable models for regression and classification, whereas they are often not competitive with other methods in terms of prediction accuracy
- Bagging, random forests and boosting are good methods for improving the prediction accuracy of trees. They work by growing many trees on the training data and then combining the predictions of the resulting ensemble of trees.
- The random forests and boosting—are among the state-of-the-art methods for supervised learning. However, their results can be difficult to interpret.

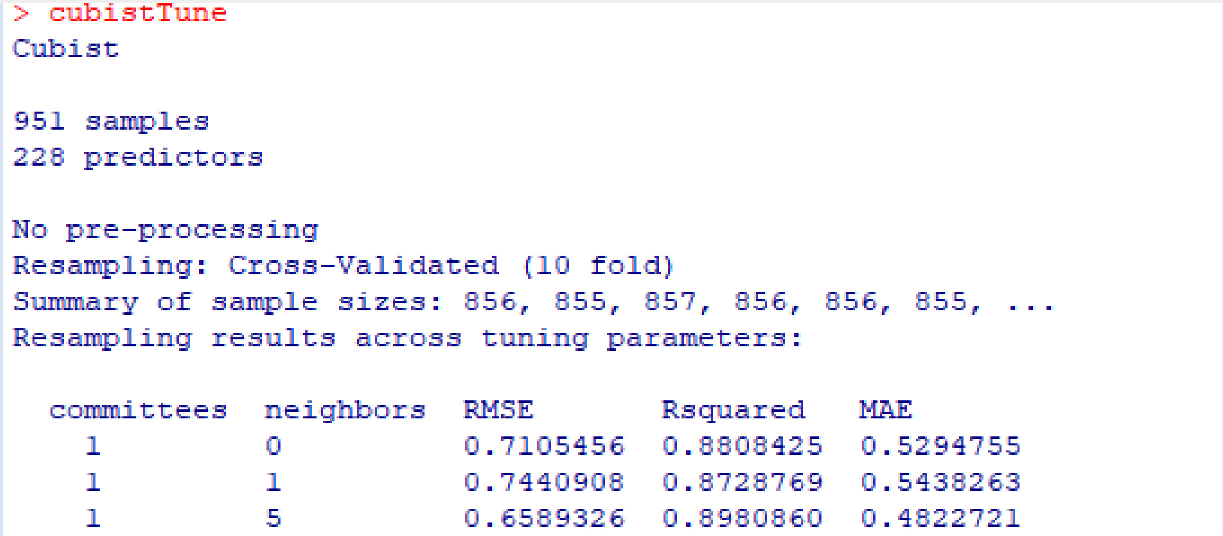
Cubist

- Cubist is a *rule-based* model that is an amalgamation of several methodologies.
- Some specific differences between Cubist and the previously described approaches for model trees and their rule-based variants are
  - The specific techniques used for linear model smoothing, creating rules, and pruning are different
  - An optional boosting—like procedure called *committees*
  - The predictions generated by the model rules can be adjusted using nearby points from the training set data
- To tune this model, different numbers of *committees* and *neighbors* were assessed.

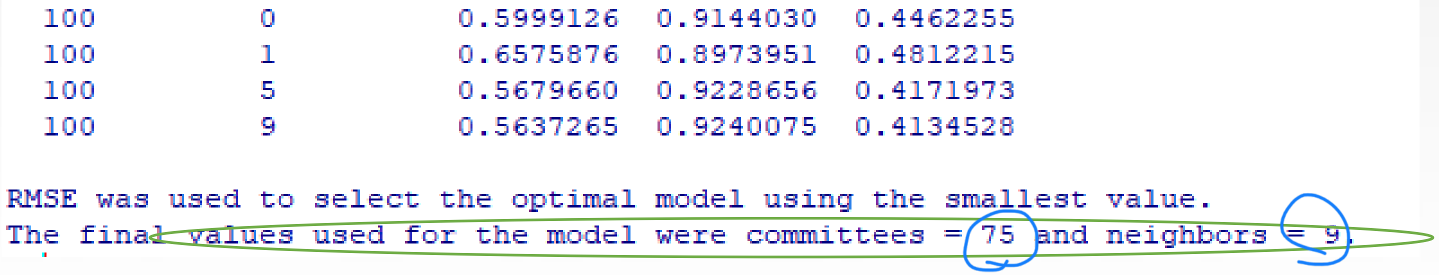
Cubist

```
library(Cubist)

cbGrid <- expand.grid(committees = c(1:10, 20, 50, 75, 100),
                      neighbors = c(0, 1, 5, 9))

set.seed(100) #takes  307.76 seconds to run in my computer
cubistTune <- train(solTrainXtrans, solTrainY,
                    "cubist",
                    tuneGrid = cbGrid,
                    trControl = ctrl)
cubistTune

plot(cubistTune, auto.key = list(columns = 4, lines = TRUE))

cbImp <- varImp(cubistTune, scale = FALSE)
cbImp

plot(cbImp, 20)
```

Cubist

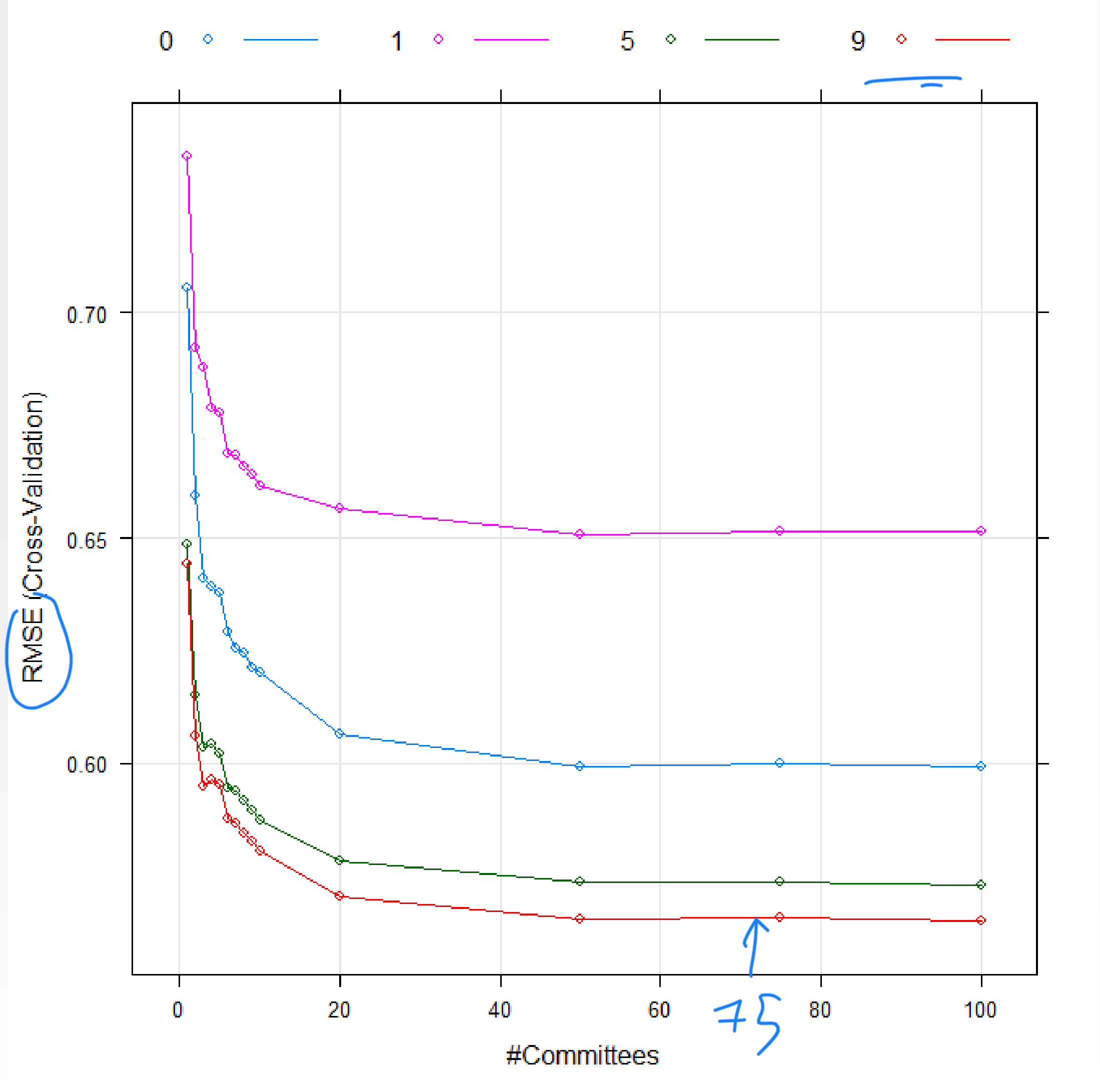


```
> cubistTune
Cubist

951 samples
228 predictors

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 856, 855, 857, 856, 856, 855, ...
Resampling results across tuning parameters:

  committees  neighbors  RMSE       Rsquared   MAE
    1         0          0.7105456  0.8808425  0.5294755
    1         1          0.7440908  0.8728769  0.5438263
    1         5          0.6589326  0.8980860  0.4822721
```

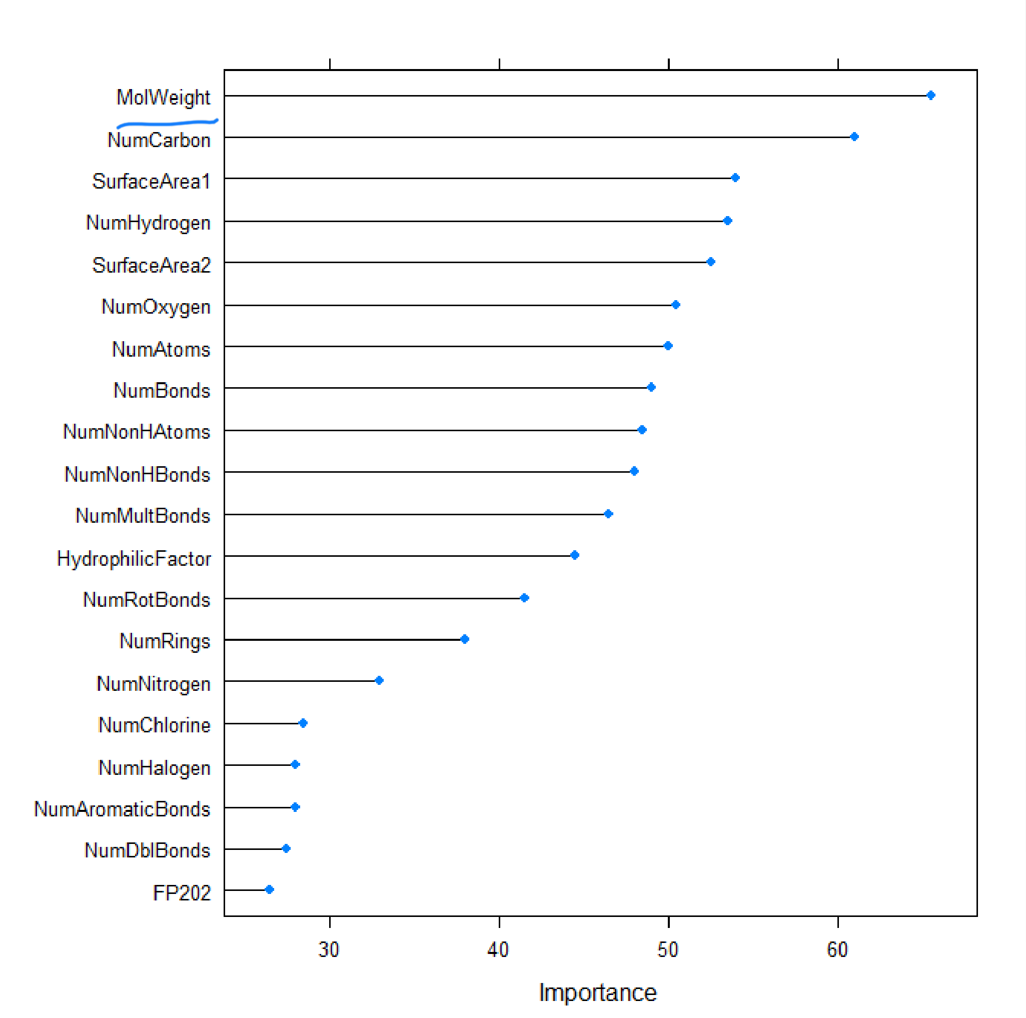


```
  100         0          0.5999126  0.9144030  0.4462255
  100         1          0.6575876  0.8973951  0.4812215
  100         5          0.5679660  0.9228656  0.4171973
  100         9          0.5637265  0.9240075  0.4134528

RMSE was used to select the optimal model using the smallest value.
The final values used for the model were committees = 75 and neighbors = 9.
```

Cubist


0
1
5
9
RMSE (Cross-Validation)
0.70
0.65
0.60
0
20
40
60
80
100
#Committees
75

Variable importance



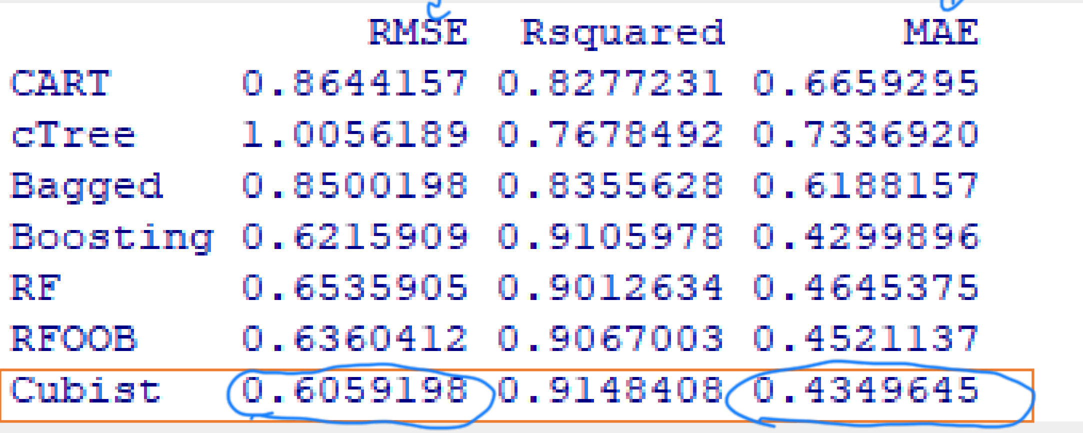
Performance comparison of nonlinear models

```
### Performance of tree-based models
set.seed(100)
Cart.pred <- predict(cartTune, solTestXtrans)
cTree.pred <- predict(ctreeTune, solTestXtrans)
Bagged.pred <- predict(treebagTune, solTestXtrans)
Boosting.pred <- predict(gbmTune, solTestXtrans)
RF.pred   <- predict(rfTune, solTestXtrans)
RFOOB.pred <- predict(rfTuneOOB, solTestXtrans)
Cubist.pred <- predict(cubistTune, solTestXtrans)


data.frame(rbind(CART=postResample(pred=Cart.pred,obs = solTestY),
                 cTree=postResample(pred=cTree.pred ,obs = solTestY),
                 Bagged=postResample(pred=Bagged.pred,obs = solTestY),
                 Boosting=postResample(pred=Boosting.pred,obs = solTestY),
                 RF=postResample(pred=RF.pred,obs = solTestY),
                 RFOOB=postResample(pred=RFOOB.pred,obs = solTestY),
                 Cubist=postResample(pred=Cubist.pred,obs = solTestY) ))
```

} single tree-based model

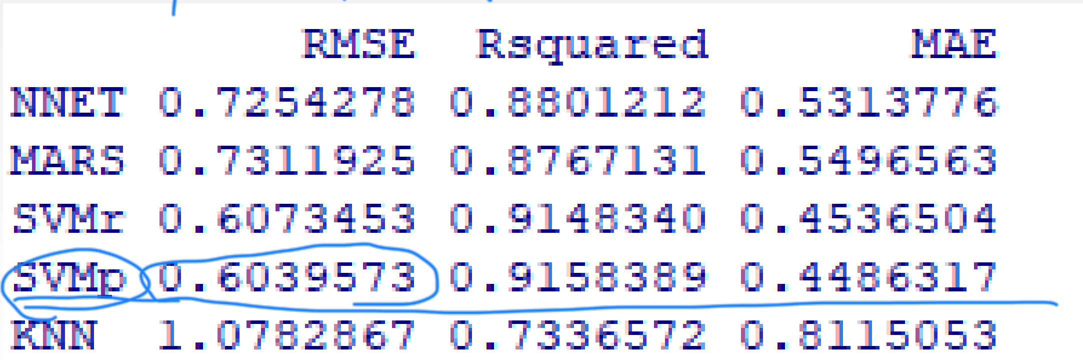
} for improving prediction accuracy.

Performance comparison of tree-based models



```
                RMSE  Rsquared       MAE
CART      0.8644157 0.8277231 0.6659295
cTree     1.0056189 0.7678492 0.7336920
Bagged    0.8500198 0.8355628 0.6188157
Boosting  0.6215909 0.9105978 0.4299896
RF        0.6535905 0.9012634 0.4645375
RFOOB     0.6360412 0.9067003 0.4521137
Cubist    0.6059198 0.9148408 0.4349645
```

Non-linear



```
          RMSE  Rsquared       MAE
NNET 0.7254278 0.8801212 0.5313776
MARS 0.7311925 0.8767131 0.5496563
SVMr 0.6073453 0.9148340 0.4536504
SVMp 0.6039573 0.9158389 0.4486317
KNN  1.0782867 0.7336572 0.8115053
```

In terms of RMSE, SVMp ✓

MAE; Cubist ✓

R^2, SVMp ✓

Introduction to R

Exercise 5