Variable Selection Using The `caret` Package

Max Kuhn
max.kuhn@pfizer.com

June 30, 2009

1 Models with Built–In Feature Selection

Many models that can be accessed using `caret`'s `train` function produce prediction equations that do not necessarily use all the predictors. These models are thought to have built–in feature selection and include `rpart`, `gbm`, `ada`, `glmboost`, `gamboost`, `blackboost`, `ctree`, `sparseLDA`, `sddaLDA`, `sddaQDA` `glmnet`, `lasso`, `lars`, `spls`, `earth`, `fda`, `bagEarth`, `bagFDA` , `pam` and others. Many of the functions have an ancillary method called `predictors` that returns a vector indicating which predictors were used in the final model.

In many cases, using these models with built–in feature selection will be more efficient than algorithms where the search routine for the right predictors is external to the model (see Section 2). Built–in feature selection typically couples the predictor search algorithm with the parameter estimation and are usually optimize with a single objective function (e.g. error rates or likelihood).

2 Feature Selection Using Search Algorithms

2.1 Searching the Feature Space

Many feature selection routines used a "wrapper" approach to find appropriate variables such that an algorithm that searches the feature space repeatedly fits the model with different predictor SETS. The best predictor set is determined by some measure of performance (i.e. R^2, classification accuracy, ECT). Examples of search functions are genetic algorithms, simulated annealing and forward/backward/stepwise selection methods. In theory, each of these search routines could converge to an optimal set of predictors.

An example of one search routine is backwards selection (a.k.a. recursive feature elimination).

2.1.1 Backwards Selection

First, the algorithm fits the model to all predictors. Each predictor is ranked ITS importance to the model. Let S be a sequence of ordered numbers which are candidate values for the number of predictors to retain ($S_1 > S_2, \ldots$). At each iteration of feature selection, the S_i top raked predictors are retained, the model is refit and performance is assessed. The value of S_i with the best performance is determined and the top S_i predictors are used to fit the final model. Algorithm 1 has a more complete definition.

The algorithm has an optional step (line 1.8) where the predictor rankings are recomputed on the model on the reduced feature set. Svetnik *el al* (2004) showed that, for random forest models, there was a decrease in performance when the rankings were re–computed at every step. However, in other cases when the initial rankings are not good (e.g. linear models with highly collinear predictors), re–calculation can slightly improve performance.

Algorithm 1: Recursive feature elimination

1.1 Tune/train the model on the training set using all predictors
1.2 Calculate model performance
1.3 Calculate variable importance or rankings
1.4 **for** *Each subset size* S_i, $i = 1 \ldots S$ **do**
1.5 Keep the S_i most important variables
1.6 Tune/train the model on the training set using S_i predictors
1.7 Calculate model performance
1.8 [Optional] Recalculate the rankings for each predictor
1.9 **end**
1.10 Calculate the performance profile over the S_i
1.11 Determine the appropriate number of predictors
1.12 Determine the final ranks of each predictor
1.13 Fit the final model based on the optimal S_i

One potential issue over–fitting to the predictor set such that the wrapper procedure could focus on nuances of the training data that are not found in future samples (i.e. over–fitting to predictors and samples).

For example, suppose a very large number of uninformative predictors were collected and one such predictor randomly correlated with the outcome. The RFE algorithm would give a good rank to this variable and the prediction error (on the same data set) would be lowered. It would take a different test/validation to find out that this predictor was uninformative. The was referred to as

"selection bias" by Ambroise and McLachlan (2002).

In the current RFE algorithm, the training data is being used for at least three purposes: predictor selection, model fitting and performance evaluation. Unless the number of samples is large, especially in relation to the number of variables, one static training set may not be able to fulfill these needs.

2.2 Resampling and External Validation

Since feature selection is part of the model building process, resampling methods (e.g. cross–validation, the bootstrap) should factor in the variability caused by feature selection when calculating performance. For example, the RFE procedure in Algorithm 1 can estimate the model performance on line 1.6, which during the selection process. Ambroise and McLachlan (2002) and Svetnik *el al* (2004) showed that improper use of resampling to measure performance will result in models that perform poorly on new samples.

To get performance estimates that incorporate the variation due to feature selection, it is suggested that the steps in Algorithm 1 be encapsulated inside an outer layer of resampling (e.g. 10–fold cross–validation). Algorithm 2 shows a version of the algorithm that uses resampling.

While this will provide better estimates of performance, it is more computationally burdensome. For users with access to machines with multiple processors, the first `For` loop in Algorithm 2 (line 2.1) can be easily parallelized. Another complication to using resampling is that multiple lists of the "best" predictors are generated at each iteration. At first this may seem like a disadvantage, but it does provide a more probabilistic assessment of predictor importance than a ranking based on a single fixed data set. At the end of the algorithm, a consensus ranking can be used to determine the best predictors to retain.

Algorithm 2: Recursive feature elimination incorporating resampling

for *Each Resampling Iteration* **do**
    Partition data into training and test/hold–back set via resampling
    Tune/train the model on the training set using all predictors
    Predict the held–back samples
    Calculate variable importance or rankings
    for *Each subset size* S_i, $i = 1 \ldots S$ **do**
        Keep the S_i most important variables
        Tune/train the model on the training set using S_i predictors
        Predict the held–back samples
        [Optional] Recalculate the rankings for each predictor
    end
end
Calculate the performance profile over the S_i using the held–back samples
Determine the appropriate number of predictors
Estimate the fial list of predictors to keep in the final model
Fit the final model based on the optimal S_i using the original training set

3 Recursive Feature Elimination via caret

In caret, Algorithm 1 is implemented by the function rfeIter. The resampling-based Algorithm 2 is in the rfe function. Given the potential selection bias issues, this document focuses on rfe. There are several arguments:

- x, a matrix or data frame of predictor variables
- y, a vector (numeric or factor) of outcomes
- sizes, a integer vector for the specific subset sizes that should be tested (which need not to include ncol(x))
- rfeControl, a list of options that can be used to specify the model and the methods for prediction, ranking etc.

For a specific model, a set of functions must be specified in rfeControl$functions. Section 3.2 below has descriptions of these sub–functions. There are a number of pre–defined sets of functions for several models, including: linear regression (in the object lmFuncs), random forests (rfFuncs), naive Bayes (nbFuncs), bagged trees (treebagFuncs) and functions that can be used with caret's train function (caretFuncs). The latter is useful if the model has tuning parameters that must be determined at each iteration.

3.1 An Example

To test the algorithm, the "Friedman 1" benchmark (Friedman, 1991) was used. There are three informative variables generated by the equation

$$y = 10 \sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + N(0, \sigma^2)$$

In the simulation used here:

```
> n <- 100
> p <- 40
> sigma <- 1
> set.seed(1)
> sim <- mlbench.friedman1(n, sd = sigma)
> x <- cbind(sim$x, matrix(rnorm(n * p), nrow = n))
> y <- sim$y
> colnames(x) <- paste("var", 1:ncol(x), sep = "")
```

Of the 50 predictors, there are 45 pure noise variables: 5 are uniform on [0, 1] and 40 are random univariate standard normals. The predictors are centered and scaled:

```
> normalization <- preProcess(x)
> x <- predict(normalization, x)
> x <- as.data.frame(x)
> subsets <- c(1:5, 10, 15, 20, 25)
```

The simulation will fit models with subset sizes of 25, 20, 15, 10, 5, 4, 3, 2, 1.

As previously mentioned, to fit linear models, the `lmFuncs` set of functions can be used. To do this, a control object is created with the `rfeControl` function. We also specify that 10–fold cross–validation should be used in line 2.1 of Algorithm 2. The number of folds can be changed via the `number` argument to `rfeControl` (defaults to 10). The `verbose` option prevents copious amounts of output from being produced and the `returnResamp` argument specifies that the 10 performance estimates should be saved only for the optimal subset size.

```
> set.seed(10)
> ctrl <- rfeControl(functions = lmFuncs, method = "cv", verbose = FALSE,
+     returnResamp = "final")
> lmProfile <- rfe(x, y, sizes = subsets, rfeControl = ctrl)
> lmProfile

Recursive feature selection

Outer resamping method was 10 iterations of cross-validation.

Resampling perfromance over subset size:

 Variables  RMSE Rsquared RMSESD RsquaredSD Selected
         1 3.473   0.5285 0.4706     0.1219
         2 3.134   0.6161 0.5937     0.1757
         3 2.954   0.6770 0.9152     0.2242        *
         4 3.055   0.6520 0.9889     0.2359
         5 3.229   0.6188 0.8714     0.1966
        10 3.493   0.5549 0.9811     0.2098
        15 3.754   0.5010 1.1806     0.2243
        20 3.893   0.4725 1.0039     0.2026
        25 4.306   0.4009 0.9284     0.1870
        50 4.306   0.4009 0.9284     0.1870

The top 3 variables (out of 3):
   var4, var5, var2
```

The output shows that the best subset size was estimated to be 3 predictors. This set includes informative variables but did not include them all. The `predictors` function can be used to get a

text string of variable names that were picked in the final model. The `lmProfile` is a list of class `"rfe"` that contains an object `fit` that is the final linear model with the remaining terms. The model can be used to get predictions for future or test samples.

```
> predictors(lmProfile)

[1] "var4" "var5" "var2"

> lmProfile$fit

Call:
lm(formula = y ~ ., data = tmp)

Coefficients:
(Intercept)         var4         var5         var2
     14.613        2.625        1.967        1.648

> lmProfile$resample

          RMSE  Rsquared Variables
 [1,] 3.640221 0.5471438         3
 [2,] 2.270739 0.7953448         3
 [3,] 2.425445 0.6704767         3
 [4,] 3.195171 0.8108668         3
 [5,] 4.994849 0.2072672         3
 [6,] 2.184304 0.7963478         3
 [7,] 2.602312 0.8716050         3
 [8,] 1.968237 0.9170471         3
 [9,] 2.714145 0.7397995         3
[10,] 3.548103 0.4141139         3
```

There are also several plot methods to visualize the results. `plot(lmProfile)` produces the performance profile across different subset sizes, as shown in Figure 1. Also the resampling results are stored in the sub–object `lmProfile$resample` and can be used with several lattice functions. Univariate lattice functions (`densityplot`, `histogram`) can be used to plot the resampling distribution while bivariate functions (`xyplot`, `stripplot`) can be used to plot the distributions for different subset sizes. In the latter case, the option `returnResamp = "all"` in `rfeControl` can be used to save all the resampling results. See Figure 4 for two examples.

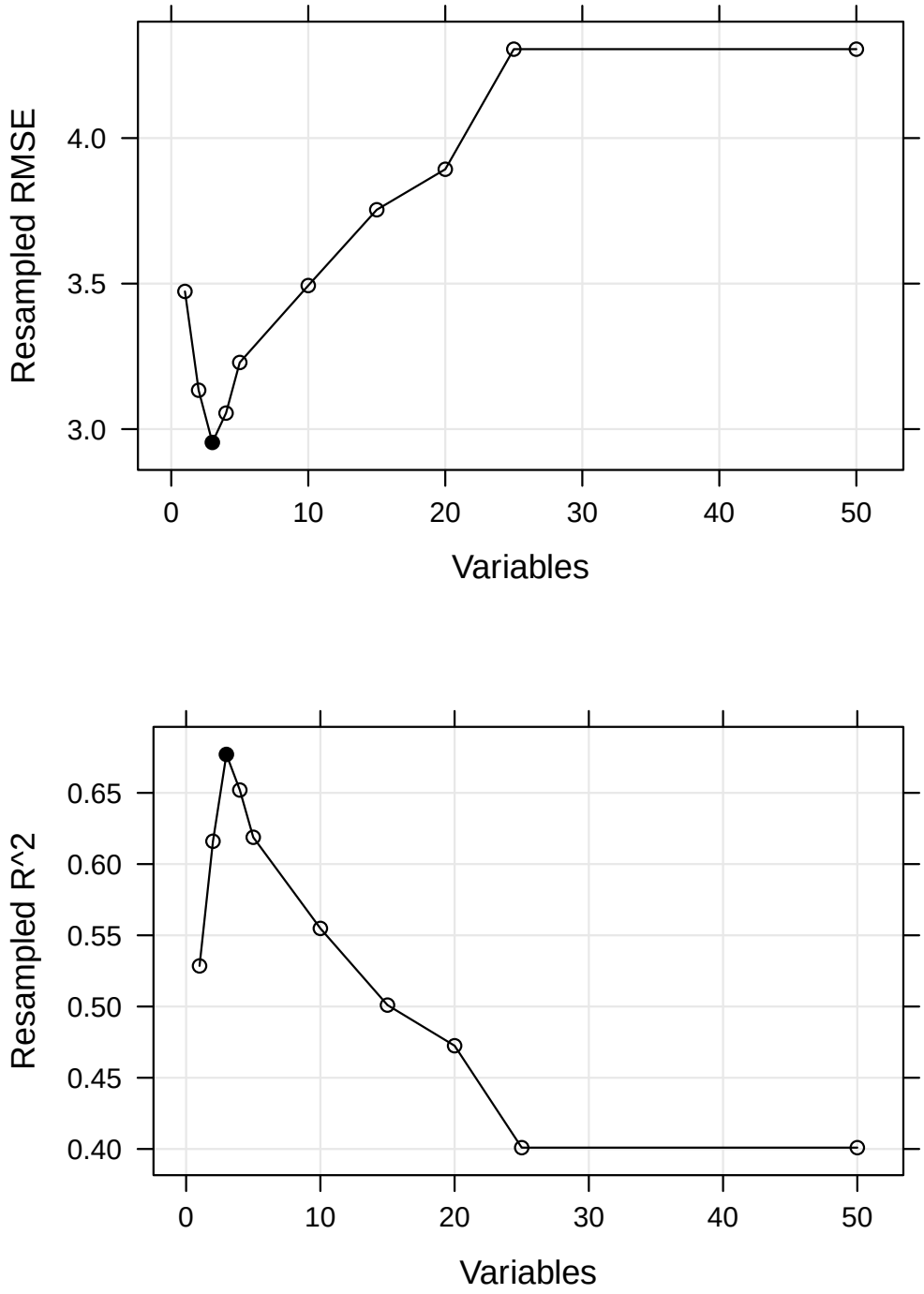

Figure 1: Performance profiles for recursive feature elimination using linear models. These images were generated by `plot(lmProfile)` and `plot(lmProfile, metric = "Rsquared")`.

3.2 Helper Functions

To use feature elimination for an arbitrary model, a set of functions must be passed to `rfe` for each of the steps in Algorithm 2. This section defines those functions and uses the existing random forest functions as an illustrative example.

3.2.1 The `fit` Function

This function builds the model based on the current data set (lines line 2.3, 2.8 and 2.16). The arguments for the function must be:

- `x`: the current training set of predictor data with the appropriate subset of variables
- `y`: the current outcome data (either a numeric or factor vector)
- `first`: a single logical value for whether the current predictor set has all possible variables (e.g. line 2.3)
- `last`: similar to `first`, but `TRUE` when the last model is fit with the final subset size and predictors. (line 2.16)
- ...: optional arguments to pass to the fit function in the call to `rfe`

The function should return a model object that can be used to generate predictions. For random forest, the fit function is simple:

```
> rfFuncs$fit
```

```
function (x, y, first, last, ...)
{
    library(randomForest)
    randomForest(x, y, importance = first, ...)
}
<environment: namespace:caret>
```

For feature selection without re–ranking at each iteration, the random forest variable importances only need to be computed on the first iterations when all of the predictors are in the model. This can be accomplished using `importance = first`.

3.2.2 The pred Function

This function returns a vector of predictions (numeric or factors) from the current model (lines 2.4 and 2.9). The input arguments must be

- object: the model generated by the fit function
- x: the current set of predictor set for the held–back samples

For random forests, the function is a simple wrapper for the predict function:

```
> rfFuncs$pred
```

```
function (object, x)
{
    predict(object, x)
}
<environment: namespace:caret>
```

For classification, it is probably a good idea to ensure that the resulting factor variables of predictions has the same levels as the input data.

3.2.3 The rank Function

This function is used to return the predictors in the order of the most important to the least important (lines 2.5 and 2.10). Inputs are:

- object: the model generated by the fit function
- x: the current set of predictor set for the training samples
- y: the current training outcomes

The function should return a data frame with a column called vars that has the current variable names. The first row should be the most important predictor etc. Other columns can be included in the output and will be returned in the final rfe object.

For random forests, the function below uses caret's varImp function to extract the random forest importances and orders them. For classification, randomForest will produce a column of importances for each class. In this case, the default ranking function orders the predictors by the averages importance across the classes.

```
> rfFuncs$rank
```

```
function (object, x, y)
{
    vimp <- varImp(object)
    if (is.factor(y)) {
        if (all(levels(y) %in% colnames(vimp))) {
            avImp <- apply(vimp[, levels(y), drop = TRUE], 1,
                mean)
            vimp$Overall <- avImp
        }
    }
    vimp <- vimp[order(vimp$Overall, decreasing = TRUE), , drop = FALSE]
    vimp$var <- rownames(vimp)
    vimp
}
<environment: namespace:caret>
```

3.2.4 The `selectSize` Function

This function determines the optimal number of predictors based on the resampling output (line 2.14). Inputs for the function are:

- `x`: a matrix with columns for the performance metrics and the number of variables, called `Variables`
- `metric`: a character string of the performance measure to optimize (e.g. RMSE, Accuracy)
- `maximize`: a single logical for whether the metric should be maximized

This function should return an integer corresponding to the optimal subset size.

`caret` comes with two examples functions for this purpose: `pickSizeBest` and `pickSizeTolerance`. The former simply selects the subset size that has the best value. The latter takes into account the whole profile and tries to pick a subset size that is small without sacrificing too much performance. For example, suppose we have computed the RMSE over a series of variables sizes:

```
> example <- data.frame(RMSE = c(3.215, 2.819, 2.414, 2.144, 2.014,
+     1.997, 2.025, 1.987, 1.971, 2.055, 1.935, 1.999, 2.047, 2.002,
+     1.895, 2.018), Variables = 1:16)
> example
```

```
    RMSE Variables
1  3.215         1
2  2.819         2
3  2.414         3
4  2.144         4
5  2.014         5
6  1.997         6
7  2.025         7
8  1.987         8
9  1.971         9
10 2.055        10
11 1.935        11
12 1.999        12
13 2.047        13
14 2.002        14
15 1.895        15
16 2.018        16
```

These are depicted in Figure 2. The solid circle identifies the subset size with the absolute smallest RMSE. However, there are many smaller subsets that produce approximately the same performance but with fewer predictors. In this case, we might be able to accept a slightly larger error for less predictors.

The `pickSizeTolerance` determines the absolute best value then the percent difference of the other points to this value. In the case of RMSE, this would be

$$RMSE_{tol} = 100 \times \frac{RMSE - RMSE_{opt}}{RMSE_{opt}}$$

where $RMSE_{opt}$ is the absolute best error rate. These "tolerance" values are plotted in the bottom panel of Figure 2. The solid triangle is the smallest subset size that is within 10% of the optimal value.

This approach can produce good results for many of the tree based models, such as random forest, where there is a plateau of good performance for larger subset sizes. For trees, this is usually because unimportant variables are infrequently used in splits and do not significantly affect performance.

3.2.5 The `selectVar` Function

After the optimal subset size is determined, this function will be used to calculate the best rankings for each variable across all the resampling iterations (line 2.15). Inputs for the function are:

- `y`: a list of variables importance for each resampling iteration and each subset size (generated by the user–defined `rank` function). In the example, each each of the cross–validation groups

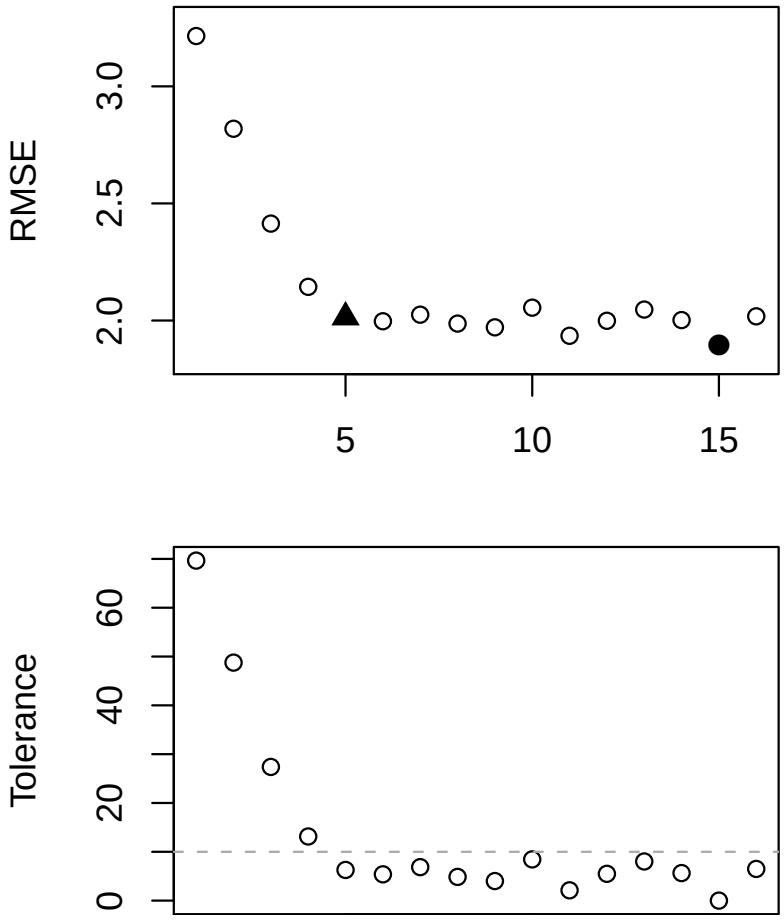

Figure 2: An example of where a smaller subset sizes is not necessarily the best choice. The solid circle in the top panel indicates the subset size with the absolute smallest RMSE. If the percent differences from the smallest RMSE are calculated (lower panel), the user may want to accept a pre–specified drop in performance as long as the drop is within some limit of the optimal.

the output of the rank function is saved for each of the 10 subset sizes (including the original subset). If the rankings are not recomputed at each iteration, the values will be the same within each cross–validation iteration.

- size: the integer returned by the selectSize function

This function should return a character string of predictor names (of length size) in the order of most important to least important

For random forests, only the first importance calculation (line 2.5) is used since these are the rankings on the full set of predictors. These importances are averaged and the top predictors are returned.

```
> rfFuncs$selectVar
```

```
function (y, size)
{
    imp <- lapply(y, function(x) x[[1]])
    imp <- do.call("rbind", imp)
    finalImp <- aggregate(imp$Overall, list(var = imp$var), mean,
        na.rm = TRUE)
    finalImp <- finalImp[order(finalImp$x, decreasing = TRUE),
        ]
    as.character(finalImp$var[1:size])
}
<environment: namespace:caret>
```

Note that if the predictor rankings are recomputed at each iteration (line 2.10) the user will need to write their own selection function to use the other ranks.

3.2.6 The Example

For random forest, we fit the same series of model sizes as the linear model. The option to save all the resampling results across subset sizes was changed for this model and are used to show the lattice plot function capabilities in Figure 4.

```
> set.seed(10)
> ctrl$functions <- rfFuncs
> ctrl$returnResamp <- "all"
> rfProfile <- rfe(x, y, sizes = subsets, rfeControl = ctrl)
> print(rfProfile)
```

```
Recursive feature selection

Outer resamping method was 10 iterations of cross-validation.

Resampling perfromance over subset size:

 Variables  RMSE Rsquared RMSESD RsquaredSD Selected
         1 3.607   0.4670 0.2765    0.16005
         2 3.186   0.6079 0.5151    0.14583
         3 2.779   0.7409 0.3943    0.06699        *
         4 2.885   0.7356 0.2721    0.10742
         5 3.177   0.6806 0.4035    0.10557
        10 3.234   0.6726 0.3771    0.11912
        15 3.350   0.6648 0.3780    0.12272
        20 3.415   0.6294 0.3848    0.14043
        25 3.588   0.6166 0.3591    0.13230
        50 3.565   0.6293 0.3716    0.14347

The top 3 variables (out of 3):
   var4, var5, var2
```

4 Session Information

- R version 2.9.0 (2009-04-17), `x86_64-apple-darwin9.6.0`
- Locale: `en_US.UTF-8/en_US.UTF-8/C/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, graphics, grDevices, grid, methods, splines, stats, tools, utils
- Other packages: caret 4.19, class 7.2-46, e1071 1.5-19, ellipse 0.3-5, gbm 1.6-3, Hmisc 3.5-0, ipred 0.8-6, kernlab 0.9-8, klaR 0.5-8, lattice 0.17-22, MASS 7.2-46, mlbench 1.1-5, nnet 7.2-46, pls 2.1-0, proxy 0.4-1, randomForest 4.5-30, rpart 3.1-43, survival 2.35-4
- Loaded via a namespace (and not attached): cluster 1.11.13

5 References

Ambroise, C. and McLachlan, J. H. (2002) "Selection bias in gene extraction on the basis of microarray gene-expression data," *Proceedings of the National Academy of Science*, 99, 6562–6566

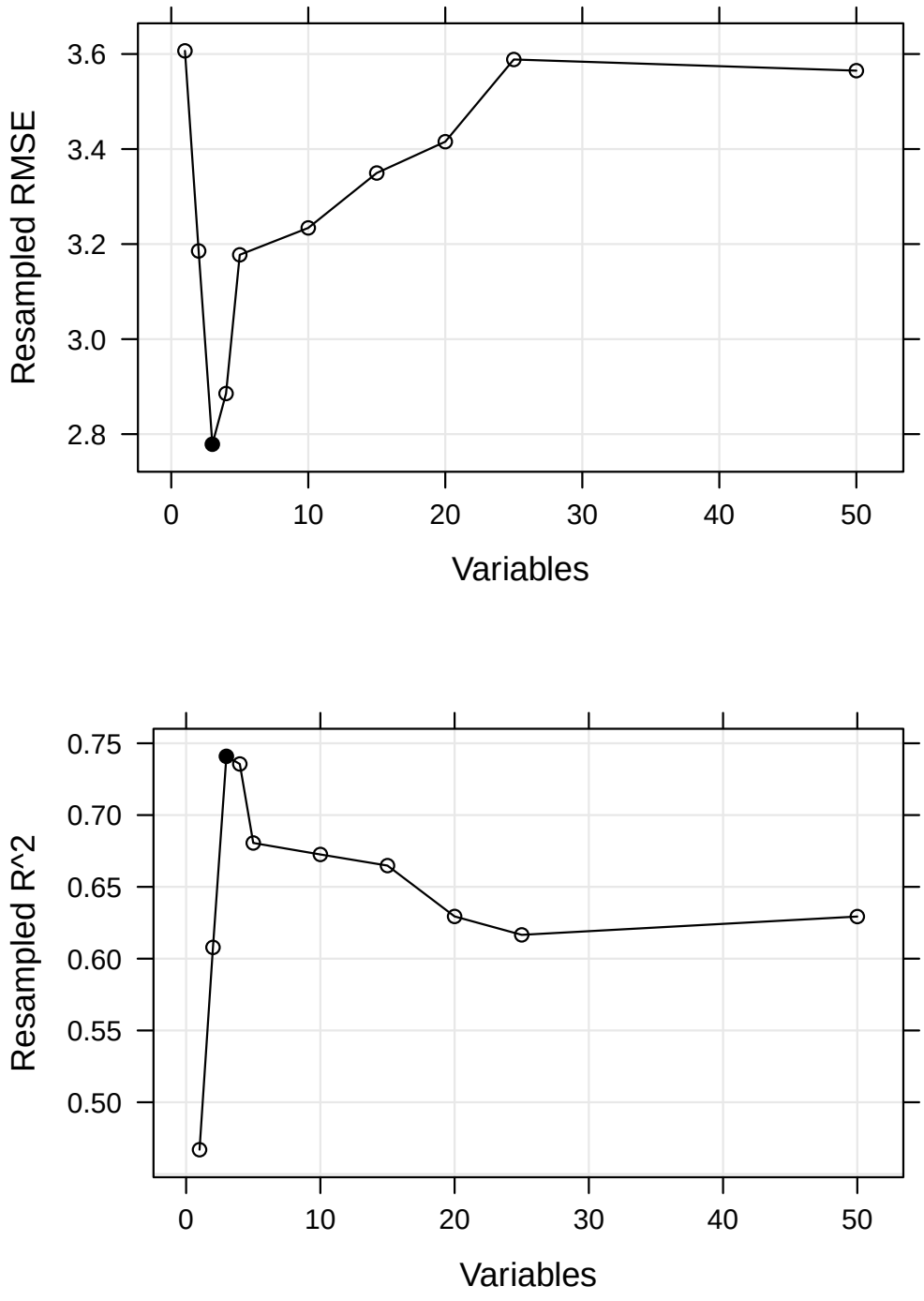

Figure 3: Performance profiles for random forest.

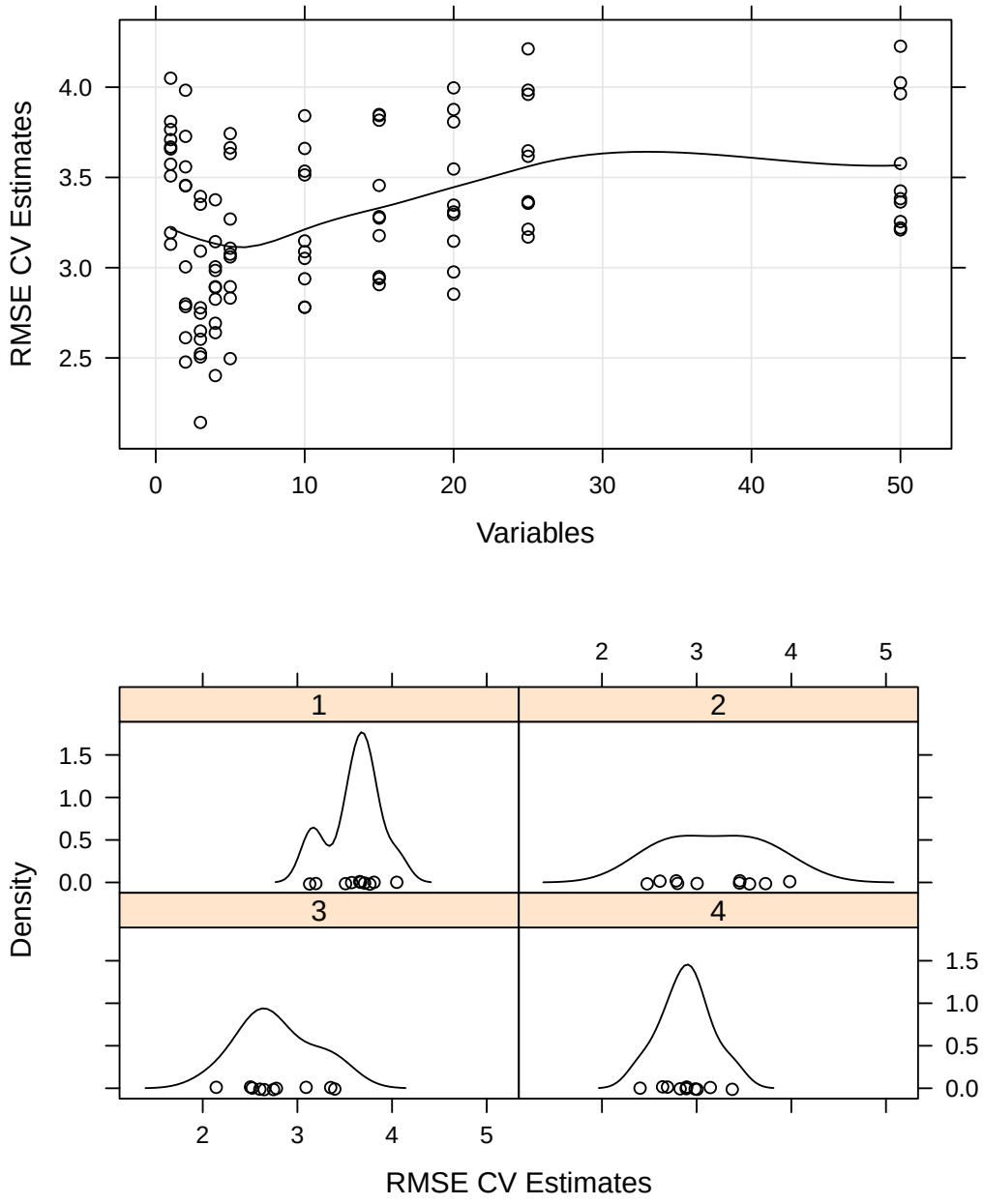

Figure 4: Resampling RMSE estimates for random forests across different subset sizes. These plots were generated using `xyplot(rfProfile)` and `densityplot(rfProfile, subset = Variables < 5)`

Friedman, J. H. (1991) "Multivariate adaptive regression splines (with discussion)," *Annals of Statistics*, 19, 1–141

Svetnik, V., Liaw, A. , Tong, C amd Wang, T. (2004) "Application of Breiman's random forest to modeling structure-activity relationships of pharmaceutical molecules," *Multiple Classier Systems, Fifth International Workshop*, 3077, 334–343