Predictive Modeling with R and the caret Package

useR! 2013

Max Kuhn, Ph.D

Pfizer Global R&D
Groton, CT
max.kuhn@pfizer.com

Outline

- Conventions in R
- Data Splitting and Estimating Performance
- Data Pre-Processing
- Over–Fitting and Resampling
- Training and Tuning Tree Models
- Training and Tuning A Support Vector Machine
- Comparing Models
- Parallel Processing

Predictive Modeling

Predictive modeling (aka machine learning)(aka pattern recognition)(...) aims to generate the most accurate estimates of some quantity or event.

As these models are not generally meant to be *descriptive* and are usually not well–suited for inference.

Good discussions of the contrast between predictive and descriptive/inferential models can be found in Shmueli (2010) and Breiman (2001)

Frank Harrell's `Design` package is very good for modern approaches to interpretable models, such as Cox's proportional hazards model or ordinal logistic regression.

Hastie *et al* (2009) is a good reference for theoretical descriptions of these models while Kuhn and Johnson (2013) focus on the practice of predictive modeling (and uses `R`).

Modeling Conventions in `R`

The Formula Interface

There are two main conventions for specifying models in R: the formula interface and the non–formula (or "matrix") interface.

For the former, the predictors are explicitly listed in an R formula that looks like: `outcome ~ var1 + var2 + ...`.

For example, the formula

```
modelFunction(price ~ numBedrooms + numBaths + acres,
              data = housingData)
```

would predict the closing price of a house using three quantitative characteristics.

The Formula Interface

The shortcut `y ~ .` can be used to indicate that all of the columns in the data set (except `y`) should be used as a predictor.

The formula interface has many conveniences. For example, transformations, such as `log(acres)` can be specified in–line.

It also autmatically converts factor predictors into dummy variables (using a less than full rank encoding). For some R functions (e.g. `klaR:::NaiveBayes`, `rpart:::rpart`, `C50:::C5.0`, ...), predictors are kept as factors.

Unfortunately, R does not efficiently store the information about the formula. Using this interface with data sets that contain a large number of predictors may unnecessarily slow the computations.

The Matrix or Non–Formula Interface

The non–formula interface specifies the predictors for the model using a matrix or data frame (all the predictors in the object are used in the model).

The outcome data are usually passed into the model as a vector object. For example:

```
modelFunction(x = housePredictors, y = price)
```

In this case, transformations of data or dummy variables must be created prior to being passed to the function.

Note that not all R functions have both interfaces.

Building and Predicting Models

Modeling in R generally follows the same workflow:

1. Create the model using the basic function:

   ```
   fit <- knn(trainingData, outcome, k = 5)
   ```

2. Assess the properties of the model using `print`, `plot`. `summary` or other methods
3. Predict outcomes for samples using the `predict` method: `predict(fit, newSamples)`.

The model can be used for prediction without changing the original model object.

Model Function Consistency

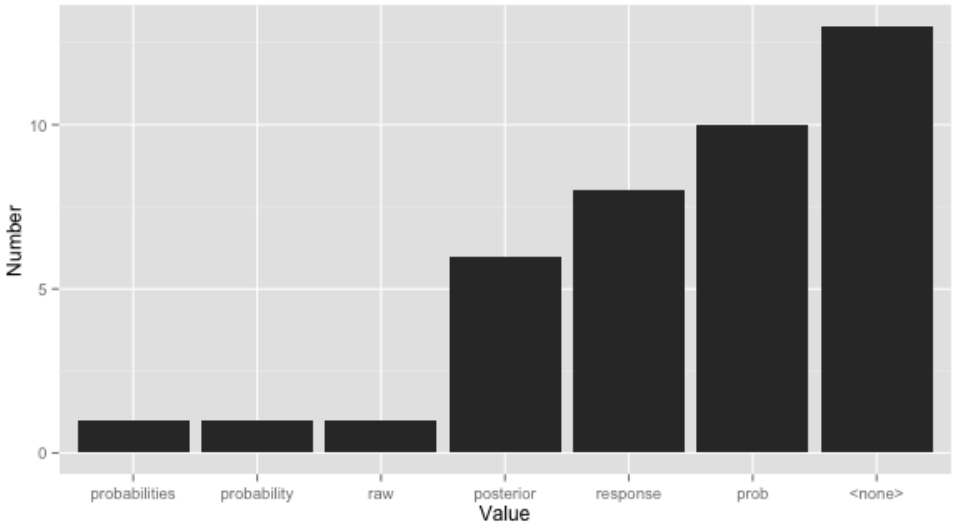
Since there are many modeling packages written by different people, there are some inconsistencies in how models are specified and predictions are made.

For example, many models have only one method of specifying the model (e.g. formula method only)

Generating Class Probabilities Using Different Packages

`obj` Class	Package	`predict` Function Syntax
`lda`	`MASS`	`predict(obj)` (no options needed)
`glm`	`stats`	`predict(obj, type = "response")`
`gbm`	`gbm`	`predict(obj, type = "response", n.trees)`
`mda`	`mda`	`predict(obj, type = "posterior")`
`rpart`	`rpart`	`predict(obj, type = "prob")`
`Weka`	`RWeka`	`predict(obj, type = "probability")`
`LogitBoost`	`caTools`	`predict(obj, type = "raw", nIter)`

`type = "what?"` (Per Package)



The `caret` Package

The `caret` package was developed to:

- create a unified interface for modeling and prediction (interfaces to 147 models)
- streamline model tuning using resampling
- provide a variety of "helper" functions and classes for day–to–day model building tasks
- increase computational efficiency using parallel processing

First commits within Pfizer: 6/2005

First version on CRAN: 10/2007

Website/detailed help pages: http://caret.r-forge.r-project.org

JSS Paper: http://www.jstatsoft.org/v28/i05/paper

Applied Predictive Modeling Blog: http://appliedpredictivemodeling.com/

Illustrative Data: Image Segmentation

We'll use data from Hill *et al* (2007) to model how well cells in an image are segmented (i.e. identified) in "high content screening" (Abraham *et al*, 2004).

Cells can be stained to bind to certain components of the cell (e.g. nucleus) and fixed in a substance that preserves the nature state of the cell.

The sample is then interrogated by an instrument (such as a confocal microscope) where the dye deflects light and the detectors quantify that degree of scattering for that specific wavelength.

If multiple characteristics of the cells are desired, then multiple dyes and multiple light frequencies can be used simultaneously.

The light scattering measurements are then processed through imaging software to quantify the desired cell characteristics.

Illustrative Data: Image Segmentation

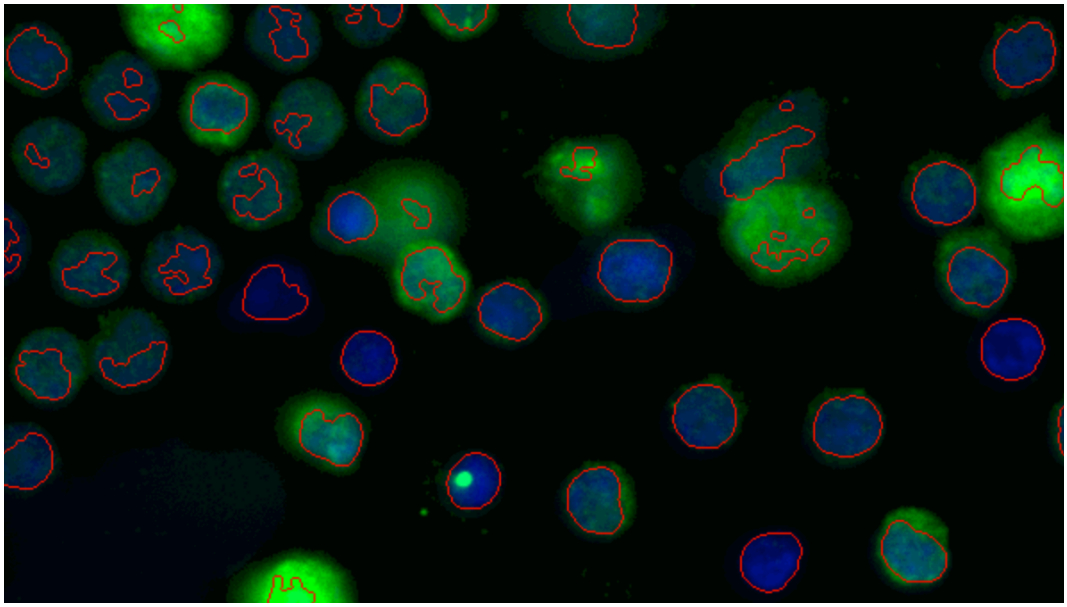
In these images, the bright green boundaries identify the cell nucleus, while the blue boundaries define the cell perimeter.

Clearly some cells are *well–segmented*, meaning that they have an accurate assessment of the location and size of the cell. Others are poorly segmented.

If cell size, shape, and/or quantity are the endpoints of interest in a study, then it is important that the instrument and imaging software can correctly segment cells.

Given a set of image measurements, how well can we predict which cells are well–segmented (WS) or poorly–segmented (PS)?

Illustrative Data: Image Segmentation



Illustrative Data: Image Segmentation

The authors scored 2019 cells into these two bins.

They used four stains to highlight the cell body, the cell nucleus, actin and tubulin (parts of the cytoskeleton).

These correspond to different optical channels (e.g. channel 3 measures actin filaments).

The data are in the `caret` package.

The authors designated a training set ($n = 1009$) and a test set ($n = 1010$).

Illustrative Data: Image Segmentation

```
> data(segmentationData)
> # get rid of the cell identifier
> segmentationData$Cell <- NULL
>
> training <- subset(segmentationData, Case == "Train")
> testing <- subset(segmentationData, Case == "Test")
> training$Case <- NULL
> testing$Case <- NULL
> str(training[,1:6])

data.frame: 1009 obs. of  6 variables:
 $ Class      : Factor w/ 2 levels "PS","WS": 1 2 1 2 1 1 1 2 2 2 ...
 $ AngleCh1   : num  133.8 106.6 69.2 109.4 104.3 ...
 $ AreaCh1    : int  819 431 298 256 258 358 158 315 246 223 ...
 $ AvgIntenCh1: num  31.9 28 19.5 18.8 17.6 ...
 $ AvgIntenCh2: num  207 116 102 127 125 ...
 $ AvgIntenCh3: num  69.9 63.9 28.2 13.6 22.5 ...
```

Since channel 1 is the cell body, `AreaCh1` measures the size of the cell.

Data Splitting and Estimating Performance

Model Building Steps

Common steps during model building are:

- estimating model parameters (i.e. training models)
- determining the values of tuning parameters that cannot be directly calculated from the data
- calculating the performance of the final model that will generalize to new data

How do we "spend" the data to find an optimal model? We typically split data into training and test data sets:

- **Training Set**: these data are used to estimate model parameters and to pick the values of the complexity parameter(s) for the model.
- **Test Set** (aka validation set): these data can be used to get an independent assessment of model efficacy. They should not be used during model training.

Spending Our Data

The more data we spend, the better estimates we'll get (provided the data is accurate). Given a fixed amount of data,

- too much spent in training won't allow us to get a good assessment of predictive performance. We may find a model that fits the training data very well, but is not generalizable (over–fitting)
- too much spent in testing won't allow us to get a good assessment of model parameters

Statistically, the best course of action would be to use all the data for model building and use statistical methods to get good estimates of error.

From a non–statistical perspective, many consumers of of these models emphasize the need for an untouched set of samples the evaluate performance.

Spending Our Data

There are a few different ways to do the split: simple random sampling, **stratified sampling based on the outcome**, by date and methods that focus on the distribution of the predictors.

The base `R` function `sample` can be used to create a completely random sample of the data. The `caret` package has a function `createDataPartition` that conducts data splits within groups of the data.

For classification, this would mean sampling within the classes as to preserve the distribution of the outcome in the training and test sets

For regression, the function determines the quartiles of the data set and samples within those groups

Estimating Performance

Later, once you have a set of predictions, various metrics can be used to evaluate performance.

For regression models:

- R^2 is very popular. In many complex models, the notion of the model degrees of freedom is difficult. Unadjusted R^2 can be used, but does not penalize complexity. (`caret:::RMSE`, `pls:::RMSEP`)
- the **root mean square error** is a common metric for understanding the performance (`caret:::Rsquared`, `pls:::R2`)
- **Spearman's correlation** may be applicable for models that are used to rank samples (`cor(, method = "spearman")`)

Of course, honest estimates of these statistics cannot be obtained by predicting the same samples that were used to train the model.

A test set and/or resampling can provide good estimates.

Estimating Performance For Classification

For classification models:

- **overall accuracy** can be used, but this may be problematic when the classes are not balanced.
- the **Kappa statistic** takes into account the expected error rate:

$$\kappa = \frac{O - E}{1 - E}$$

  where O is the observed accuracy and E is the expected accuracy under chance agreement (`psych:::cohen.kappa`, `vcd:::Kappa`, ...)
- For 2–class models, **Receiver Operating Characteristic** (ROC) curves can be used to characterize model performance (more later)

Estimating Performance For Classification

A " confusion matrix" is a cross–tabulation of the observed and predicted classes

`R` functions for confusion matrices are in the `e1071` package (the `classAgreement` function), the `caret` package (`confusionMatrix`), the `mda` (`confusion`) and others.

ROC curve functions are found in the `ROCR` package (`performance`), the `verification` package (`roc.area`), the `pROC` package (`roc`) and others.

We'll use the `confusionMatrix` function and the `pROC` package later in this class.

Estimating Performance For Classification

For 2–class classification models we might also be interested in:

- **Sensitivity**: given that a result is truly an event, what is the probability that the model will predict an event results?
- **Specificity**: given that a result is truly not an event, what is the probability that the model will predict a negative results?

(an "event" is really the event of interest)

These *conditional* probabilities are directly related to the false positive and false negative rate of a method.

Unconditional probabilities (the positive–predictive values and negative–predictive values) can be computed, but require an estimate of what the overall event rate is in the population of interest (aka the prevalence)

Estimating Performance For Classification

For our example, let's choose the event to be the **poor segmentation** (PS):

$$\text{Sensitivity} = \frac{\#\text{ PS predicted to be PS}}{\#\text{ true PS}}$$

$$\text{Specificity} = \frac{\#\text{ true WS to be WS}}{\#\text{ true WS}}$$

The `caret` package has functions called `sensitivity` and `specificity`

ROC Curve

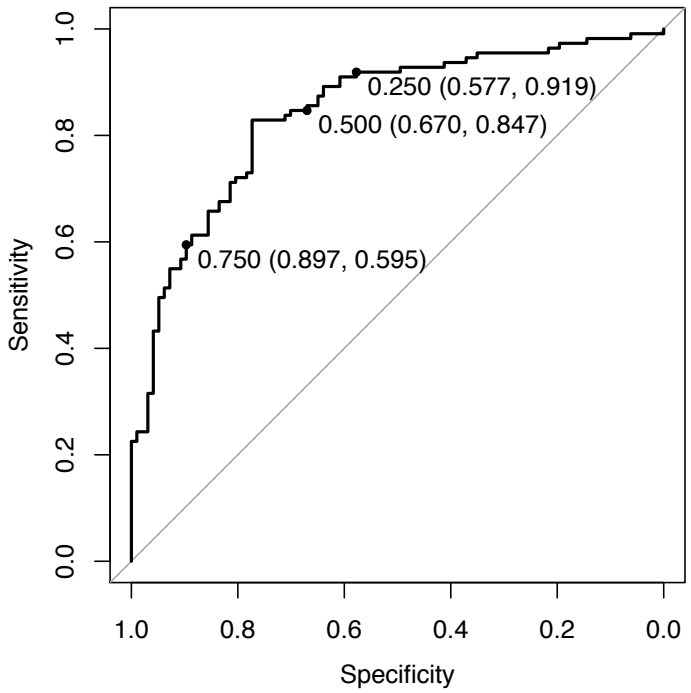
With two classes the Receiver Operating Characteristic (ROC) curve can be used to estimate performance using a combination of sensitivity and specificity.

Given the probability of an event, many alternative cutoffs can be evaluated (instead of just a 50% cutoff). For each cutoff, we can calculate the sensitivity and specificity.

The ROC curve plots the sensitivity (eg. true positive rate) by one minus specificity (eg. the false positive rate).

The area under the ROC curve is a common metric of performance.

ROC Curve From Class Probabilities



Data Pre–Processing

Pre–Processing the Data

There are a wide variety of models in R. Some models have different assumptions on the predictor data and may need to be pre–processed.

For example, methods that use the inverse of the predictor cross–product matrix (i.e. $(X'X)^{-1}$) may require the elimination of collinear predictors.

Others may need the predictors to be centered and/or scaled, etc.

If any data processing is required, it is a good idea to base these calculations on the training set, then apply them to any data set used for model building or prediction.

Pre–Processing the Data

Examples of of pre–processing operations:

- centering and scaling
- imputation of missing data
- transformations of individual predictors (e.g. Box–Cox transformations of the *predictors*)
- transformations of the groups of predictors, such as the
  - the "spatial–sign" transformation (i.e. $x' = x/||x||$)
  - feature extraction via PCA or ICA

Centering, Scaling and Transforming

There are a few different functions for data processing in `R`:

- `scale` in base `R`
- `ScaleAdv` in `pcaPP`
- `stdize` in `pls`
- `preProcess` in `caret`
- `normalize` in `sparseLDA`

The first three functions do simple centering and scaling. `preProcess` can do a variety of techniques, so we'll look at this in more detail.

Centering and Scaling

The input is a matrix or data frame of predictor data. Once the values are calculated, the `predict` method can be used to do the actual data transformations.

First, estimate the standardization parameters:

```
> trainX <- training[, names(training) != "Class"]
> ## Methods are "BoxCox", "YeoJohnson", center", "scale",
> ## "range", "knnImpute", "bagImpute", "pca", "ica" and
> ## "spatialSign"
> preProcValues <- preProcess(trainX, method = c("center", "scale"))
> preProcValues

Call:
preProcess.default(x = trainX, method = c("center", "scale"))

Created from 1009 samples and 58 variables
Pre-processing: centered, scaled
```

Apply them to the data sets:

```
> scaledTrain <- predict(preProcValues, trainX)
```


Pre–Processing and Resampling

To get honest estimates of performance, all data transformations should be included within the cross–validation loop.

The would be especially true for feature selection as well as pre–processing techniques (e.g. imputation, PCA, etc)

One function considered later called `train` that can apply `preProcess` within resampling loops.

Over–Fitting and Resampling

Over–Fitting

Over–fitting occurs when a model inappropriately picks up on trends in the training set that do not generalize to new samples.

When this occurs, assessments of the model based on the training set can show good performance that does not reproduce in future samples.

Some models have specific "knobs" to control over-fitting

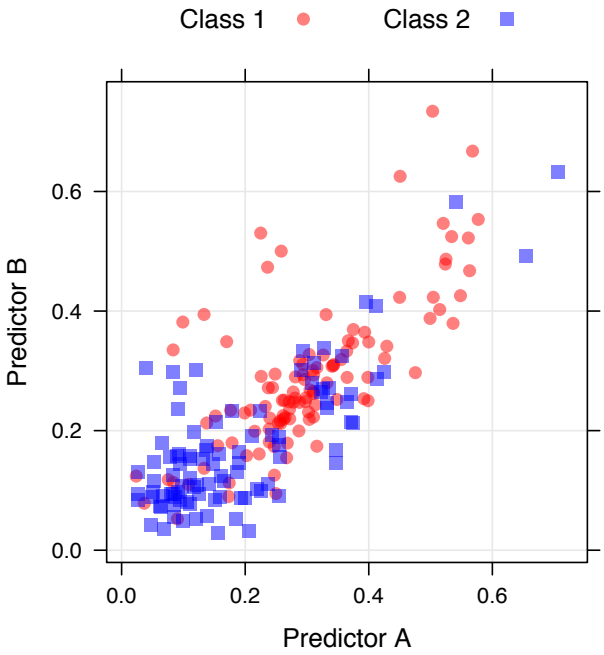
- neighborhood size in nearest neighbor models is an example
- the number if splits in a tree model

Often, poor choices for these parameters can result in over-fitting

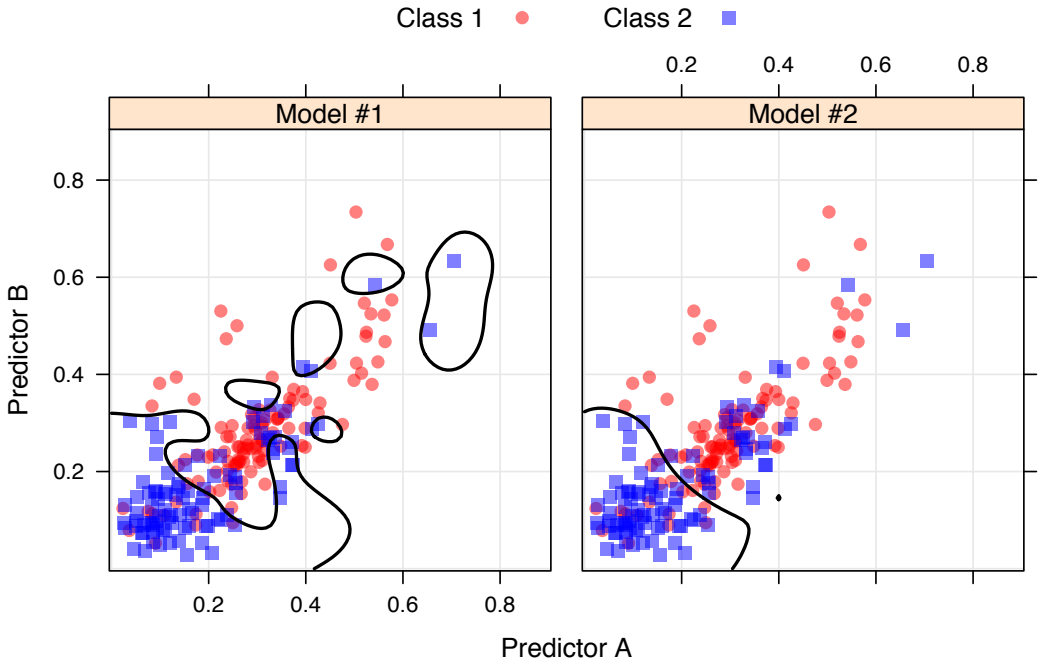
For example, the next slide shows a data set with two predictors. We want to be able to produce a line (i.e. decision boundary) that differentiates two classes of data.

Two model fits are shown; one over–fits the training data.

The Data



Two Model Fits



Characterizing Over–Fitting Using the Training Set

One obvious way to detect over–fitting is to use a test set. However, repeated "looks" at the test set can also lead to over–fitting

Resampling the training samples allows us to know when we are making poor choices for the values of these parameters (the test set is not used).

Resampling methods try to "inject variation" in the system to approximate the model's performance on future samples.

We'll walk through several types of resampling methods for training set samples.

K–Fold Cross–Validation

Here, we randomly split the data into K distinct blocks of roughly equal size.

1. We leave out the first block of data and fit a model.
2. This model is used to predict the held-out block
3. We continue this process until we've predicted all K held–out blocks

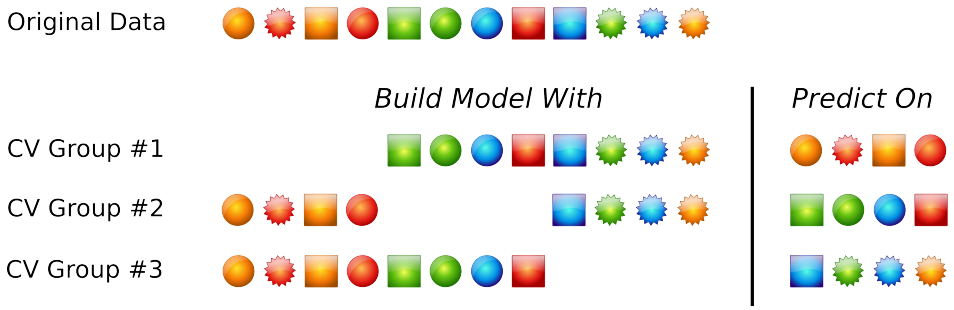
The final performance is based on the hold-out predictions

K is usually taken to be 5 or 10 and leave one out cross–validation has each sample as a block

Repeated K–fold CV creates multiple versions of the folds and aggregates the results (I prefer this method)

`caret:::createFolds`, `caret:::createMultiFolds`

K–Fold Cross–Validation



Repeated Training/Test Splits

(aka leave–group–out cross–validation)

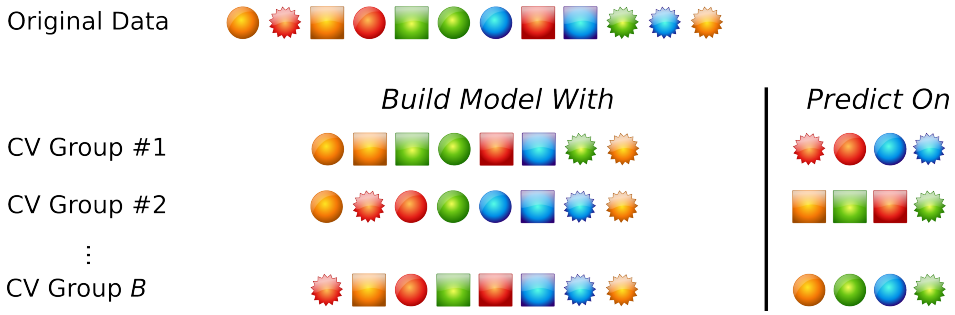
A random proportion of data (say 80%) are used to train a model while the remainder is used for prediction. This process is repeated many times and the average performance is used.

These splits can also be generated using stratified sampling.

With many iterations (20 to 100), this procedure has smaller variance than K–fold CV, but is likely to be biased.

`caret:::createDataPartition`

Repeated Training/Test Splits



Bootstrapping

Bootstrapping takes a random sample with replacement. The random sample is the same size as the original data set.

Samples may be selected more than once and each sample has a 63.2% chance of showing up at least once.

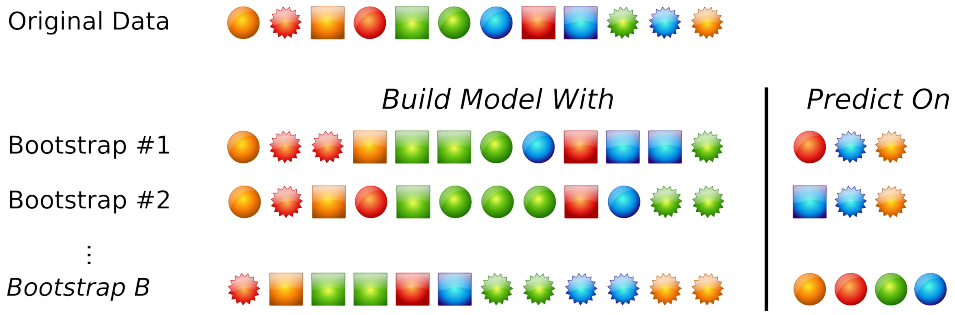
Some samples won't be selected and these samples will be used to predict performance.

The process is repeated multiple times (say 30–100).

This procedure also has low variance but non–zero bias when compared to K–fold CV.

`sample`, `caret:::createResample`

Bootstrapping



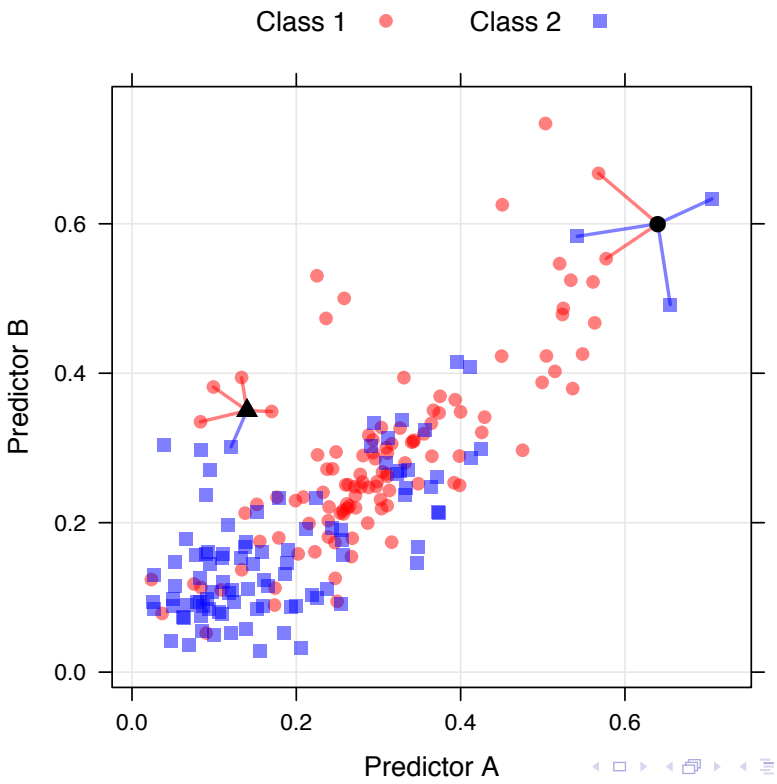
The Big Picture

We think that resampling will give us honest estimates of future performance, but there is still the issue of which model to select.

One algorithm to select models:

```
Define sets of model parameter values to evaluate;
for each parameter set do
    for each resampling iteration do
        Hold–out specific samples ;
        Fit the model on the remainder;
        Predict the hold–out samples;
    end
    Calculate the average performance across hold–out predictions
end
Determine the optimal parameter set;
```

K–Nearest Neighbors Classification

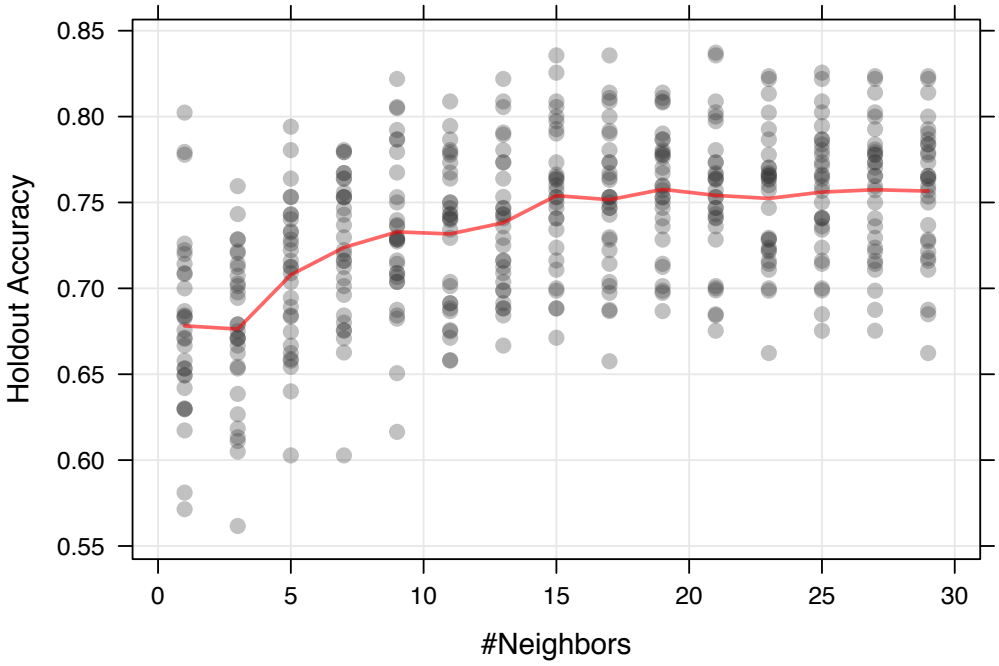


The Big Picture – KNN Example

Using k–nearest neighbors as an example:

Randomly put samples into 10 distinct groups;
for $i = 1 \ldots 30$ **do**
  Create a bootstrap sample;
  Hold–out data not in sample;
  for $k = 1, 3, \ldots 29$ **do**
    Fit the model on the boostrapped sample;
    Predict the i^{th} holdout and save results;
  end
  Calculate the average accuracy across the 30 hold–out sets of predictions
end
Determine k based on the highest cross–validated accuracy;

The Big Picture – KNN Example



A General Strategy

There is usually a inverse relationship between model flexibility/power and interpretability.

In the best case, we would like a parsimonious and interpretable model that has excellent performance.

Unfortunately, that is not usually realistic.

One strategy:

1. start with the most powerful black–box type models
2. get a sense of the best possible performance
3. then fit more simplistic/understandable models
4. evaluate the performance cost of using a simpler model

Training and Tuning Tree Models

Classification Trees

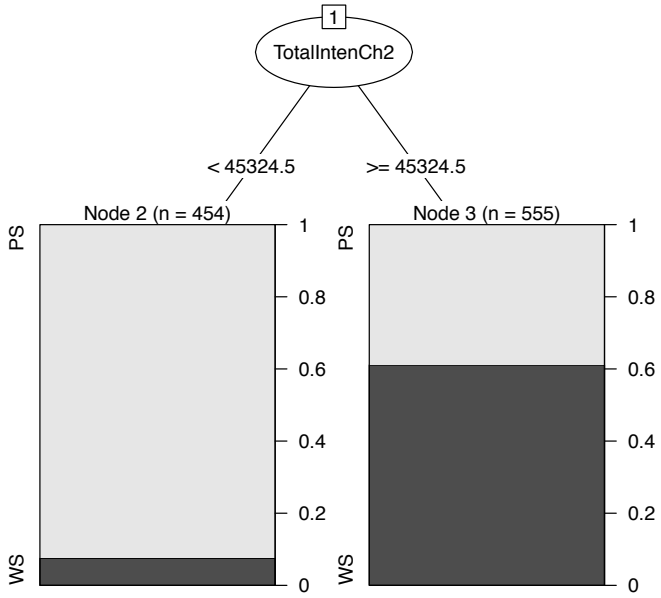
A classification tree searches through each predictor to find a value of a single variable that best splits the data into two groups.

- typically, the best split minimizes impurity of the outcome in the resulting data subsets.

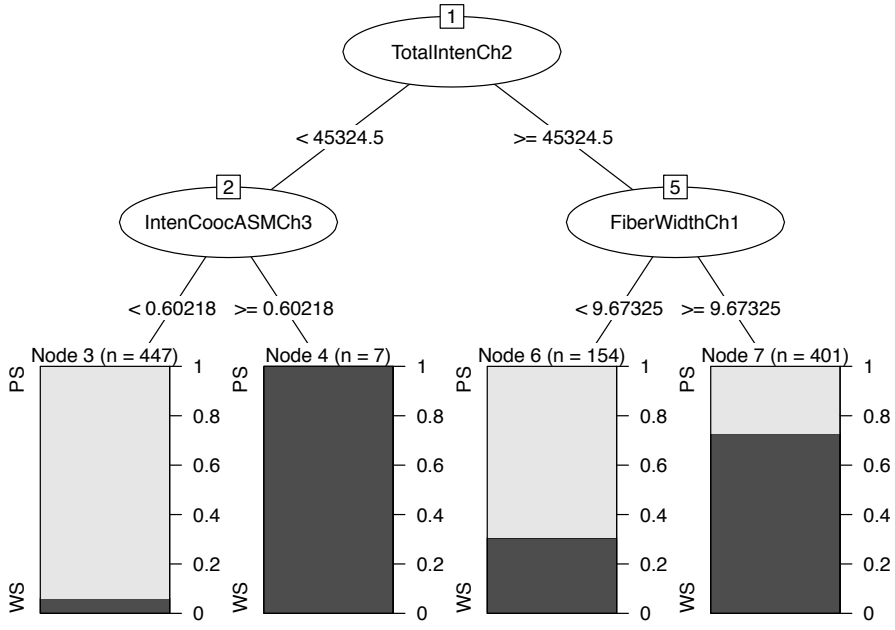
For the two resulting groups, the process is repeated until a hierarchical structure (a tree) is created.

- in effect, trees partition the X space into rectangular sections that assign a single value to samples within the rectangle.

An Example First Split



The Next Round of Splitting



An Example

There are many tree–based packages in R. The main package for fitting single trees are rpart, RWeka, evtree, C50 and party. rpart fits the classical "CART" models of Breiman *et al* (1984).

To obtain a shallow tree with rpart:

```
> library(rpart)
> rpart1 <- rpart(Class ~ ., data = training,
+                 control = rpart.control(maxdepth = 2))
> rpart1
```

```
n= 1009

node), split, n, loss, yval, (yprob)
      * denotes terminal node

1) root 1009 373 PS (0.63032706 0.36967294)
  2) TotalIntenCh2< 45324.5 454  34 PS (0.92511013 0.07488987)
    4) IntenCoocASMCh3< 0.6021832 447  27 PS (0.93959732 0.06040268) *
    5) IntenCoocASMCh3>=0.6021832 7   0 WS (0.00000000 1.00000000) *
  3) TotalIntenCh2>=45324.5 555 216 WS (0.38918919 0.61081081)
    6) FiberWidthCh1< 9.673245 154  47 PS (0.69480519 0.30519481) *
    7) FiberWidthCh1>=9.673245 401 109 WS (0.27182045 0.72817955) *
```

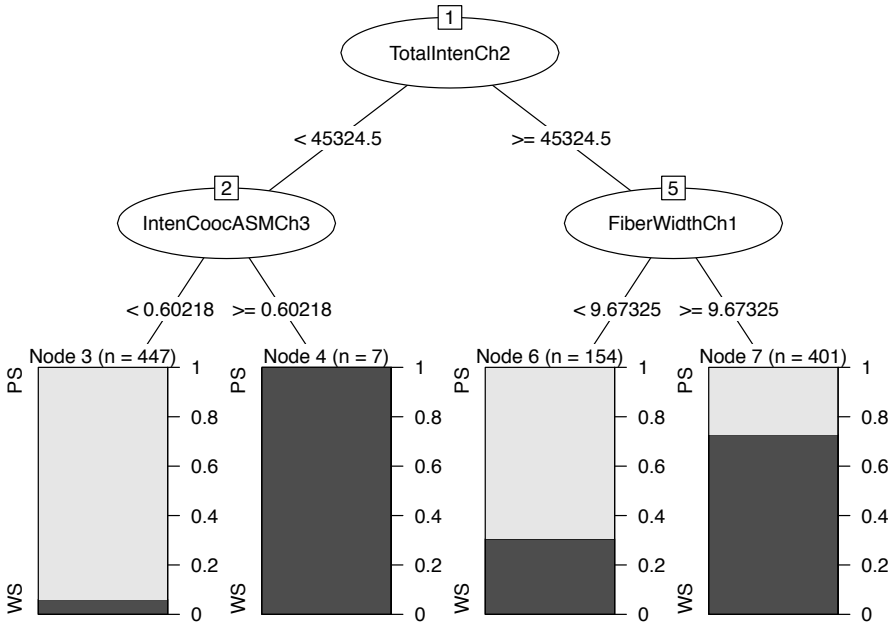

Visualizing the Tree

The rpart package has functions plot.rpart and text.rpart to visualize the final tree.

The partykit package (at r-forge.r-project.org) also has enhanced plotting functions for recursive partitioning. We can convert the rpart object to a new class called party and plot it to see more in the terminal nodes:

```
> rpart1a <- as.party(rpart1)
> plot(rpart1a)
```

A Shallow rpart Tree Using the party Package



Tree Fitting Process

Splitting would continue until some criterion for stopping is met, such as the minimum number of observations in a node

The largest possible tree may over-fit and "pruning" is the process of iteratively removing terminal nodes and watching the changes in resampling performance (usually 10–fold CV)

There are many possible pruning paths: how many possible trees are there with 6 terminal nodes?

Trees can be indexed by their maximum depth and the classical CART methodology uses a cost-complexity parameter (C_p) to determine best tree depth

The Final Tree

Previously, we told `rpart` to use a maximum of two splits.

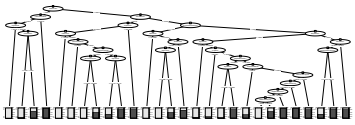
By default, `rpart` will conduct as many splits as possible, then use 10–fold cross–validation to prune the tree.

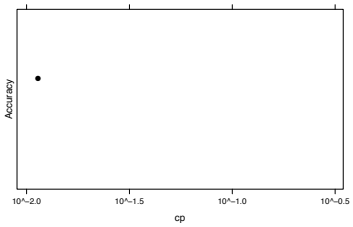
Specifically, the "one SE" rule is used: estimate the standard error of performance for each tree size then choose the simplest tree within one standard error of the absolute best tree size.

```
> rpartFull <- rpart(Class ~ ., data = training)
```

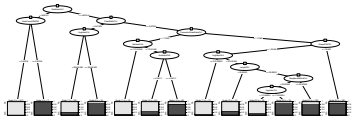

Tree Growing and Pruning

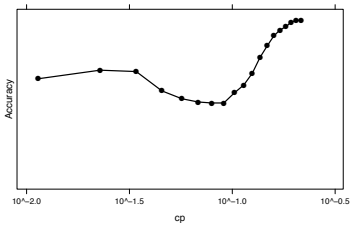
Full Tree



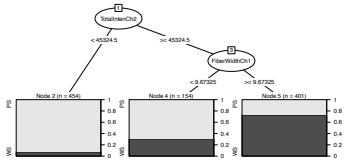


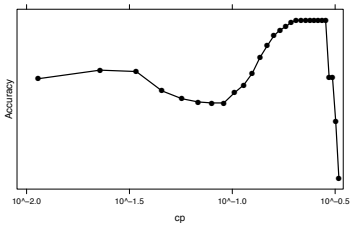
Start Pruning





Too Much!



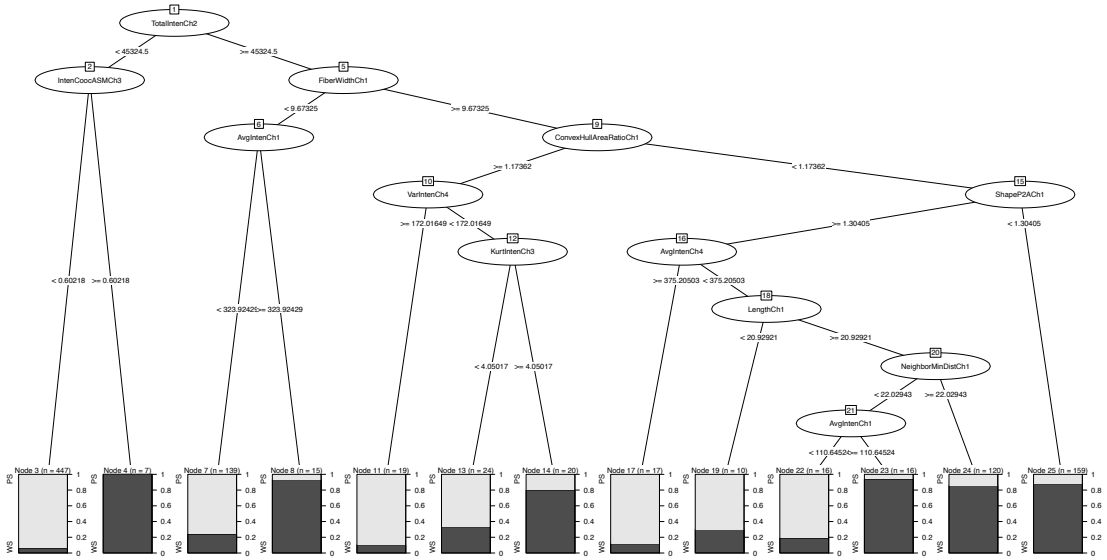


The Final Tree

```
> rpartFull
```

```
n= 1009

node), split, n, loss, yval, (yprob)
      * denotes terminal node

  1) root 1009 373 PS (0.63032706 0.36967294)
    2) TotalIntenCh2< 45324.5 454  34 PS (0.92511013 0.07488987)
      4) IntenCoocASMCh3< 0.6021832 447  27 PS (0.93959732 0.06040268) *
      5) IntenCoocASMCh3>=0.6021832 7   0 WS (0.00000000 1.00000000) *
    3) TotalIntenCh2>=45324.5 555 216 WS (0.38918919 0.61081081)
      6) FiberWidthCh1< 9.673245 154  47 PS (0.69480519 0.30519481)
       12) AvgIntenCh1< 323.9243 139  33 PS (0.76258993 0.23741007) *
       13) AvgIntenCh1>=323.9243 15   1 WS (0.06666667 0.93333333) *
      7) FiberWidthCh1>=9.673245 401 109 WS (0.27182045 0.72817955)
       14) ConvexHullAreaRatioCh1>=1.173618 63  26 PS (0.58730159 0.41269841)
         28) VarIntenCh4>=172.0165 19   2 PS (0.89473684 0.10526316) *
         29) VarIntenCh4< 172.0165 44  20 WS (0.45454545 0.54545455)
           58) KurtIntenCh3< 4.05017 24   8 PS (0.66666667 0.33333333) *
           59) KurtIntenCh3>=4.05017 20   4 WS (0.20000000 0.80000000) *
       15) ConvexHullAreaRatioCh1< 1.173618 338  72 WS (0.21301775 0.78698225)
         30) ShapeP2ACh1>=1.304052 179  53 WS (0.29608939 0.70391061)
           60) AvgIntenCh4>=375.205 17   2 PS (0.88235294 0.11764706) *
           61) AvgIntenCh4< 375.205 162  38 WS (0.23456790 0.76543210)
            122) LengthCh1< 20.92921 10   3 PS (0.70000000 0.30000000) *
            123) LengthCh1>=20.92921 152  31 WS (0.20394737 0.79605263)
              246) NeighborMinDistCh1< 22.02943 32  14 WS (0.43750000 0.56250000)
                492) AvgIntenCh1< 110.6452 16   3 PS (0.81250000 0.18750000) *
                493) AvgIntenCh1>=110.6452 16   1 WS (0.06250000 0.93750000) *
              247) NeighborMinDistCh1>=22.02943 120  17 WS (0.14166667 0.85833333) *
         31) ShapeP2ACh1< 1.304052 159  19 WS (0.11949686 0.88050314) *
```


The Final `rpart` Tree



Test Set Results

```
> rpartPred <- predict(rpartFull, testing, type = "class")
> confusionMatrix(rpartPred, testing$Class)   # requires 2 factor vectors
```

```
Confusion Matrix and Statistics

          Reference
Prediction  PS  WS
        PS 561 108
        WS 103 238

               Accuracy : 0.7911
                 95% CI : (0.7647, 0.8158)
    No Information Rate : 0.6574
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.5346
 Mcnemars Test P-Value : 0.783

            Sensitivity : 0.8449
            Specificity : 0.6879
         Pos Pred Value : 0.8386
         Neg Pred Value : 0.6979
             Prevalence : 0.6574
         Detection Rate : 0.5554
   Detection Prevalence : 0.6624

       Positive Class : PS
```


Manually Tuning the Model

CART conducts an internal 10-fold CV to tune the model to be within one SE of the absolute minimum.

We might want to tune the model ourselves for several reasons:

- 10-Fold CV can be very noisy for small to moderate sample sizes
- We might want to risk some over–fitting in exchange for higher performance
- Using a performance metric other than error may be preferable, especially with severe class imbalances.
- We can manually make tradeoffs between sensitivity and specificity for different values of C_p

To this end, we will look at the `train` funciton in the `caret` package.

Tuning the Model

There are a few functions that can be used for this purpose in R.

- the `errorest` function in the `ipred` package can be used to resample a single model (e.g. a `gbm` model with a specific number of iterations and tree depth)
- the `e1071` package has a function (`tune`) for five models that will conduct resampling over a grid of tuning values.
- `caret` has a similar function called `train` for over 147 models. Different resampling methods are available as are custom performance metrics and facilities for parallel processing.

The `train` Function

The basic syntax for the function is:

```
> train(formula, data, method)
```

Looking at `?train`, using `method = "rpart"` can be used to tune a tree over C_p, so we can use:

```
train(Class ~ ., data = training, method = "rpart")
```

We'll add a bit of customization too.

The train Function

By default, the function will tune over 3 values of the tuning parameter (C_p for this model).

For rpart, the train function determines the distinct number of values of C_p for the data.

The tuneLength function can be used to evaluate a broader set of models:

```
train(Class ~ ., data = training, method = "rpart", tuneLength = 30)
```


The train Function

The default resampling scheme is the bootstrap. Let's use repeated 10–fold cross–validation instead.

To do this, there is a control function that handles some of the optional arguments.

To use three repeats of 10–fold cross–validation, we would use

```
cvCtrl <- trainControl(method = "repeatedcv", repeats = 3)
train(Class ~ ., data = training, method = "rpart",
      tuneLength = 30,
      trControl = cvCtrl)
```

The train Function

Also, the default CART algorithm uses overall accuracy and the one standard–error rule to prune the tree.

We might want to choose the tree complexity based on the largest absolute area under the ROC curve.

A custom performance function can be passed to train. The package has one that calculates the ROC curve, sensitivity and specificity:

```
> ## Make some random example data to show usage of twoClassSummary()
> fakeData <- data.frame(pred = testing$Class,
+                        obs = sample(testing$Class),
+                        ## Requires a column for class probabilities
+                        ## named after the first level
+                        PS = runif(nrow(testing)))
> twoClassSummary(fakeData, lev = levels(fakeData$obs))

      ROC      Sens      Spec
0.5220550 0.6822289 0.3901734
```


The train Function

We can pass the twoClassSummary function in through trainControl.

However, to calculate the ROC curve, we need the model to predict the class probabilities. The classProbs option will also do this:

Finally, we tell the function to optimize the area under the ROC curve using the metric argument:

```
cvCtrl <- trainControl(method = "repeatedcv", repeats = 3,
                       summaryFunction = twoClassSummary,
                       classProbs = TRUE)
set.seed(1)
rpartTune <- train(Class ~ ., data = training, method = "rpart",
                   tuneLength = 30,
                   metric = "ROC",
                   trControl = cvCtrl)
```

train Results

```
> rpartTune
```

```
1009 samples
  58 predictors
   2 classes: PS, WS

No pre-processing
Resampling: Cross-Validation (10 fold, repeated 3 times)

Summary of sample sizes: 909, 907, 908, 908, 908, 909, ...

Resampling results across tuning parameters:

  cp      ROC    Sens   Spec   ROC SD  Sens SD
  0       0.854  0.831  0.685  0.0435  0.0494
  0.0114  0.842  0.845  0.732  0.0513  0.0413
  0.0227  0.843  0.839  0.756  0.0516  0.0422
  0.0341  0.825  0.82   0.752  0.0468  0.0377
  0.0455  0.819  0.83   0.699  0.0431  0.0424

    :       :      :      :      :       :

  0.273   0.772  0.65   0.893  0.0399  0.0565
  0.284   0.772  0.65   0.893  0.0399  0.0565
  0.296   0.724  0.712  0.736  0.109   0.142
  0.307   0.724  0.712  0.736  0.109   0.142
  0.318   0.686  0.758  0.613  0.128   0.166
  0.33    0.635  0.812  0.458  0.132   0.183

ROC was used to select the optimal model using  the largest value.
The final value used for the model was cp = 0.
```


Working With the train Object

There are a few methods of interest:

- plot.train can be used to plot the resampling profiles across the different models
- print.train shows a textual description of the results
- predict.train can be used to predict new samples
- there are a few others that we'll mention shortly.

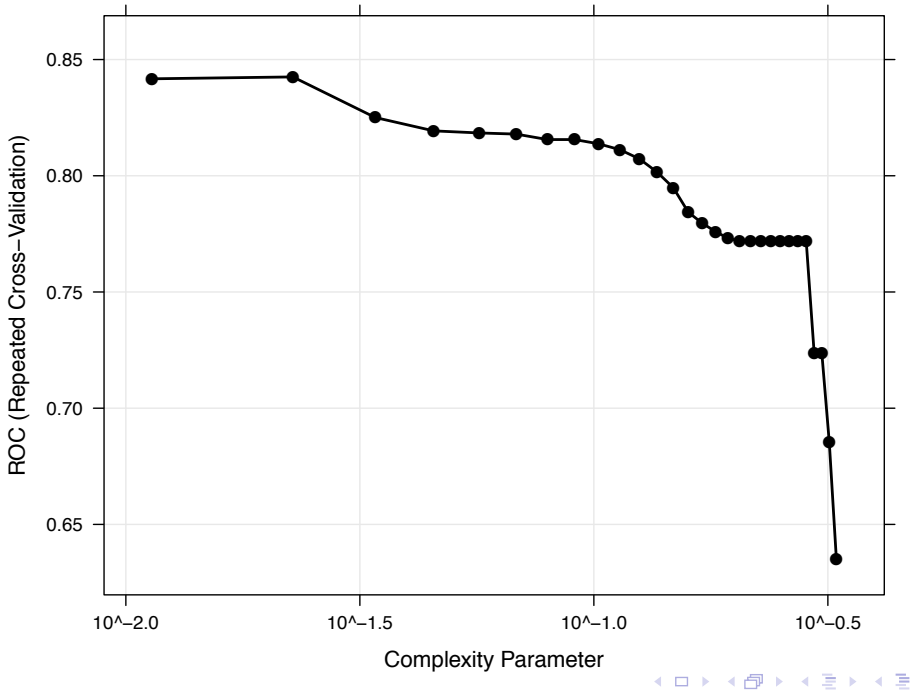
Additionally, the final model fit (i.e. the model with the best resampling results) is in a sub–object called finalModel.

So in our example, rpartTune is of class train and the object rpartTune$finalModel is of class rpart.

Let's look at what the plot method does.

Resampled ROC Profile

```
plot(rpartTune, scales = list(x = list(log = 10)))
```



Predicting New Samples

There are at least two ways to get predictions from a `train` object:

- `predict(rpartTune$finalModel, newdata, type = "class")`
- `predict(rpartTune, newdata)`

The first method uses `predict.rpart`. If there is any extra or non–standard syntax, this must also be specified.

`predict.train` does the same thing, but takes care of any minutia that is specific to the predict method in question.

Test Set Results

```
> rpartPred2 <- predict(rpartTune, testing)
> confusionMatrix(rpartPred2, testing$Class)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  PS  WS
        PS 554 104
        WS 110 242

               Accuracy : 0.7881
                 95% CI : (0.7616, 0.8129)
    No Information Rate : 0.6574
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.5316
 Mcnemars Test P-Value : 0.7325

            Sensitivity : 0.8343
            Specificity : 0.6994
         Pos Pred Value : 0.8419
         Neg Pred Value : 0.6875
             Prevalence : 0.6574
         Detection Rate : 0.5485
   Detection Prevalence : 0.6515

       Positive Class : PS
```


Predicting Class Probabilities

`predict.train` has an argument `type` that can be used to get predicted class probabilities for different models:

```
> rpartProbs <- predict(rpartTune, testing, type = "prob")
> head(rpartProbs)
```

```
          PS         WS
1 0.97681159 0.02318841
5 0.97681159 0.02318841
6 0.06034483 0.93965517
7 0.06034483 0.93965517
8 0.97681159 0.02318841
9 0.97681159 0.02318841
```

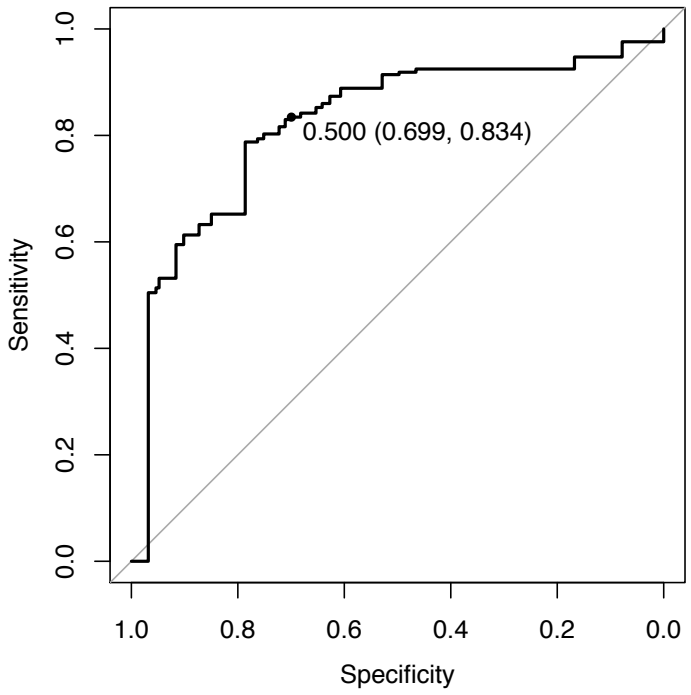
Creating the ROC Curve

The pROC package can be used to create ROC curves.

The function `roc` is used to capture the data and compute the ROC curve. The functions `plot.roc` and `auc.roc` generate plot and area under the curve, respectively.

```
> library(pROC)
> rpartROC <- roc(testing$Class, rpartProbs[, "PS"], levels = rev(testProbs$Class))
> plot(rpartROC, type = "S", print.thres = .5)

> rpartROC

Call:
roc.default(response = testing$Class, predictor = rpartProbs[,     "PS"], levels =

Data: rpartProbs[, "PS"] in 346 controls (testing$Class 2) < 664 cases (testing$Cla
Area under the curve: 0.8436
```


Classification Tree ROC Curve



Pros and Cons of Single Trees

Trees can be computed very quickly and have simple interpretations.

Also, they have built-in feature selection; if a predictor was not used in any split, the model is completely independent of that data.

Unfortunately, trees do not usually have optimal performance when compared to other methods.

Also, small changes in the data can drastically affect the structure of a tree.

This last point has been exploited to improve the performance of trees via ensemble methods where many trees are fit and predictions are aggregated across the trees. Examples are bagging, **boosting** and random forests.

Boosting Algorithms

A method to "boost" weak learning algorithms (e.g. single trees) into strong learning algorithms.

Boosted trees try to improve the model fit over different trees by considering past fits (not unlike iteratively reweighted least squares)

The basic tree boosting algorithm:

Initialize equal weights per sample;
for $j = 1 \ldots M$ *iterations* **do**
  Fit a classification tree using sample weights (denote the model equation as $f_j(x)$);
  forall the *misclassified samples* **do**
    increase sample weight
  end
  Save a "stage–weight" (β_j) based on the performance of the current model;
end

Boosted Trees (Original " adaBoost" Algorithm)

In this formulation, the categorical response y_i is coded as either $\{-1, 1\}$ and the model $f_j(x)$ produces values of $\{-1, 1\}$.

The final prediction is obtained by first predicting using all M trees, then weighting each prediction

$$f(x) = \frac{1}{M} \sum_{j=1}^{M} \beta_j f_j(x)$$

where f_j is the j^{th} tree fit and β_j is the stage weight for that tree.

The final class is determined by the sign of the model prediction.

In English: the final prediction is a weighted average of each tree's prediction. The weights are based on quality of each tree.

Advances in Boosting

Although originally developed in the computer science literature, statistical views on boosting made substantive improvements to the model.

For example:

- different loss functions could be used, such as squared-error loss, binomial likelihood etc
- the rate at which the boosted tree adapts to new model fits could be controlled and tweaked
- sub–sampling the training set within each boosting iteration could be used to increase model performance

The most well known statistical boosting model is the *stochastic gradient boosting* of Friedman (2002).

However, there are numerous approaches to boosting. Many use non–tree models as the *base learner*.

Boosting Packages in R

Boosting functions for *trees* in R:

- `gbm` in the `gbm` package
- `ada` in `ada`
- `blackboost` in `mboost`
- `C50` in `C50`

There are also packages for boosting other models (e.g. the `mboost` package)

The CRAN Machine Learning Task View has a more complete list:

```
http://cran.r-project.org/web/views/MachineLearning.html
```


Boosting via C5.0

C5.0 is an evolution of the C4.5 of Quinlan (1993). Compared to CART, *some* of the differences are:

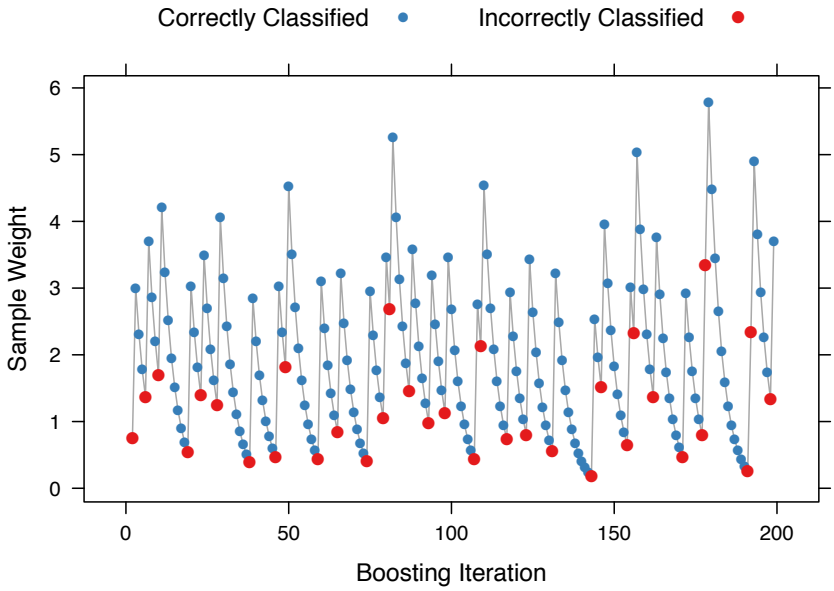
- A different impurity measure is used (entropy)
- Tree pruning is done using *pessimistic pruning*
- Splits on categorical predictors are handled very differently

While the stochastic gradient boosting machines diverged from the original adaboost algorithm, C5.0 does something similar to adaboost.

After the first tree is created, weights are determined and subsequent iterations create weighted trees of about the same size as the first.

The final prediction is a simple average (i.e. no stage weights).

C5.0 Weighting Scheme



C5.0 Syntax

This function has pretty standard syntax:

```
> C50(x = predictors, y = factorOutcome)
```

There are a few arguments:

- `trials`: the number of boosting iterations
- `rules`: a logical to indicate whether the tree(s) should be collapsed into rules.
- `winnow`: a logical that enacts a feature selection step prior to model building.
- `costs`: a matrix that places unbalanced costs of different types of errors.

The prediction code is fairly standard

```
predict(object, trials, type)
## type is "class" or "prob"
## trials can be <= the original argument
```

Tuning the C5.0 Model

Let's use the basic tree model (i.e. no rules) with no additional feature selection.

We will tune the model over the number of boosting iterations (1 . . . 100)

Note that we do no have to fit 100 models for each iteration of resampling. We fit the 100 iteration model and derive the other 99 predictions using just the `predict` method.

We call this the "sub–model" trick. `train` uses it whenever possible (including `blackboost`, `C5.0`, `cubist`, `earth`, `enet`, `gamboost`, `gbm`, `glmboost`, `glmnet`, `lars`, `lasso`, `logitBoost`, `pam`, `pcr`, `pls`, `rpart` and others)

Using Different Performance Metrics

`train` was designed to make the syntax changes between models minimal. Here, we will specify exactly what values the model should tune over.

A data frame is used with one row per tuning variable combination and the parameters start with periods.

```
grid <- expand.grid(.model = "tree",
                    .trials = c(1:100),
                    .winnow = FALSE)

c5Tune <- train(trainX, training$Class,
                method = "C5.0",
                metric = "ROC",
                tuneGrid = grid,
                trControl = cvCtrl)
```

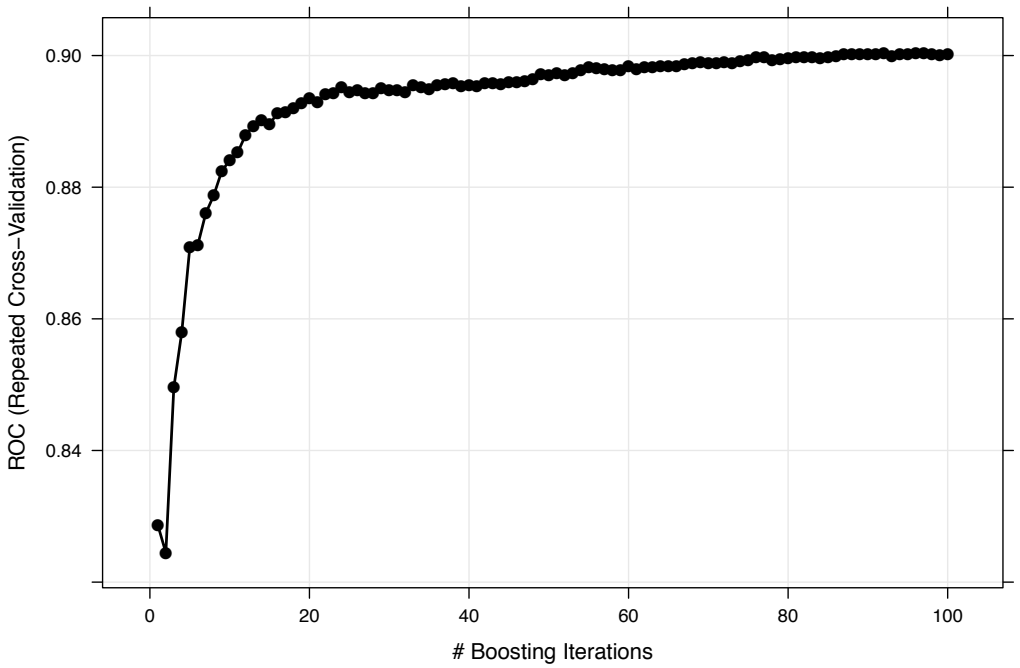
Model Output

```
> c5Tune
```

```
1009 samples
  58 predictors
   2 classes: PS, WS

No pre-processing
Resampling: Cross-Validation (10 fold, repeated 3 times)

Summary of sample sizes: 909, 907, 908, 908, 908, 909, ...

Resampling results across tuning parameters:

  trials  ROC    Sens   Spec   ROC SD  Sens SD  Spec SD
  1       0.829  0.82   0.694  0.0489  0.0515   0.0704
  2       0.824  0.855  0.676  0.0456  0.05     0.0789
  3       0.85   0.841  0.714  0.0406  0.0452   0.0775
  4       0.858  0.852  0.706  0.0322  0.0469   0.069
  :         :      :      :      :       :        :
  95      0.9    0.859  0.764  0.0335  0.0391   0.0606
  96      0.9    0.859  0.763  0.0331  0.0397   0.0629
  97      0.9    0.859  0.76   0.0336  0.0387   0.0652
  98      0.9    0.857  0.761  0.0334  0.0397   0.0662
  99      0.9    0.858  0.762  0.0334  0.0393   0.066
  100     0.9    0.859  0.764  0.0331  0.0393   0.069

Tuning parameter model was held constant at a value of tree

Tuning parameter winnow was held constant at a value of 0
ROC was used to select the optimal model using  the largest value.
The final values used for the model were model = tree, trials = 97 and winnow
 = FALSE.
```


Boosted Tree Resampling Profile

`plot(c5Tune)`



Test Set Results

```
> c5Pred <- predict(c5Tune, testing)
> confusionMatrix(c5Pred, testing$Class)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  PS  WS
        PS 561  84
        WS 103 262

               Accuracy : 0.8149
                 95% CI : (0.7895, 0.8384)
    No Information Rate : 0.6574
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.5943
 Mcnemars Test P-Value : 0.1881

            Sensitivity : 0.8449
            Specificity : 0.7572
         Pos Pred Value : 0.8698
         Neg Pred Value : 0.7178
             Prevalence : 0.6574
         Detection Rate : 0.5554
   Detection Prevalence : 0.6386

       Positive Class : PS
```


Test Set ROC Curve

```
> c5Probs <- predict(c5Tune, testing, type = "prob")
> head(c5Probs, 3)
```

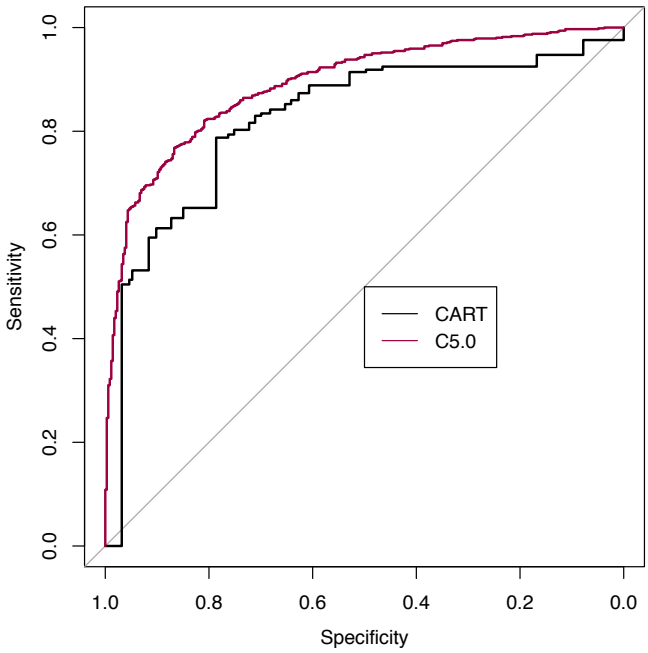
```
          PS        WS
1 0.77879280 0.2212072
5 0.87112431 0.1288757
6 0.07587952 0.9241205
```

```
> library(pROC)
> c5ROC <- roc(predictor = c5Probs$PS,
+              response = testing$Class,
+              levels = rev(levels(testing$Class)))
> c5ROC
```

```
Call:
roc.default(response = testing$Class, predictor = c5Probs$PS,     levels = rev(leve

Data: c5Probs$PS in 346 controls (testing$Class WS) < 664 cases (testing$Class PS).
Area under the curve: 0.8909
```

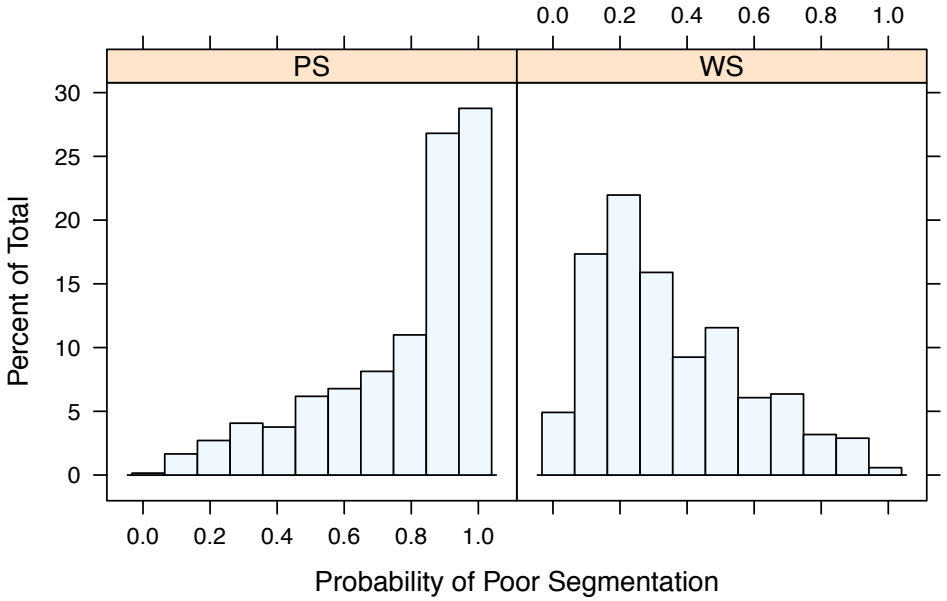
```
> # plot(rpartROC, type = "S")
> # plot(c5ROC, add = TRUE, col = "#9E0142")
```

Test Set ROC Curves



Test Set Probabilities

```
> histogram(~c5Probs$PS|testing$Class, xlab = "Probability of Poor Segmentation")
```



Support Vector Machines (SVM)

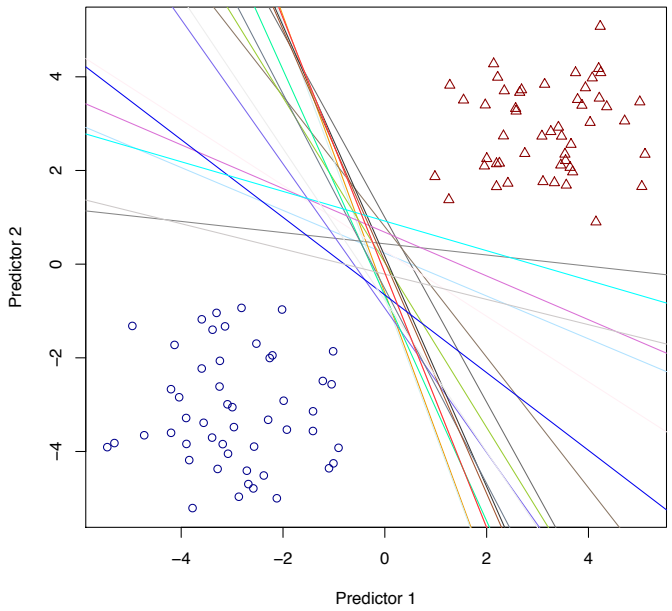
This is a class of powerful and very flexible models.

SVMs for classification use a completely different objective function called the **margin**.

Suppose a hypothetical situation: a dataset of two predictors and we are trying to predict the correct class (two possible classes).

Let's further suppose that these two predictors completely separate the classes

Support Vector Machines



Support Vector Machines (SVM)

There are an infinite number of straight lines that we can use to separate these two groups. Some must be better than others...

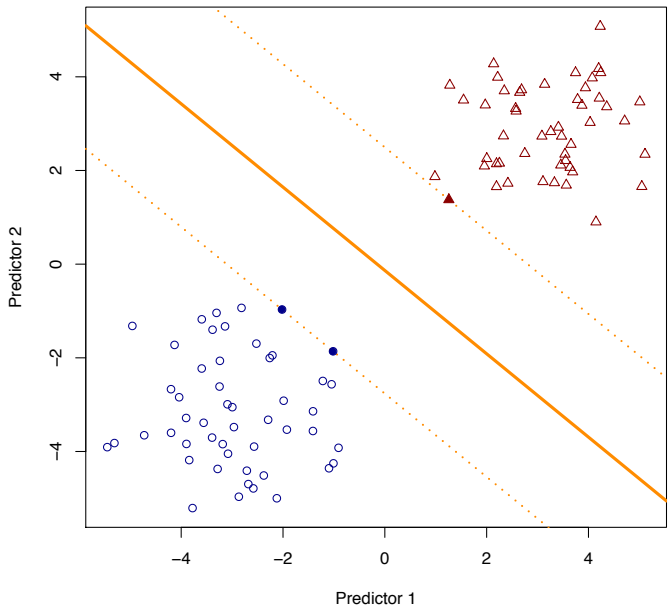
The margin is a defined by equally spaced boundaries on each side of the line.

To maximize the margin, we try to make it as large as possible without capturing any samples.

As the margin increases, the solution becomes more robust.

SVMs determine the slope and intercept of the line which maximizes the margin.

Maximizing the Margin



SVM Prediction Function

Suppose our two classes are coded as –1 or 1.

The SVM model estimates n parameters ($\alpha_1 \ldots \alpha_n$) for the model. Regularization is used to avoid saturated models (more on that in a minute).

For a new point u, the model predicts:

$$f(u) = \beta_0 + \sum_{i=1}^{n} \alpha_i y_i x_i' u$$

The decision rule would predict class $\#1$ if $f(u) > 0$ and class $\#2$ otherwise.

Data points that are support vectors have $\alpha_i \neq 0$, so the prediction equation is only affected by the support vectors.

Support Vector Machines (SVM)

When the classes overlap, points are allowed within the margin. The number of points is controlled by a cost parameter.

The points that are within the margin (or on it's boundary) are the support vectors

Consequences of the fact that the prediction function only uses the support vectors:

- the prediction equation is more compact and efficient
- the model may be more robust to outliers

The Kernel Trick

You may have noticed that the prediction function was a function of an inner product between two samples vectors ($x_i'u$). It turns out that this opens up some new possibilities.

Nonlinear class boundaries can be computed using the "kernel trick".

The predictor space can be expanded by adding nonlinear functions in x. These functions, which must satisfy specific mathematical criteria, include common functions:

$$\text{Polynomial} : K(x, u) = (1 + x'u)^p$$

$$\text{Radial basis function} : K(x, u) = exp\left[\frac{-\sigma}{2}(x - u)^2\right]$$

We don't need to store the extra dimensions; these functions can be computed quickly.

SVM Regularization

As previously discussed, SVMs also include a regularization parameter that controls how much the regression line can adapt to the data smaller values result in more linear (i.e. flat) surfaces

This parameter is generally referred to as "Cost"

If the cost parameter is large, there is a significant penalty for having samples within the margin $\Rightarrow$ the boundary becomes very flexible.

Tuning the cost parameter, as well as any kernel parameters, becomes very important as these models have the ability to greatly over–fit the training data.

(animation)

SVM Models in R

There are several packages that include SVM models:

- `e1071` has the function `svm` for classification (2+ classes) and regression with 4 kernel functions
- `klaR` has `svmlight` which is an interface to the C library of the same name. It can do classification and regression with 4 kernel functions (or user defined functions)
- `svmpath` has an efficient function for computing 2–class models (including 2 kernel functions)
- `kernlab` contains `ksvm` for classification and regression with 9 built–in kernel functions. Additional kernel function classes can be written. Also, `ksvm` can be used for text mining with the `tm` package.

Personally, I prefer `kernlab` because it is the most general and contains other kernel method functions (`e1071` is probably the most popular).

Tuning SVM Models

We need to come up with reasonable choices for the cost parameter and any other parameters associated with the kernel, such as

- polynomial degree for the polynomial kernel
- σ, the scale parameter for radial basis functions (RBF)

We'll focus on RBF kernel models here, so we have two tuning parameters.

However, there is a potential shortcut for RBF kernels. Reasonable values of σ can be derived from elements of the kernel matrix of the training set.

The manual for the `sigest` function in `kernlab` has "The estimation [for σ] is based upon the 0.1 and 0.9 quantile of $|x - x'|^2$."

Anecdotally, we have found that the mid–point between these two numbers can provide a good estimate for this tuning parameter. This leaves only the cost function for tuning.

SVM Example

We can tune the SVM model over the cost parameter.

```
set.seed(1)
svmTune <- train(x = trainX,
                 y = training$Class,
                 method = "svmRadial",
                 # The default grid of cost parameters go from 2^-2,
                 # 0.5 to 1,
                 # Well fit 9 values in that sequence via the tuneLength
                 # argument.
                 tuneLength = 9,
                 ## Also add options from preProcess here too
                 preProc = c("center", "scale"),
                 metric = "ROC",
                 trControl = cvCtrl)
```


SVM Example

```
> svmTune
```

```
1009 samples
  58 predictors
   2 classes: PS, WS

Pre-processing: centered, scaled
Resampling: Cross-Validation (10 fold, repeated 3 times)

Summary of sample sizes: 909, 907, 908, 908, 908, 909, ...

Resampling results across tuning parameters:

  C     ROC    Sens   Spec   ROC SD  Sens SD  Spec SD
  0.25  0.877  0.876  0.665  0.0355  0.0421   0.076
  0.5   0.882  0.867  0.721  0.0365  0.0407   0.078
  1     0.883  0.861  0.741  0.035   0.0421   0.073
  2     0.876  0.851  0.734  0.034   0.0349   0.0733
  4     0.864  0.847  0.72   0.0347  0.0328   0.0679
  8     0.852  0.84   0.718  0.0383  0.0475   0.0771
  16    0.839  0.828  0.703  0.0434  0.0484   0.0855
  32    0.826  0.816  0.693  0.0441  0.0498   0.0966
  64    0.824  0.814  0.687  0.0445  0.05     0.0977

Tuning parameter sigma was held constant at a value of 0.0208
ROC was used to select the optimal model using  the largest value.
The final values used for the model were C = 1 and sigma = 0.0208.
```

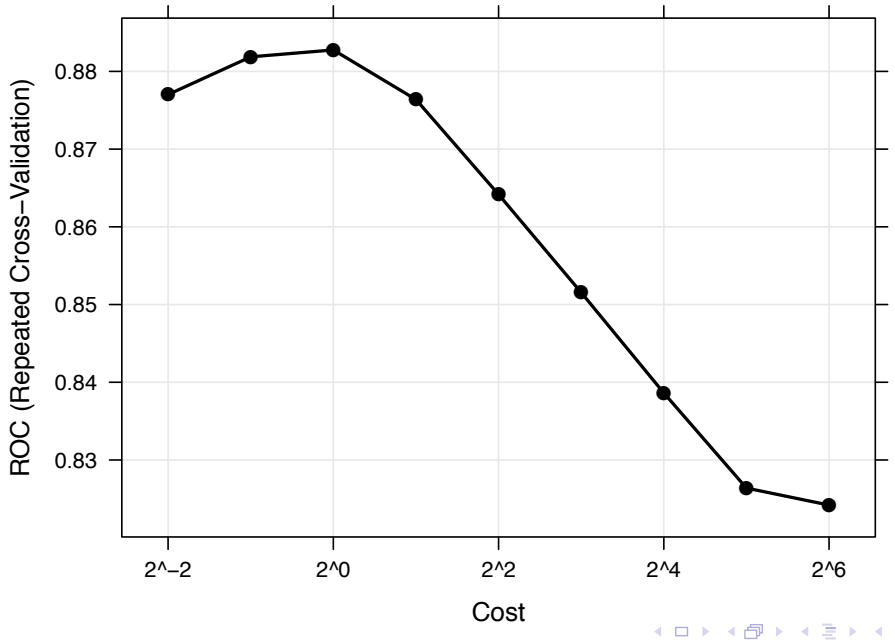
SVM Example

```
> svmTune$finalModel

Support Vector Machine object of class "ksvm"

SV type: C-svc  (classification)
 parameter : cost C = 1

Gaussian Radial Basis kernel function.
 Hyperparameter : sigma =  0.0207957172685357

Number of Support Vectors : 566

Objective Function Value : -374.7523
Training error : 0.108028
Probability model included.
```

566 training data points (out of 1009 samples) were used as support vectors.

SVM Accuracy Profile

```
plot(svmTune, metric = "ROC", scales = list(x = list(log = 2)))
```

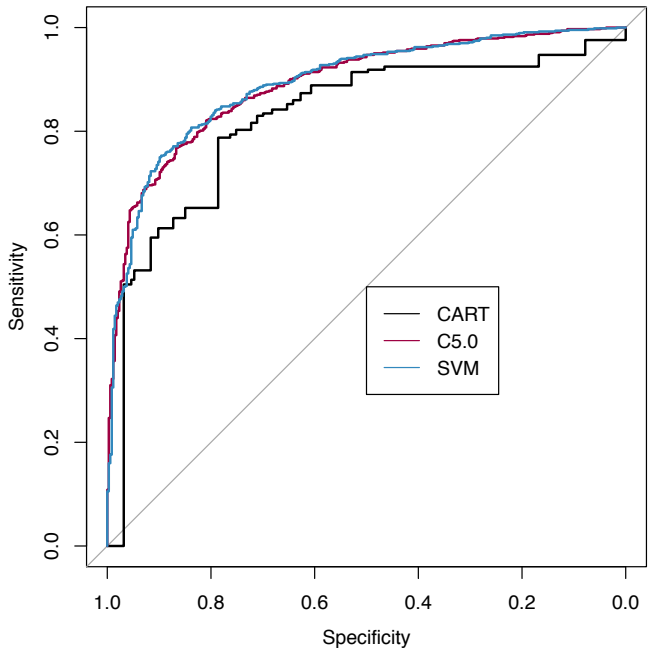


Test Set Results

```
> svmPred <- predict(svmTune, testing[, names(testing) != "Class"])
> confusionMatrix(svmPred, testing$Class)
```

```
Confusion Matrix and Statistics

          Reference
Prediction  PS  WS
        PS 567  85
        WS  97 261

               Accuracy : 0.8198
                 95% CI : (0.7947, 0.843)
    No Information Rate : 0.6574
    P-Value [Acc > NIR] : <2e-16

                  Kappa : 0.6032
 Mcnemars Test P-Value : 0.4149

            Sensitivity : 0.8539
            Specificity : 0.7543
         Pos Pred Value : 0.8696
         Neg Pred Value : 0.7291
             Prevalence : 0.6574
         Detection Rate : 0.5614
   Detection Prevalence : 0.6455

       Positive Class : PS
```


Test Set ROC Curves



Comparing Models

Comparing Models Using Resampling

Notice that, before each call to `train`, we set the random number seed.

That has the effect of using the same resamping data sets for the boosted tree and support vector machine.

Effectively, we have *paired* estimates for performance.

Hothorn *et al* (2005) and Eugster *et al* (2008) demonstrate techniques for making inferential comparisons using resampling.

Collecting Results With `resamples`

`caret` has a function and classes for collating resampling results from objects of class `train`, `rfe` and `sbf`.

```
> cvValues <- resamples(list(CART = rpartTune, SVM = svmTune, C5.0 = c5Tune))
> summary(cvValues)
Call:
summary.resamples(object = cvValues)

Models: CART, SVM, C5.0
Number of resamples: 30

ROC
       Min. 1st Qu. Median   Mean 3rd Qu.   Max. NAs
CART 0.7323  0.8335 0.8484 0.8536  0.8765 0.9234    0
SVM  0.7698  0.8698 0.8847 0.8828  0.9112 0.9301    0
C5.0 0.8136  0.8889 0.9074 0.9004  0.9226 0.9485    0

Sens
       Min. 1st Qu. Median   Mean 3rd Qu.   Max. NAs
CART 0.6825  0.8125 0.8347 0.8312  0.8594 0.9219    0
SVM  0.7778  0.8314 0.8594 0.8611  0.8906 0.9524    0
C5.0 0.7619  0.8413 0.8594 0.8589  0.8854 0.9375    0

Spec
```


Visualizing the Resamples

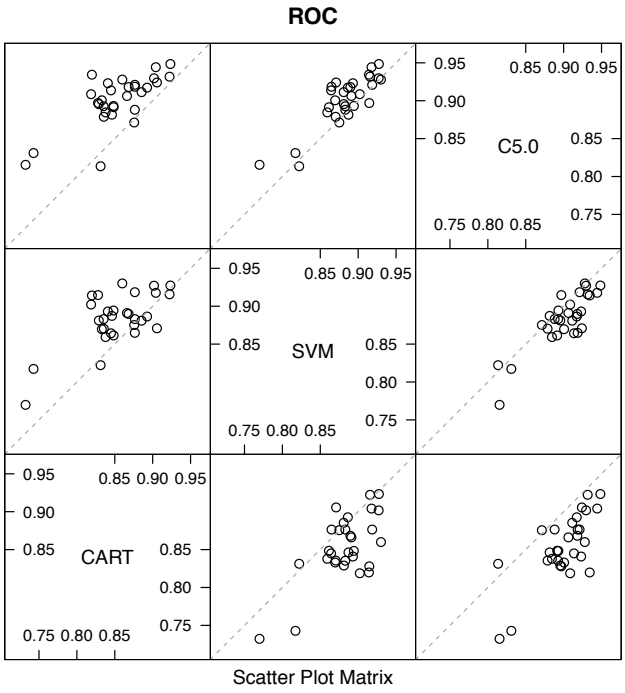
There are a number of `lattice` plot methods to display the results: `bwplot`, `dotplot`, `parallelplot`, `xyplot`, `splom`.

For example:

```
> splom(cvValues, metric = "ROC")
```

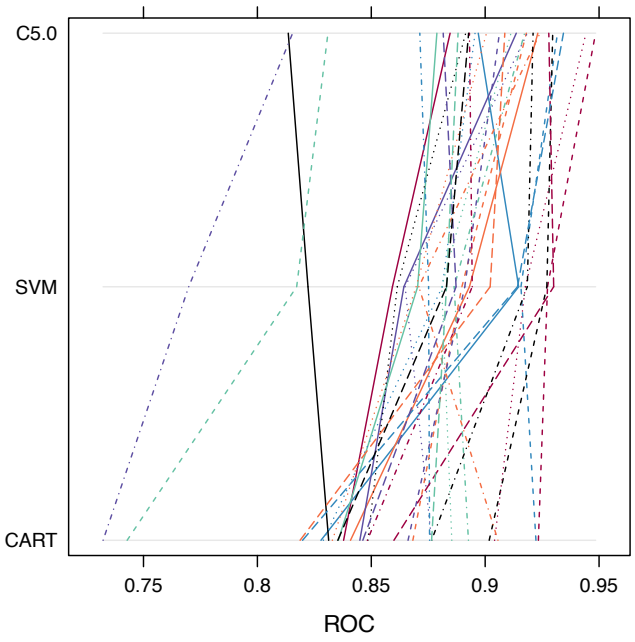
Visualizing the Resamples

```
xyplot(cvValues, metric = "ROC")
```



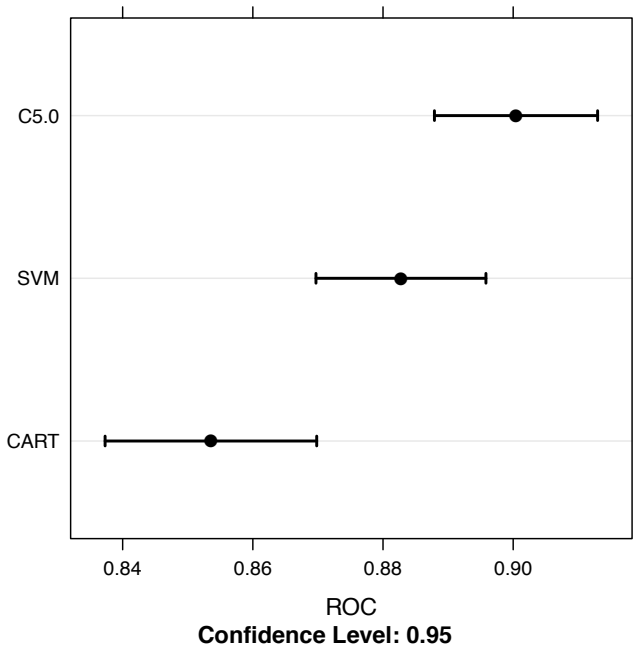
Visualizing the Resamples

```
parallelplot(cvValues, metric = "ROC")
```



Visualizing the Resamples

`dotplot(cvValues, metric = "ROC")`



Comparing Models

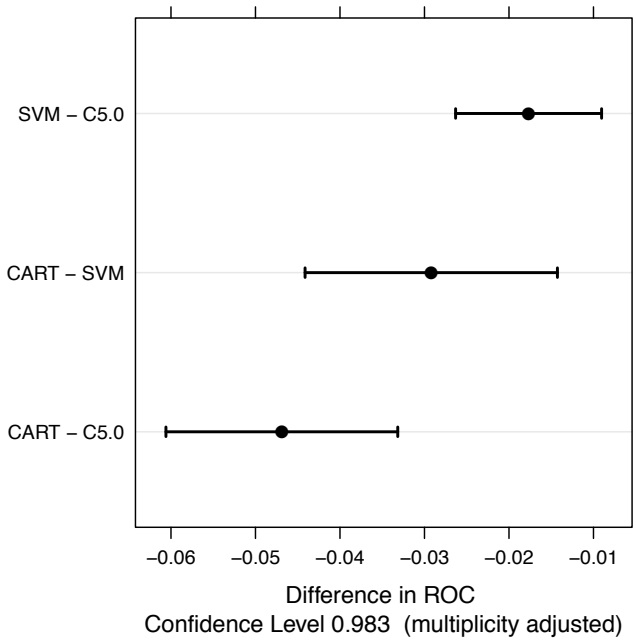
We can also test to see if there are differences between the models:

```
> rocDiffs <- diff(cvValues, metric = "ROC")
> summary(rocDiffs)

Call:
summary.diff.resamples(object = rocDiffs)

p-value adjustment: bonferroni
Upper diagonal: estimates of the difference
Lower diagonal: p-value for H0: difference = 0

ROC
     CART      SVM       C5.0
CART           -0.02920  -0.04688
SVM  8.310e-05           -0.01768
C5.0 4.389e-09 4.359e-05
```

There are `lattice` plot methods, such as `dotplot`.

Visualizing the Differences

```
dotplot(rocDiffs, metric = "ROC")
```



Parallel Processing

Since we are fitting a lot of independent models over different tuning parameters and sampled data sets, there is no reason to do these sequentially.

R has many facilities for splitting computations up onto multiple cores or machines

See Schmidberger *et al* (2009) for a recent review of these methods

foreach and caret

To loop through the models and data sets, caret uses the foreach package, which parallelizes for loops.

foreach has a number of *parallel backends* which allow various technologies to be used in conjunction with the package.

On CRAN, these are the doSomething packages, such as doMC, doMPI, doSMP and others.

For example, doMC uses the multicore package, which forks processes to split computations (for unix and OS X only for now).

foreach and caret

To use parallel processing in caret, no changes are needed when calling train.

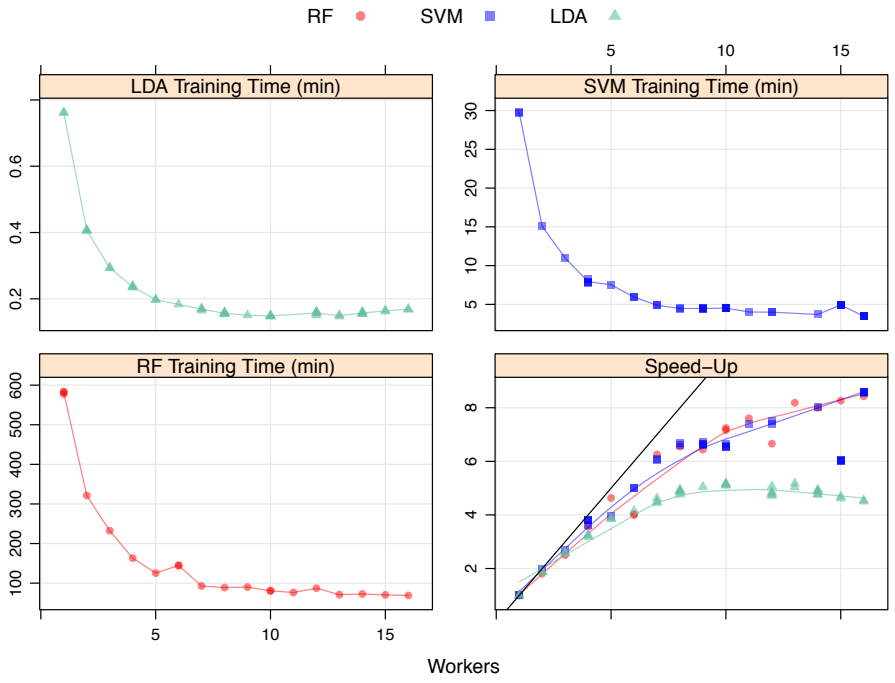
The parallel technology must be *registered* with foreach prior to calling train.

For multicore

```
> library(doMC)
> registerDoMC(cores = 2)
```

Training Times and Speedups

HPC job scheduling data from Kuhn and Johnson (2013) and the `multicore` package:



References

Breiman L, Friedman J, Olshen R, Stone C (1984). *Classification and Regression Trees*. Chapman and Hall, New York.

Breiman, L (2001). "Statistical modeling: The two cultures." *Statistical Science*. 16(3), 199-231.

Eugster M, Hothorn T, Leisch F (2008). "Exploratory and Inferential Analysis of Benchmark Experiments." *Ludwigs-Maximilians-Universitat Munchen, Department of Statistics*, Tech. Rep, 30.

Friedman J (2002). "Stochastic Gradient Boosting." *Computational Statistics and Data Analysis*, 38(4), 367-378.

Hastie T, Tibshirani R, Friedman J (2008). *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. Springer

References

Hill A, LaPan P, Li Y, Haney S (2007). "Impact of Image Segmentation on High-Content Screening Data Quality for SK-BR-3 Cells." *BMC Bioinformatics*, 8(1), 340.

Hothorn T, Leisch F, Zeileis A, Hornik K (2005). "The Design and Analysis of Benchmark Experiments." *Journal of Computational and Graphical Statistics*, 14(3), 675-699.

Kuhn M, Johnson K (2013). *Applied Predictive Modeling*. Springer

Quinlan R (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers.

Schmidberger M *et al* (2009). "State-of-the-art in Parallel Computing with R." *Journal of Statistical Software* 47.1.

Shmueli, G. (2010). "To explain or to predict?." *Statistical Science*, 25(3), 289-310.

Versions

- R version 3.0.0 (2013-04-03), `x86_64-apple-darwin10.8.0`
- Locale: `en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, graphics, grDevices, grid, methods, parallel, splines, stats, tools, utils
- Other packages: AppliedPredictiveModeling 1.1-001, C50 0.1.0-15, caret 5.16-06, class 7.3-7, cluster 1.14.4, codetools 0.2-8, CORElearn 0.9.41, ctv 0.7-8, digest 0.6.3, doMC 1.3.0, e1071 1.6-1, foreach 1.4.1, Hmisc 3.10-1, iterators 1.0.6, kernlab 0.9-18, lattice 0.20-15, MASS 7.3-26, mlbench 2.1-1, partykit 0.1-5, plyr 1.8, pROC 1.5.4, reshape2 1.2.2, rpart 4.1-1, survival 2.37-4, svmpath 0.953, weaver 1.26.0
- Loaded via a namespace (and not attached): compiler 3.0.0, stringr 0.6.2