Predictive Modeling in R

Max Kuhn
RStudio

IFCS

August 7, 2017

Outline

The slides and code for this presentation are in the github repository https://github.com/topepo/IFCS-2017.

The following R packages are used (and their dependencies): caret (version 6.0-71 or greater), glmnet, nnet, ipred, recipes, and rpart.

Statistical Issues with Class Imbalances

Reid (2015):

> *"Coyotes, bobcats and gray foxes are all common mammalian mesopredators in coastal California and are found sympatrically in much of North America. Scats produced by these three animals are quite similar, but have historically been differentiated largely by morphology. I tested the efficacy of morphological classification of scat to species by building predictive models for species identification with a set of well-described, DNA-verified scats."*

Reid, R. E. B. (2015). A morphometric modeling approach to distinguishing among bobcat, coyote and gray fox scats. *Wildlife Biology*, 21(5), 254–262, http://www.bioone.org/doi/10.2981/wlb.00105

Load the Data

```
> library(caret)
> data(scat)
> str(scat)
```

```
'data.frame': 110 obs. of  19 variables:
 $ Species  : Factor w/ 3 levels "bobcat","coyote",..: 2 2 1 2 2 2 1 1 1 1 ...
 $ Month    : Factor w/ 9 levels "April","August",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ Year     : int  2012 2012 2012 2012 2012 2012 2012 2012 2012 2012 ...
 $ Site     : Factor w/ 2 levels "ANNU","YOLA": 2 2 2 2 2 2 1 1 1 1 ...
 $ Location : Factor w/ 3 levels "edge","middle",..: 1 1 2 2 1 1 3 3 3 2 ...
 $ Age      : int  5 3 3 5 5 5 1 3 5 5 ...
 $ Number   : int  2 2 2 2 4 3 5 7 2 1 ...
 $ Length   : num  9.5 14 9 8.5 8 9 6 5.5 11 20.5 ...
 $ Diameter : num  25.7 25.4 18.8 18.1 20.7 21.2 15.7 21.9 17.5 18 ...
 $ Taper    : num  41.9 37.1 16.5 24.7 20.1 28.5 8.2 19.3 29.1 21.4 ...
 $ TI       : num  1.63 1.46 0.88 1.36 0.97 1.34 0.52 0.88 1.66 1.19 ...
 $ Mass     : num  15.9 17.6 8.4 7.4 25.4 ...
 $ d13C     : num  -26.9 -29.6 -28.7 -20.1 -23.2 ...
 $ d15N     : num  6.94 9.87 8.52 5.79 7.01 8.28 4.2 3.89 7.34 6.06 ...
 $ CN       : num  8.5 11.3 8.1 11.5 10.6 9 5.4 5.6 5.8 7.7 ...
 $ ropey    : int  0 0 1 1 0 1 1 0 0 1 ...
 $ segmented: int  0 0 1 0 1 0 1 1 1 1 ...
 $ flat     : int  0 0 0 0 0 0 0 0 0 0 ...
 $ scrape   : int  0 0 1 0 0 0 1 0 0 0 ...
```

Some Data Are Missing

```
> pct_nonmissing <- function(x) mean(!is.na(x))
> unlist(lapply(scat, pct_nonmissing))
```

```
  Species     Month      Year      Site  Location       Age    Number    Length
1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000
 Diameter     Taper        TI      Mass      d13C      d15N        CN     ropey
0.9454545 0.8454545 0.8454545 0.9909091 0.9818182 0.9818182 0.9818182 1.0000000
segmented      flat    scrape
1.0000000 1.0000000 1.0000000
```

Before Exploring the Data

There is no a huge amount of data here and one of our main concerns should be biasing ourselves. For example, we don't what to look at the data and then create models to fit our expectations.

A better approach is to randomly holdback some data (a *test set*) to evaluate trends that we see in the remaining data (called the *training set*).

Let's hold out 25% of the data back for testing. We can do a stratified random split of the data within the species so that we preserve the frequencies of each animal.

caret has a function called `createDataPartition` that will do this.

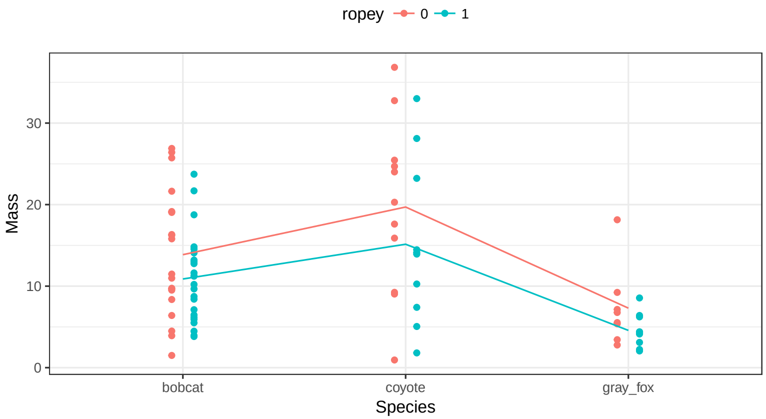
Split the Data

```
> set.seed(11218)
> in_train <- createDataPartition(scat$Species, p = 3/4, list = FALSE)
> head(in_train)

     Resample1
[1,]         1
[2,]         2
[3,]         3
[4,]         4
[5,]         5
[6,]         6

> train_data <- scat[ in_train,]
> test_data  <- scat[-in_train,]
> ## It isn't much data but it's better than nothing...
> table(test_data$Species)


  bobcat    coyote gray_fox
      14         7        6
```

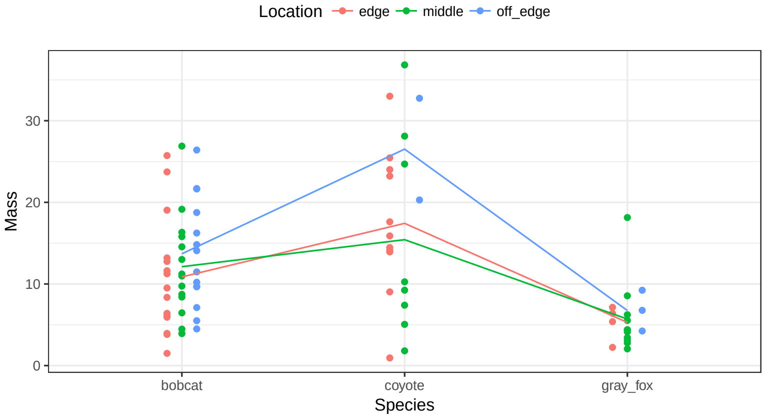
Interaction Plot Code

```
> int_plot <- function(dat, y, x, group) {
+   library(plyr)
+   if(!is.factor(dat[,group])) dat[,group] <- factor(dat[,group])
+   means <- ddply(dat, c(y, group),
+                  function(obj) c(mean = mean(obj[,x], na.rm = TRUE)))
+   ggplot(dat,
+          aes_string(y = x,  x = y, color = group, group = group)) +
+     geom_point(position = position_dodge(width = 0.2)) +
+     geom_line(data = means, aes_string(y = "mean")) +
+     theme(legend.position = "top")
+ }
```

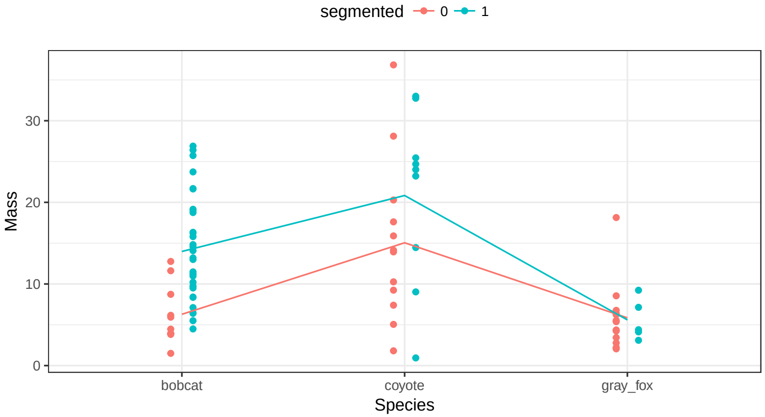
Investigate Differences in Species×Mass×Morphology



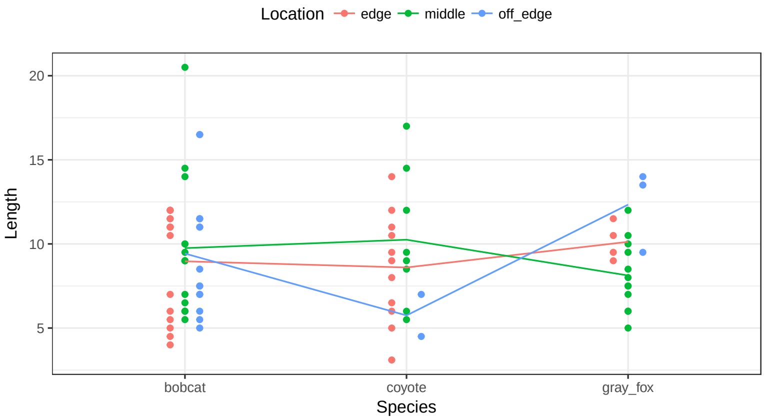
Species × Mass × Location



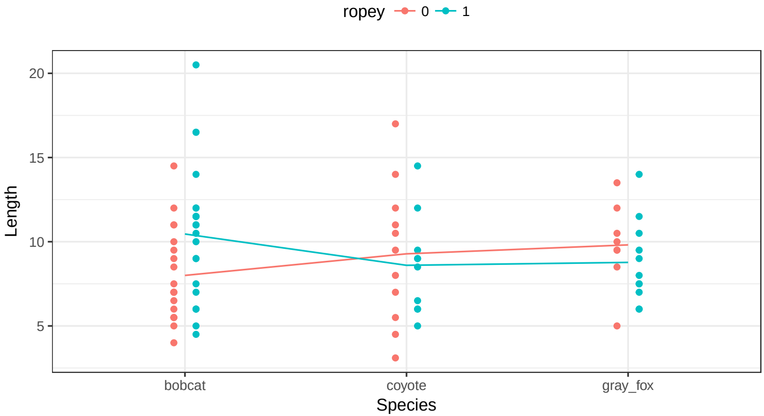
Species × Mass × Morphology



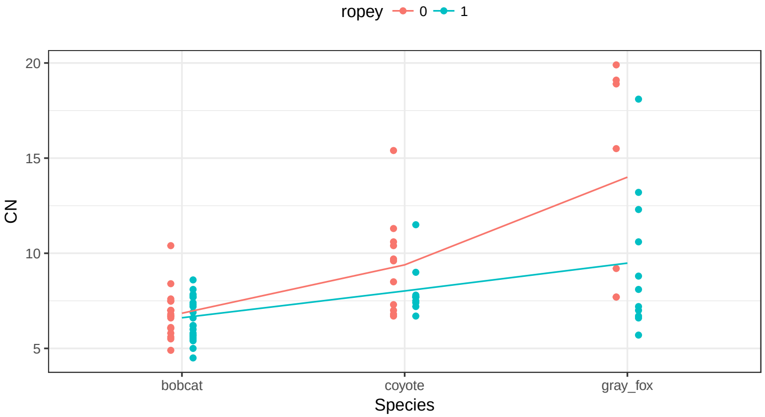
Species × Length × Location



Species × Length × Morphology



Species×C/N Ratio×Morphology



Dummy Variables

A few of the potential predictors are categorical in nature. *Most* models require numeric representations of the data.

When a predictor has C possible values, a common approach is to create $C-1$ binary dummy variables to use in the model. For example, `Location` has three levels:

	Dummy Variable Columns		
Data Value	`edge`	`middle`	`off_edge`
`"edge"`	1	0	0
`"middle"`	0	1	0
`"off_edge"`	0	0	1

For *ordered* categorical predictors, the default encoding is more complex. See "The Basics of Encoding Categorical Data for Predictive Models" at `http://bit.ly/1CtXg0x`

Dummy Variables and Model Functions

The primary convention in R is to convert factors to dummy variables when a model uses the formula interface (examples later).

However, this is not always the case. Many models using trees or rules (e.g. `rpart`, `C5.0`, `randomForest`, etc):

- do not require numeric representations of the predictors
- do not create dummy variables

Other notable exceptions are naive Bayes models and support vector machines using string kernel functions.

The Formula Interface

There are two main conventions for specifying models in **R**[1]: 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(Species ~ Location + Mass + Number,
              data = train_data)
```

[1]There is a third, *new* method called recipes shown at the end.

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.

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.

NOTE: Many functions do classification or regression on the basis of the outcome class (e.g. factor or numeric)

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 = train_data[, -1],
              ## Species is the first column
              y = train_data$Species)
```

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.

An Initial Model

Let's fit an initial *small* model to the predictors that do not have missing data.

We'll use the formula method so that we don't worry about factors predictors and eliminate one data point that has a missing values for `Mass`.

```
> small_form <- paste("Species ~ Month + Year + Site + Age +",
+                     "Number + Length*ropey + (Location + segmented)*Mass + ",
+                     "flat + scrape")
> small_form <- as.formula(small_form)
>
> small_tr_dat <- train_data[, all.vars(small_form)]
> small_tr_dat <- small_tr_dat[complete.cases(small_tr_dat),]
```

We will fit a multinomial regression model where each class is a slope and intercept on the log–odds scale

$$logit(\pi_j) = \beta_0 + \beta_1 x_1 + \ldots + \beta_p x_p$$

Measuring Performance

What metric should we use to tell if the model is predictive?

Overall accuracy can be used, but this may be problematic since 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

We could also use the multinomial log–likelihood but this isn't really connected to predictive performance in the way that accuracy and Kappa are. I'll use Kappa.

Resampling

Building data model and re–predicting the same samples can result in highly optimistic estimates of performance.

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.

See the two blog posts "Comparing Different Species of Cross-Validation" at `http://bit.ly/1yE0Ss5` and `http://bit.ly/1zfoFj2`

V–Fold Cross–Validation

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

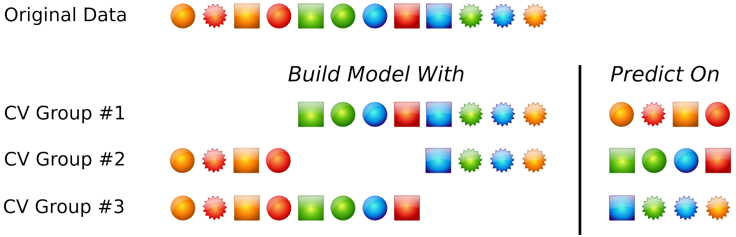
1. We leave out the first block of data and fit a model.
2. This model is used to predict the held-out block
3. We continue this process until we've predicted all V held–out blocks

The final performance is based on the hold-out predictions

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

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

V–Fold Cross–Validation



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)

```
> ## only one way here:
> rpart(y ~ ., data = dat)
>
> ## and both ways here:
> lda(y ~ ., data = dat)
>
> lda(x = predictors, y = outcome)
```

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)`

The caret Package

The caret package was developed to:

- create a unified interface for modeling and prediction (interfaces to 232 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: http://topepo.github.io/caret/

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

Model List: http://topepo.github.io/caret/bytag.html

Many computing sections in APM

Multinomial Model

We will use caret's interface to the multinom function in the nnet package.

We'll use the formula method so that we don't worry about factors predictors and eliminate one data point that has a missing values for Mass.

```
> ctrl <- trainControl(method = "repeatedcv", repeats = 5, classProbs = TRUE)
> set.seed(2592) ## locks in the resamples
> mnr_tune <- train(small_form, data = small_tr_dat,
+                   method = "multinom",
+                   preProc = c("center", "scale"),
+                   ## avoid regularization for now
+                   tuneGrid = data.frame(decay = 0),
+                   trControl = ctrl,
+                   ## this next argument is passed to `multinom`
+                   trace = FALSE)
```

Multinomial Model

```
> mnr_tune

Penalized Multinomial Regression

82 samples
12 predictors
 3 classes: 'bobcat', 'coyote', 'gray_fox'

Pre-processing: centered (24), scaled (24)
Resampling: Cross-Validated (10 fold, repeated 5 times)
Summary of sample sizes: 74, 74, 74, 73, 73, 73, ...
Resampling results:

  Accuracy   Kappa
  0.5210794  0.2089317

Tuning parameter 'decay' was held constant at a value of 0

> predict(mnr_tune, head(test_data)) ## or type = "prob"

[1] gray_fox gray_fox bobcat   gray_fox bobcat   bobcat
Levels: bobcat coyote gray_fox
```

Variable Importance

```
> print(varImp(mnr_tune, scale = FALSE), top = 10)

multinom variable importance

  only 10 most important variables shown (out of 24)

                        Overall
`Length:ropey`            28.87
`Locationmiddle:Mass`     28.85
`segmented:Mass`          25.36
ropey                     21.06
`Locationoff_edge:Mass`   18.81
Locationoff_edge          18.12
SiteYOLA                  17.53
Mass                      16.81
Locationmiddle            13.89
scrape                    12.92
```

Those Missing Data

We have ignored the predictors with missing data so far.

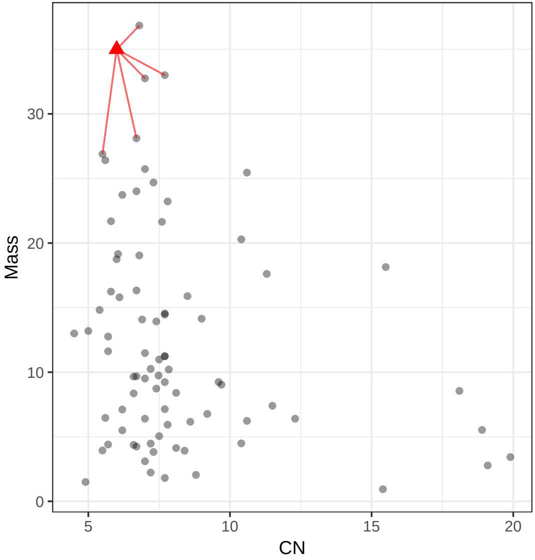
As a pre–processing method, we will use an imputation method to fill in their data prior to modeling.

It is crucial that we do this inside of every resample so that the performance estimates account for the variation generated by the imputation method.

There are several methods for imputing the data. We will use a 5–nearest neighbor model *for imputing* each of the predictors that had missing data.

Suppose a model contained three terms (`Mass`, `CN`, and `Length`). If `Length` were missing, we would find the most similar scats in the other two dimensions and use these to impute with their average `Length`.

K–Nearest Neighbors Imputation



Multinomial Model – All Data

```
> full_form <- paste("Species ~ Month + Year + Site + Age + Number +",
+                    "Length*ropey + (Location + segmented)*Mass +",
+                    "flat + scrape +",
+                    "TI + d13C + d15N + CN + Diameter + Taper")
> full_form <- as.formula(full_form)
> set.seed(2592)
> mnr_impute <- train(full_form, data = train_data,
+                     method = "multinom",
+                     ## Add imputation to the list of pre-processing steps
+                     preProc = c("center", "scale", "knnImpute", "zv"),
+                     tuneGrid = data.frame(decay = 0),
+                     trControl = ctrl,
+                     ## do not remove missing data before modeling
+                     na.action = na.pass,
+                     trace = FALSE)
```

Multinomial Model – All Data

```
> mnr_impute

Penalized Multinomial Regression

70 samples
18 predictors
 3 classes: 'bobcat', 'coyote', 'gray_fox'

Pre-processing: centered (30), scaled (30), nearest neighbor imputation (30)
Resampling: Cross-Validated (10 fold, repeated 5 times)
Summary of sample sizes: 75, 75, 75, 74, 74, 74, ...
Resampling results:

  Accuracy   Kappa
  0.6358968  0.4144155

Tuning parameter 'decay' was held constant at a value of 0
```

Variable Importance

```
> print(varImp(mnr_impute, scale = FALSE), top = 10)
multinom variable importance

  only 10 most important variables shown (out of 30)

               Overall
`Length:ropey`  161.97
ropey           150.91
CN              119.29
flat             84.24
Locationmiddle   74.31
segmented        62.37
d15N             60.34
d13C             60.08
Length           56.90
SiteYOLA         56.26
```

Resampled Confusion Matrix

```
> confusionMatrix(mnr_tune)

Cross-Validated (10 fold, repeated 5 times) Confusion Matrix

(entries are percentual average cell counts across resamples)

          Reference
Prediction bobcat coyote gray_fox
  bobcat     32.0   11.0      9.3
  coyote     11.2   10.7      3.2
  gray_fox    9.3    3.9      9.5

 Accuracy (average) : 0.522
```

Model Tuning

Now suppose we want to see if regularizing the regression coefficients will result in better fits

The glmnet package can be used to build a similar model using L_1 or L_2 regularization (or a mixture of the two).

- an L_1 penalty can have the effect of setting coefficients to zero
- L_2 regularization is basically ridge regression where the magnitude of the coefficients are dampened to avoid overfitting

For a glmnet model, we need to determine the total amount regularization (called `lambda`) and the mixture of L_1 and L_2 (called `alpha`).

`alpha`$= 1$ is a lasso model while `alpha`$= 0$ is ridge regression (aka weight decay).

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

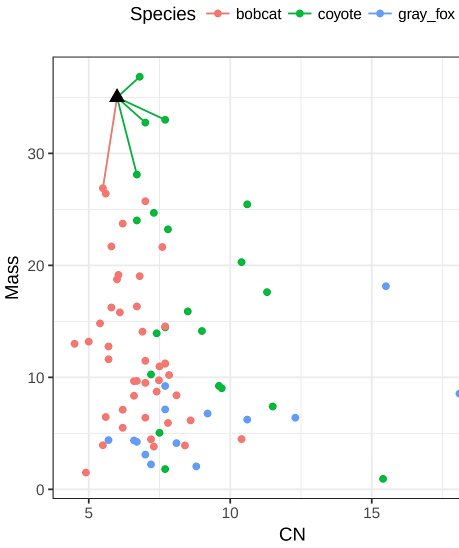
- neighborhood size in nearest neighbor models is an example
- the number if splits in a tree model

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.

A new point is to be predicted. A 5–nearest neighbor model is illustrated.

Example: K–Nearest Neighbors *Classification*



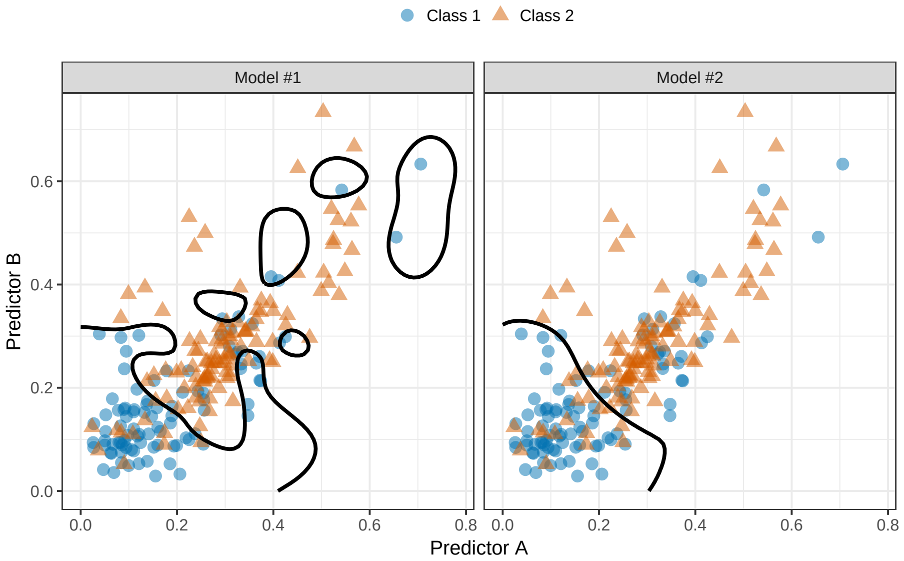
Over–Fitting

On the next slide, two classification boundaries are shown for the a different model type not yet discussed.

The difference in the two panels is solely due to different choices in tuning parameters.

One over–fits the training data.

Two Model Fits



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 ;
        Pre–process the data and 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;
```

Model Tuning

train can incorporate the model tuning and new pre–processing techniques. For model tuning, there are two interfaces

- Let train derive a grid of points to test using the tuneLength argument
- Use the tuneGrid argument to dictate the exact set of candidate models to evaluate during resampling. The column names should match the tuning parameters.

```
> glmn_grid <- expand.grid(alpha = c(0.05, seq(.1, 1, by = 0.025)),
+                          lambda = c(.001, .01, .1))
> nrow(glmn_grid)

[1] 114
```

To be clear: we are evaluating $114 \times 50 = 5700$ models just to determine the values of alpha and lambda.

Model #5701 is on the entire training set.

Model Tuning

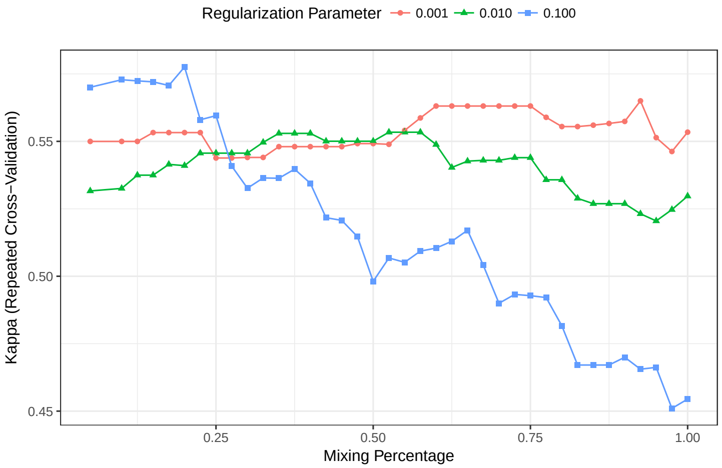
```
> set.seed(2592) ## use the same resamples as mnr_impute
> glmn_tune <- train(full_form, data = train_data,
+                    method = "glmnet",
+                    preProc = c("center", "scale", "knnImpute", "zv"),
+                    ## pass in the tuning grid
+                    tuneGrid = glmn_grid,
+                    ## pick the sub-model with the best kappa value
+                    metric = "Kappa",
+                    na.action = na.pass,
+                    trControl = ctrl)
> ## best sub-model results:
> glmn_tune$bestTune
```

```
   alpha lambda
18   0.2    0.1
```

```
> getTrainPerf(glmn_tune)
```

```
  TrainAccuracy TrainKappa method
1     0.7558413  0.5775722 glmnet
```

glmnet Profile



Resampled Confusion Matrix

```
> confusionMatrix(glmn_tune)
Cross-Validated (10 fold, repeated 5 times) Confusion Matrix

(entries are percentual average cell counts across resamples)

          Reference
Prediction bobcat coyote gray_fox
  bobcat     46.3    4.6     11.8
  coyote      2.4   19.3      1.0
  gray_fox    3.1    1.4     10.1

 Accuracy (average) : 0.7566
```

Model Comparison

Since the `glmnet` and second `multinom` fits used the same training set and resamples, we get a set of 50 *paired* comparisons in Kappa. Here are 15 examples:

```
             multinom glmnet Difference
Fold01 Rep1     0.304  0.294     -0.010
Fold01 Rep2     0.600  0.789      0.189
Fold01 Rep3     0.250  0.308      0.058
Fold01 Rep4     0.413  0.786      0.373
Fold01 Rep5     0.368  0.556      0.187
Fold02 Rep1     0.742  0.407     -0.335
Fold02 Rep2     0.158  0.200      0.042
Fold02 Rep3     0.471  0.800      0.329
Fold02 Rep4     0.048  0.600      0.552
Fold02 Rep5     0.400  0.800      0.400
Fold03 Rep1    -0.200  0.368      0.568
Fold03 Rep2     0.304  0.579      0.275
Fold03 Rep3     0.500  0.647      0.147
Fold03 Rep4     0.273  0.789      0.517
Fold03 Rep5     0.810  0.800     -0.010
```

Model Comparison

We can use these to formally test if the model fit improved:

```
> compare_models(glmn_tune, mnr_impute, metric = "Kappa")


One Sample t-test

data:  x
t = 4.1288, df = 49, p-value = 0.0001414
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 0.0837440 0.2425694
sample estimates:
mean of x
0.1631567
```

The `resamples` function can be used to compare and visualize performance from many different models with the same resamples.

Fitting Other Models

The value of `train` is to be able to fit different models without learning the syntactical minutiae for each modeling package.

For example, to fit a bagged CART tree:

```
> set.seed(2592)
> bagged_tree <- train(Species ~ ., data = train_data,
+                      method = "treebag",
+                      metric = "Kappa",
+                      na.action = na.pass,
+                      trControl = ctrl)
> getTrainPerf(bagged_tree)

  TrainAccuracy TrainKappa  method
1     0.7000873  0.4992813 treebag
```

Fitting Other Models

. . . or a K-nearest neighbor model:

```
> set.seed(2592)
> knn_tune <- train(Species ~ ., data = train_data,
+                   method = "knn",
+                   preProc = c("center", "scale", "knnImpute", "zv"),
+                   ## pass in the tuning grid _size_
+                   tuneLength = 20,
+                   metric = "Kappa",
+                   na.action = na.pass,
+                   trControl = ctrl)
> getTrainPerf(knn_tune)

  TrainAccuracy TrainKappa method
1     0.6860635  0.4617916    knn
```

Preprocessing Too!

. . . using different preprocessing the predictors:

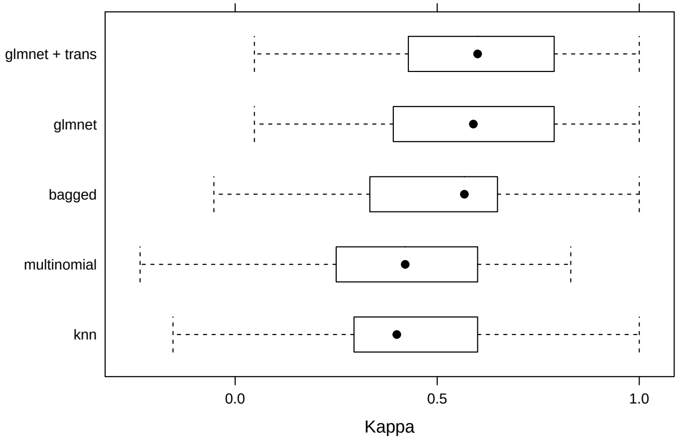
```
> set.seed(2592)
> transformed <- train(full_form, data = train_data,
+                      method = "glmnet",
+                      ## Also transform the predictors
+                      preProc = c("center", "scale", "knnImpute",
+                                  "zv", "YeoJohnson"),
+                      tuneGrid = glmn_grid,
+                      metric = "Kappa",
+                      na.action = na.pass,
+                      trControl = ctrl)
> getTrainPerf(transformed)

  TrainAccuracy TrainKappa method
1     0.7700873   0.609226 glmnet
```

Collecting the Results

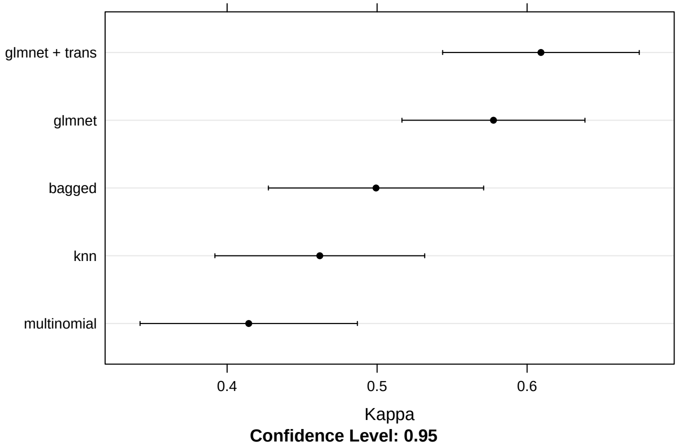
As previously mentioned, the `resamples` function can be used to compare multiple models

```
> rs <- resamples(list(knn = knn_tune, bagged = bagged_tree,
+                      multinomial = mnr_impute, glmnet = glmn_tune,
+                      "glmnet + trans" = transformed))
> summary(rs, metric = "Kappa")

Call:
summary.resamples(object = rs, metric = "Kappa")

Models: knn, bagged, multinomial, glmnet, glmnet + trans
Number of resamples: 50

Kappa
                   Min. 1st Qu. Median   Mean 3rd Qu.   Max. NA's
knn            -0.15380  0.2975 0.4000 0.4618  0.6000 1.0000    0
bagged         -0.05263  0.3364 0.5673 0.4993  0.6486 1.0000    0
multinomial    -0.23530  0.2557 0.4208 0.4144  0.6000 0.8305    0
glmnet          0.04762  0.3953 0.5895 0.5776  0.7885 1.0000    0
glmnet + trans  0.04762  0.4603 0.6000 0.6092  0.7895 1.0000    0
```

Resampling Distributions



Resampling Distributions



Test Set Results

Let's predict the test set

```
> test_pred <- predict(glmn_tune, newdata = test_data, na.action = na.pass)
> str(test_pred)

 Factor w/ 3 levels "bobcat","coyote",..: 3 3 1 3 1 1 1 1 1 3 ...

> test_prob <- predict(glmn_tune, newdata = test_data,
+                      na.action = na.pass, type = "prob")
> str(test_prob)

'data.frame': 27 obs. of  3 variables:
 $ bobcat  : num  0.0133 0.0193 0.8056 0.3918 0.7465 ...
 $ coyote  : num  0.0883 0.1928 0.0767 0.0881 0.0568 ...
 $ gray_fox: num  0.898 0.788 0.118 0.52 0.197 ...
```

Test Set Results

```
> confusionMatrix(test_pred, test_data$Species)

Confusion Matrix and Statistics

          Reference
Prediction bobcat coyote gray_fox
  bobcat       14      4        1
  coyote        0      3        0
  gray_fox      0      0        5

Overall Statistics

               Accuracy : 0.8148
                 95% CI : (0.6192, 0.937)
    No Information Rate : 0.5185
    P-Value [Acc > NIR] : 0.001421

                  Kappa : 0.6723
 Mcnemar's Test P-Value : NA

Statistics by Class:

                     Class: bobcat Class: coyote Class: gray_fox
Sensitivity                 1.0000        0.4286          0.8333
Specificity                 0.6154        1.0000          1.0000
Pos Pred Value              0.7368        1.0000          1.0000
Neg Pred Value              1.0000        0.8333          0.9545
Prevalence                  0.5185        0.2593          0.2222
Detection Rate              0.5185        0.1111          0.1852
Detection Prevalence        0.7037        0.1111          0.1852
Balanced Accuracy           0.8077        0.7143          0.9167
```

Recipes

Recipes are an alternate method for:

- specifying variables for a model
- the *roles* of each variable
- a sequence of preprocessing or computational steps executed before modeling

We can approach the design matrix and preprocessing steps by first specifying a *sequence of steps*.

A recipe is a specification of *intent*.

One issue with the formula method is that it couples the specification for your predictors along with the implementation.

Recipes, as you'll see, separates the *planning* from the *doing*.

Sequentially Creating a Recipe

Previously, we had a model formula:

```
> paste("Species ~ Month + Year + Site + Age +",
+       "Number + Length*ropey + (Location + segmented)*Mass + ",
+       "flat + scrape")
```

To create a recipe, we first specify the variables and their roles. The easiest method is a simple formula:

```
> library(recipes)
> scat_rec <- recipe(Species ~ ., data = scat)
> scat_rec
Data Recipe

Inputs:

      role #variables
   outcome          1
 predictor         18
```

Imputation

The next action is to setup how we will impute missing variables

```
> scat_rec <- scat_rec %>%
+   step_bagimpute(Diameter, Taper, TI, Mass, d13C, d15N, CN)
```

Note that this delays execution and the dplyr-like variable specification.

Now let's setup the dummy variables using general selectors:

```
> scat_rec <- scat_rec %>%
+   step_dummy(all_nominal(), -all_outcomes())
> scat_rec
```

```
Data Recipe

Inputs:

      role #variables
   outcome          1
 predictor         18

Steps:

Bagged tree imputation for Diameter, Taper, TI, Mass, d13C, d15N, CN
Dummy variables from all_nominal(), -all_outcomes()
```

Interactions

The one step that does not use the dplyr convention is for making interactions:

```
> scat_rec <- scat_rec %>%
+   step_interact(~ Length:ropey) %>%
+   step_interact(~ Location_middle:Mass) %>%
+   step_interact(~ Location_off_edge:Mass) %>%
+   step_interact(~ segmented:Mass)
```

Estimating the Required Statistics

The prep function applies the required computations for each step using a training set:

```
> scat_rec_trained <- prep(scat_rec, training = train_data, retain = TRUE)
```

```
step 1 bagimpute training
step 2 dummy training
step 3 interact training
step 4 interact training
step 5 interact training
step 6 interact training
```

Estimating the Required Statistics

Now we have populated the general selectors:

```
> scat_rec_trained

Data Recipe

Inputs:

      role #variables
   outcome          1
 predictor         18

Training data contained 83 data points and 13 incomplete rows.

Steps:

Bagged tree imputation for Diameter, Taper, TI, Mass, d13C, d15N, CN [trained]
Dummy variables from ~Month, ~Site, ~Location [trained]
Interactions with Length:ropey [trained]
Interactions with Location_middle:Mass [trained]
Interactions with Location_off_edge:Mass [trained]
Interactions with segmented:Mass [trained]
```

Processing Data

To *apply* these computations to new data sets, we can use the bake function.

```
> proc_train_data <- bake(scat_rec_trained, newdata = train_data)
> proc_test_data  <- bake(scat_rec_trained, newdata = test_data)
>
> mean(!complete.cases(train_data))

[1] 0.1566265

> mean(!complete.cases(proc_train_data))

[1] 0

> names(proc_train_data)

 [1] "Year"                         "Age"
 [3] "Number"                       "Length"
 [5] "Diameter"                     "Taper"
 [7] "TI"                           "Mass"
 [9] "d13C"                         "d15N"
[11] "CN"                           "ropey"
[13] "segmented"                    "flat"
[15] "scrape"                       "Month_August"
[17] "Month_February"               "Month_January"
[19] "Month_June"                   "Month_May"
[21] "Month_November"               "Month_October"
```

How Are Recipes used with Models?

Right now, the development version of caret has a recipe interface. For example:

```
> train(scat_rec,
+       data = train_data,
+       method = "glmnet",
+       tuneGrid = glmn_grid,
+       metric = "Kappa",
+       na.action = na.pass,
+       trControl = ctrl)
```

More packages are coming with other interfaces to models

Session Info (pt1)

```
setting  value
version  R version 3.3.3 (2017-03-06)
os       macOS Sierra 10.12.6
system   x86_64, darwin13.4.0
ui       X11
language (EN)
collate  en_US.UTF-8
tz       America/New_York
date     2017-07-30

package                    * version   date       source
abind                        1.4-5     2016-07-21 CRAN (R 3.3.0)
acepack                      1.4.1     2016-10-29 CRAN (R 3.3.0)
AppliedPredictiveModeling  * 1.1-6     2014-07-25 CRAN (R 3.3.0)
arules                       1.5-2     2017-03-13 CRAN (R 3.3.2)
assertthat                   0.2.0     2017-04-11 CRAN (R 3.3.2)
backports                    1.0.5     2017-01-18 CRAN (R 3.3.2)
base64enc                    0.1-3     2015-07-28 CRAN (R 3.3.0)
bindr                        0.1       2016-11-13 CRAN (R 3.3.2)
bindrcpp                     0.2       2017-06-17 cran (@0.2)
bitops                       1.0-6     2013-08-17 CRAN (R 3.3.0)
C50                        * 0.1.0-24  2015-03-09 CRAN (R 3.3.0)
car                          2.1-5     2017-07-04 cran (@2.1-5)
caret                      * 6.0-76    2017-04-18 CRAN (R 3.3.2)
caTools                      1.17.1    2014-09-10 CRAN (R 3.3.0)
checkmate                    1.8.2     2016-11-02 CRAN (R 3.3.0)
class                        7.3-14    2015-08-30 CRAN (R 3.3.3)
clisymbols                   1.2.0     2017-05-21 CRAN (R 3.3.2)
cluster                      2.0.5     2016-10-08 CRAN (R 3.3.3)
codetools                    0.2-15    2016-10-05 CRAN (R 3.3.3)
colorspace                   1.3-2     2016-12-14 CRAN (R 3.3.2)
```

Session Info (pt2)

```
package    * version date       source
CORElearn    1.50.3  2017-03-28 CRAN (R 3.3.2)
CVST         0.2-1   2013-12-10 CRAN (R 3.3.0)
data.table   1.10.4  2017-02-01 CRAN (R 3.3.3)
ddalpha      1.2.1   2016-10-10 CRAN (R 3.3.0)
DEoptimR     1.0-8   2016-11-19 CRAN (R 3.3.2)
digest       0.6.12  2017-01-27 CRAN (R 3.3.2)
dimRed       0.1.0   2017-05-04 CRAN (R 3.3.2)
DMwR       * 0.4.1   2013-08-08 CRAN (R 3.3.0)
doMC       * 1.3.4   2015-10-13 CRAN (R 3.3.0)
dplyr      * 0.7.2   2017-07-20 cran (@0.7.2)
DRR          0.0.2   2016-09-15 CRAN (R 3.3.0)
e1071      * 1.6-8   2017-02-02 CRAN (R 3.3.2)
evaluate     0.10    2016-10-11 CRAN (R 3.3.0)
foreach    * 1.4.3   2015-10-13 CRAN (R 3.3.0)
foreign      0.8-67  2016-09-13 CRAN (R 3.3.3)
Formula    * 1.2-1   2015-04-07 CRAN (R 3.3.0)
gbm          2.1.3   2017-03-21 CRAN (R 3.3.2)
gdata        2.17.0  2015-07-04 CRAN (R 3.3.0)
ggplot2    * 2.2.1   2016-12-30 CRAN (R 3.3.2)
ggthemes   * 3.4.0   2017-02-19 CRAN (R 3.3.3)
glmnet     * 2.0-10  2017-05-06 CRAN (R 3.3.2)
glue         1.1.1   2017-06-21 CRAN (R 3.3.2)
gower        0.1.2   2017-02-23 CRAN (R 3.3.2)
gplots       3.0.1   2016-03-30 CRAN (R 3.3.0)
gridExtra    2.2.1   2016-02-29 CRAN (R 3.3.3)
gtable       0.2.0   2016-02-26 CRAN (R 3.3.0)
gtools       3.5.0   2015-05-29 CRAN (R 3.3.0)
highr        0.6     2016-05-09 CRAN (R 3.3.0)
Hmisc      * 4.0-3   2017-05-02 CRAN (R 3.3.2)
htmlTable    1.9     2017-01-26 CRAN (R 3.3.2)
```

Session Info (pt3)

```
package      * version date       source
htmltools      0.3.6   2017-04-28 CRAN (R 3.3.2)
htmlwidgets    0.8     2016-11-09 CRAN (R 3.3.2)
inTrees      * 1.1     2014-07-25 CRAN (R 3.3.0)
ipred        * 0.9-6   2017-03-01 cran (@0.9-6)
iterators    * 1.0.8   2015-10-13 CRAN (R 3.3.0)
kernlab      * 0.9-25  2016-10-03 CRAN (R 3.3.0)
KernSmooth     2.23-15 2015-06-29 CRAN (R 3.3.3)
knitr        * 1.16    2017-05-18 CRAN (R 3.3.3)
labeling       0.3     2014-08-23 CRAN (R 3.3.0)
lattice      * 0.20-35 2017-03-25 CRAN (R 3.3.3)
latticeExtra   0.6-28  2016-02-09 CRAN (R 3.3.3)
lava           1.5     2017-03-16 cran (@1.5)
lazyeval       0.2.0   2016-06-12 CRAN (R 3.3.0)
lme4           1.1-13  2017-04-19 CRAN (R 3.3.2)
lmtest         0.9-35  2017-02-11 CRAN (R 3.3.2)
lubridate      1.6.0   2016-09-13 CRAN (R 3.3.0)
magrittr       1.5     2014-11-22 CRAN (R 3.3.0)
MASS           7.3-47  2017-04-21 CRAN (R 3.3.3)
Matrix       * 1.2-8   2017-01-20 CRAN (R 3.3.3)
MatrixModels   0.4-1   2015-08-22 CRAN (R 3.3.0)
mgcv           1.8-17  2017-02-08 CRAN (R 3.3.3)
minqa          1.2.4   2014-10-09 CRAN (R 3.3.0)
mlbench      * 2.1-1   2012-07-10 CRAN (R 3.3.0)
ModelMetrics   1.1.0   2016-08-26 CRAN (R 3.3.0)
munsell        0.4.3   2016-02-13 CRAN (R 3.3.0)
nlme           3.1-131 2017-02-06 CRAN (R 3.3.3)
nloptr         1.0.4   2014-08-04 CRAN (R 3.3.0)
nnet         * 7.3-12  2016-02-02 CRAN (R 3.3.3)
partykit     * 1.1-1   2016-09-20 CRAN (R 3.3.3)
pbkrtest       0.4-7   2017-03-15 CRAN (R 3.3.2)
```

Session Info (pt4)

```
package      * version  date       source
pkgconfig      2.0.1    2017-03-21 cran (@2.0.1)
plyr         * 1.8.4    2016-06-08 CRAN (R 3.3.0)
pROC         * 1.9.1    2017-02-05 CRAN (R 3.3.3)
prodlim        1.6.1    2017-03-06 cran (@1.6.1)
proxy        * 0.4-17   2017-02-01 CRAN (R 3.3.3)
purrr          0.2.2.2  2017-05-11 cran (@0.2.2.2)
quantmod       0.4-8    2017-04-19 CRAN (R 3.3.2)
quantreg       5.33     2017-04-18 CRAN (R 3.3.2)
R6             2.2.2    2017-06-17 cran (@2.2.2)
RANN           2.5.1    2017-05-21 CRAN (R 3.3.2)
RColorBrewer * 1.1-2    2014-12-07 CRAN (R 3.3.3)
Rcpp           0.12.12  2017-07-15 cran (@0.12.12)
RcppRoll       0.2.2    2015-04-05 CRAN (R 3.3.0)
recipes      * 0.1.0    2017-07-27 CRAN (R 3.3.2)
reshape2       1.4.2    2016-10-22 CRAN (R 3.3.3)
rlang          0.1.1    2017-05-18 CRAN (R 3.3.2)
robustbase     0.92-7   2016-12-09 CRAN (R 3.3.2)
ROCR           1.0-7    2015-03-26 CRAN (R 3.3.0)
ROSE         * 0.0-3    2014-07-15 CRAN (R 3.3.0)
rpart        * 4.1-11   2017-04-21 CRAN (R 3.3.3)
RRF            1.7      2017-01-26 CRAN (R 3.3.2)
scales         0.4.1    2016-11-09 CRAN (R 3.3.2)
sessioninfo  * 1.0.0    2017-06-21 CRAN (R 3.3.2)
SparseM        1.77     2017-04-23 CRAN (R 3.3.2)
stringi        1.1.5    2017-04-07 CRAN (R 3.3.2)
stringr        1.2.0    2017-02-18 CRAN (R 3.3.2)
survival     * 2.40-1   2016-10-30 CRAN (R 3.3.3)
tibble         1.3.3    2017-05-28 CRAN (R 3.3.2)
tidyselect     0.1.1    2017-07-24 CRAN (R 3.3.2)
timeDate       3012.100 2015-01-23 cran (@3012.10)
```

Session Info (pt5)

```
package * version date       source
TTR       0.23-1  2016-03-21 CRAN (R 3.3.0)
vcd     * 1.4-3   2016-09-17 CRAN (R 3.3.0)
withr     2.0.0   2017-07-29 Github (jimhester/withr@190d293)
xtable    1.8-2   2016-02-05 CRAN (R 3.3.3)
xts       0.9-7   2014-01-02 CRAN (R 3.3.0)
zoo       1.8-0   2017-04-12 CRAN (R 3.3.2)
```

Backup/Extra Slides

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 Tierney *et al* (2009, *Journal of Statistical Software*) 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). doParallel works well for Windows (I'm told)

foreach and caret

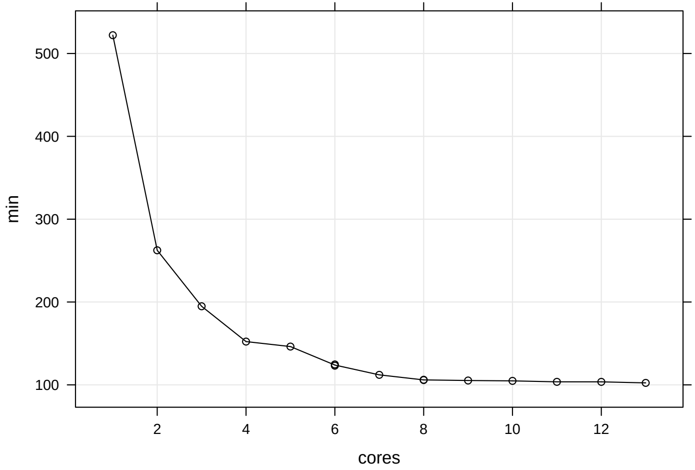
To use parallel processing in caret, no changes are needed when calling `train`.

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

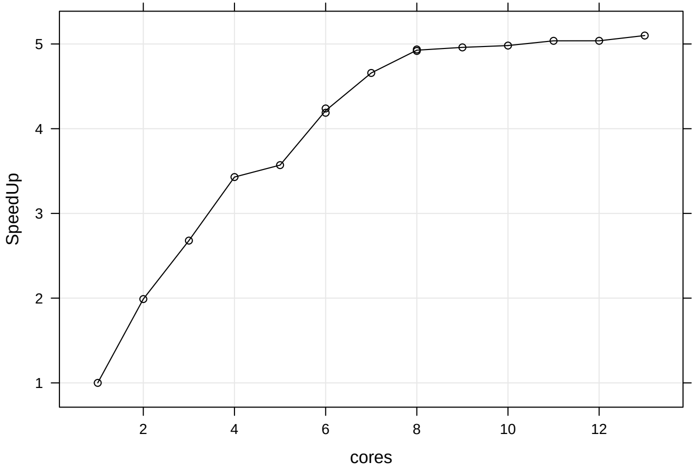
```
> library(doMC)          # on unix, linux or OS X
> ## library(doParallel) # windows and others
> registerDoMC(cores = 2)
```

Training Time (min)

50 bootstraps of a SVM model with 1000 samples and 400 predictors and the multicore package



Speed–Up


SpeedUp
cores