R for Machine Learning

Max Kuhn, Ph.D

RStudio
max@RStudio.com

Outline

- Modeling conventions
- Modeling capabilities
- Data splitting
- Pre–processing
- Measuring performance
- Over–fitting and resampling
- Logistic Regression
- Classification trees, boosting
- Extra topics as time allows:
  - Introduction to feature selection
  - Comparing mModels
  - Misc. functions in caret

Modeling Conventions in **R**

Terminology

Data:

- numeric data: numbers of any type (eg. counts, sales price)
- categorical or nominal data: non–numeric data (eg. color, gender)

Variables:

- outcomes: the data to be predicted
- predictors (aka independent variables, descriptors, inputs): data used to predict the outcome

Models:

- classification: models to predict categorical outcomes
- regression: models to predict numeric outcomes

(these last two are imperfect definitions)

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.

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 = 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

Almost all modeling functions in **R** follow 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.

Modeling Capabilities

Predictive Modeling Methods in R

As previously mentioned, there is a machine learning Task View page on the R website that does a good job of describing the range of models available

- **parametric regression models**: ordinary/generalized/robust regression models; partial least squares; projection pursuit regression; multivariate adaptive regression splines; principal component regression; neural networks; long short-term memory networks; recurrent neural networks; autoencoders
- **sparse/penalized models**: ridge regression; the lasso; the elastic net; generalized linear models; partial least squares; nearest shrunken centroids; logistic regression
- **kernel methods**: support vector machines; relevance vector machines; least squares support vector machine; Gaussian processes

(more)

Predictive Modeling Methods in **R**

- **trees/rule–based models**: CART; C4.5; conditional inference trees; node harvest, Cubist, C5.0
- **ensembles**: random forest; boosting (trees, linear models, generalized additive models, generalized linear models, others); bagging (trees, multivariate adaptive regression splines), rotation forests
- **prototype methods**: k nearest neighbors; learned vector quantization
- **discriminant analysis**: linear; quadratic; penalized; stabilized; sparse; mixture; regularized; stepwise; flexible
- **others**: naive Bayes; Bayesian multinomial probit models; Bayesian networks

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 237)
- 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

The Future

At RStudio, a new suite of modern modeling tool are being created that are more modular and consistent with the tidyverse (e.g. dplyr, purrr, and other packages).

This ecosystem is not finalized and is not discussed here. The current package list includes

- rsample
- recipes
- tidyposterior
- yardstick
- broom

More materials can be found here.

Example Data

Software Reseller Data Set

These data can be found at this course website and are described here in more detail.

Data were collected to predict if someone would make a purchase from their catalogs.

The data are moderate size: 2000 samples and 8 predictors.

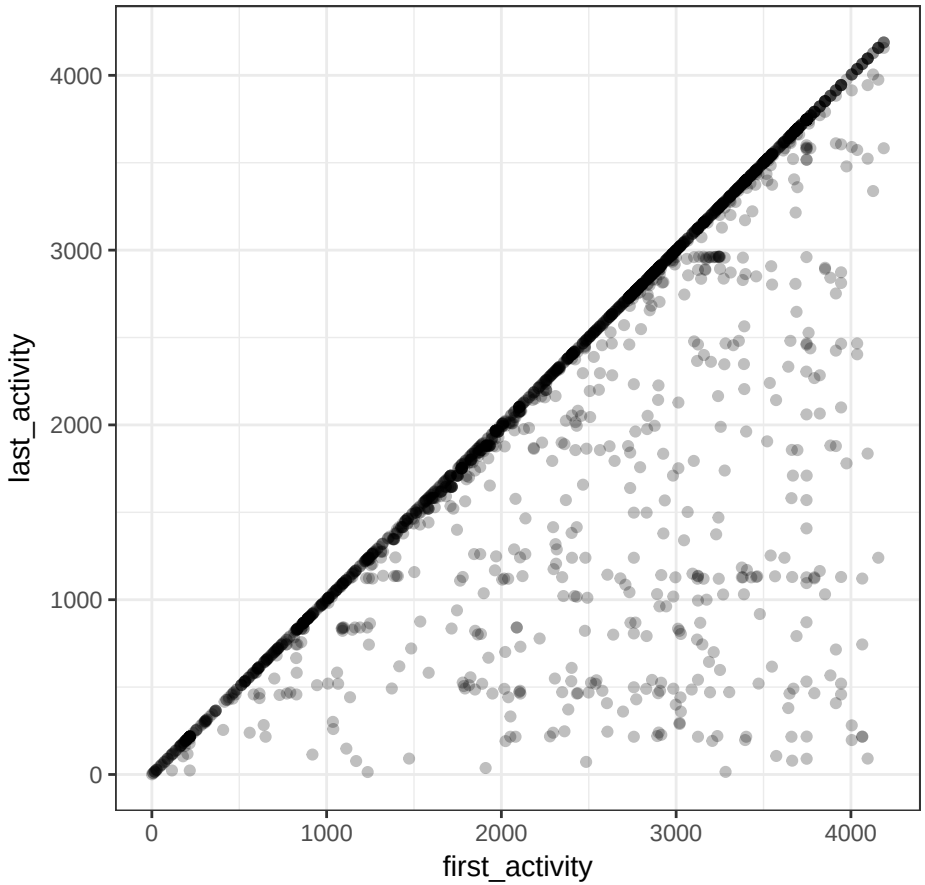
In reality, the rate of purchases is about 5% but these data have been pre–balanced to be 50%. This won't complicate our analysis but balancing the test set is generally **a bad idea**.

Software Reseller Data Set

```
> load("Tayko.RData")
> str(all_data)
'data.frame': 2000 obs. of  9 variables:
 $ in_us         : int  1 1 1 1 1 1 1 1 1 1 ...
 $ num_trans     : int  2 0 2 1 1 1 2 1 4 1 ...
 $ last_activity : int  3662 2900 3883 829 869 1995 1498 3397 525 3215 ...
 $ first_activity: int  3662 2900 3914 829 869 2002 1529 3397 2914 3215 ...
 $ web           : int  1 1 0 0 0 0 0 0 1 0 ...
 $ male          : int  0 1 0 1 0 0 0 1 1 0 ...
 $ residence     : int  1 0 0 0 0 1 1 0 0 0 ...
 $ purch         : Factor w/ 2 levels "yes","no": 1 2 1 2 2 2 2 2 1 1 ...
 $ catalog       : Factor w/ 16 levels "other","a","b",..: 3 6 12 4 4 10 16 3 2 2 .
```

The Two Activity Columns

```
> ggplot(all_data, aes(x = first_activity, y= last_activity)) +
+   geom_point(alpha = .25) + coord_equal()
```



Activity Columns

Since 63.3% of these data are the same, let's re-encode it as the *activity length* instead.

```
> all_data$activity_length <- all_data$first_activity - all_data$last_activity
> all_data <- all_data[, -grep("_activity", names(all_data))]
> predictors <- names(all_data)[names(all_data) != "purch"]
> all_data <- subset(all_data, num_trans > 0)
```

Also, it isn't quite fair to include data where there have been zero previous transactions (as none of these made a purchase)

```
> all_data <- subset(all_data, num_trans > 0)
> nrow(all_data)
```

```
[1] 1602
```

General Strategies

APM Ch 1, 2 and 4. FES Review Chapter

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**: 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

Software Reseller Data Set

For these data, let's take a stratified random sample of the data for training.

```
> set.seed(8131)
> in_train <- createDataPartition(all_data$purch, p = 3/4, list = FALSE)
> head(in_train)
```

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

```
> train_data <- all_data[ in_train,]
> test_data  <- all_data[-in_train,]
```

Estimating Performance

APM Ch. 5 and 11, FES Review Chapter

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
- the **root mean square error** is a common metric for understanding the performance
- **Spearman's correlation** may be applicable for models that are used to rank samples

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
- 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.

We'll use the `confusionMatrix` function 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 a **purchase**:

$$\text{Sensitivity} = \frac{\#\text{ truly purchase predicted to be purchase}}{\#\text{ truly purchase}}$$

$$\text{Specificity} = \frac{\#\text{ truly no purchase predicted to be no purchase}}{\#\text{ truly no purchase}}$$

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

Probability Cutoffs

Most classification models produce a predicted class probability that is converted into a predicted class.

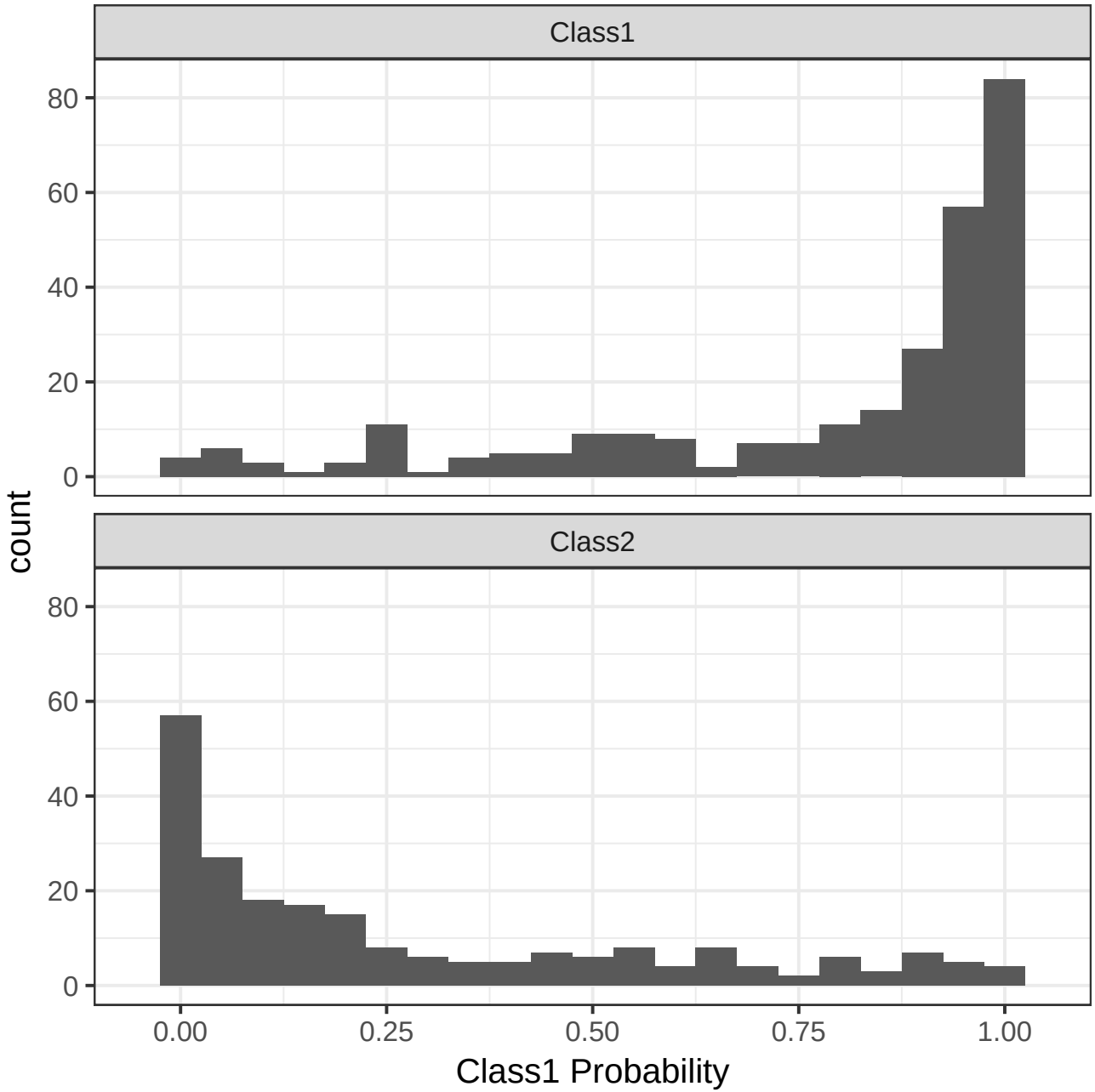
For two classes, the 50% cutoff is customary; if the probability of a purchase is $\geq 50\%$, they would be labelled as an actual purchase.

What happens when you change the cutoff?

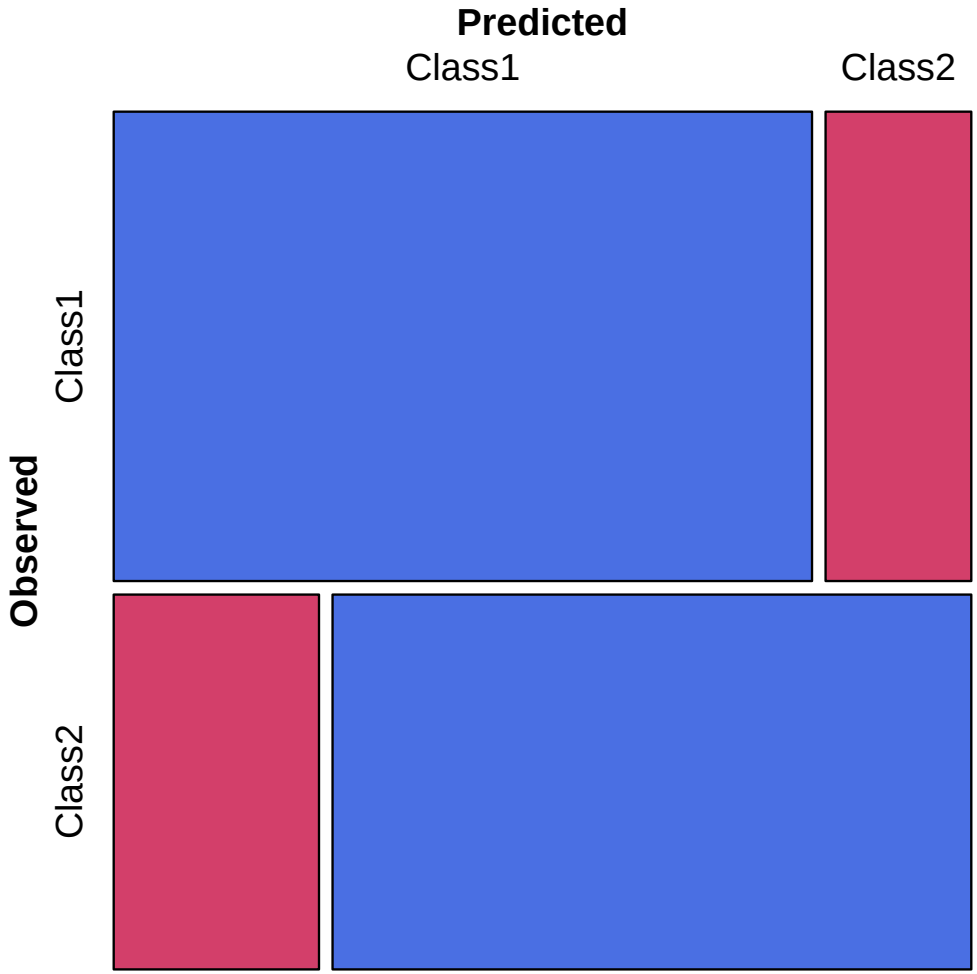
Increasing it makes it harder to predict a purchase $\rightarrow$ fewer predicted events, specificity $\uparrow$, sensitivity $\downarrow$

Decreasing the cutoff makes it easier to predict a purchase $\rightarrow$ more predicted events, specificity $\downarrow$, sensitivity $\uparrow$

Example Predictions for Two Classes

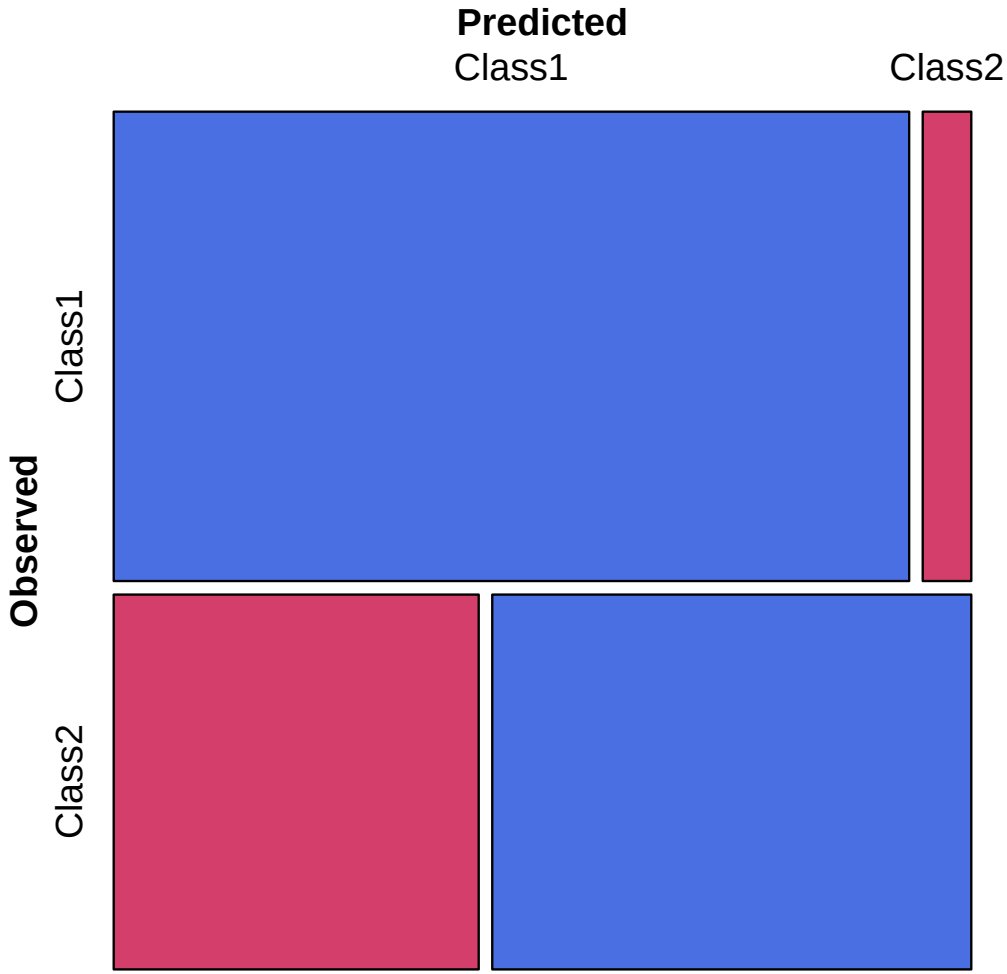


Confusion Matrix where $Prob \geq 50\%$ Is an Event



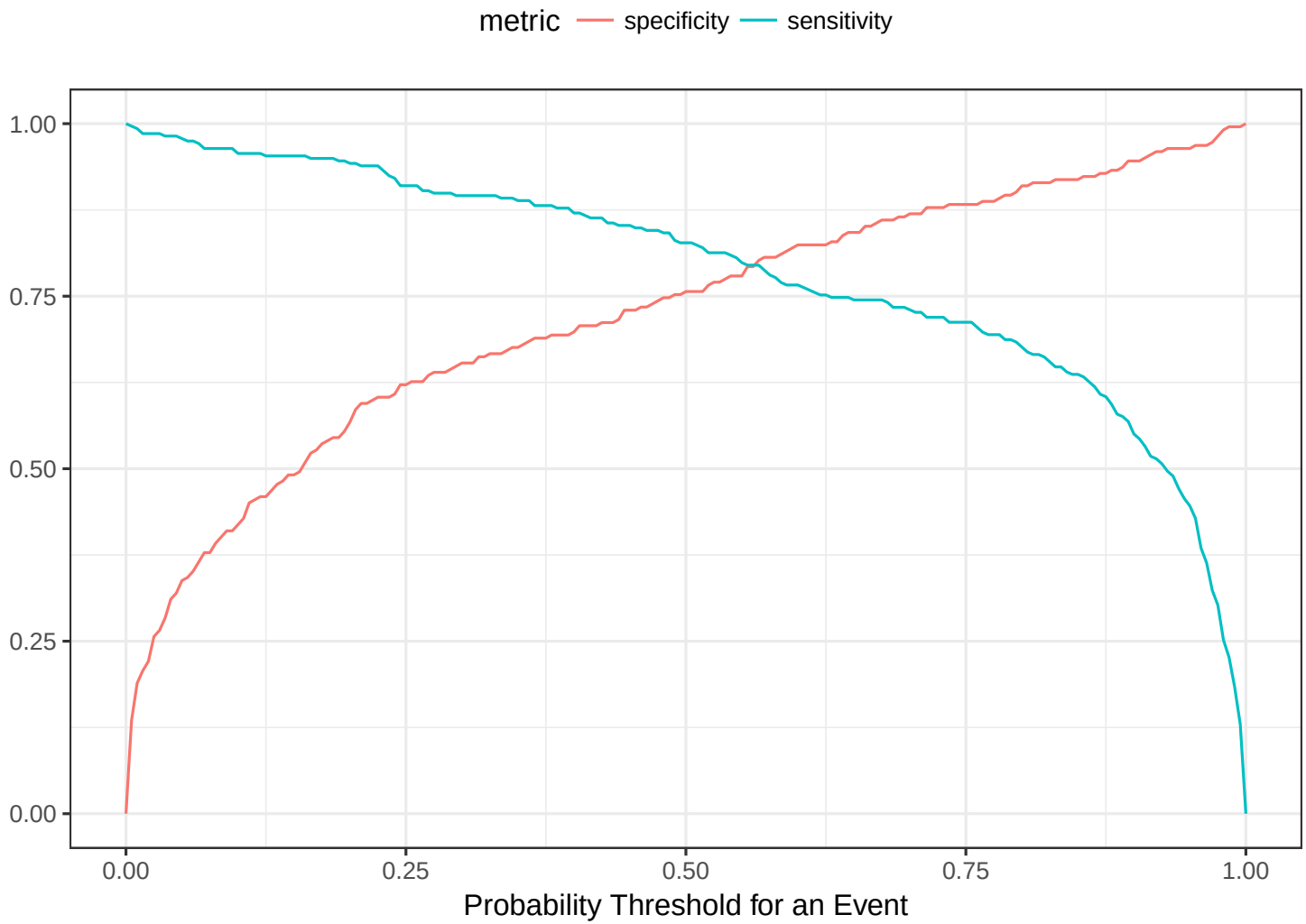
Sensitivity $= 82.7\%$, Specificity $= 75.7\%$,

Confusion Matrix with $Prob \geq 20\%$ Is an Event



Sensitivity $= 94.2\%$, Specificity $= 56.8\%$,

Changes Over Probability Thresholds



ROC Curve

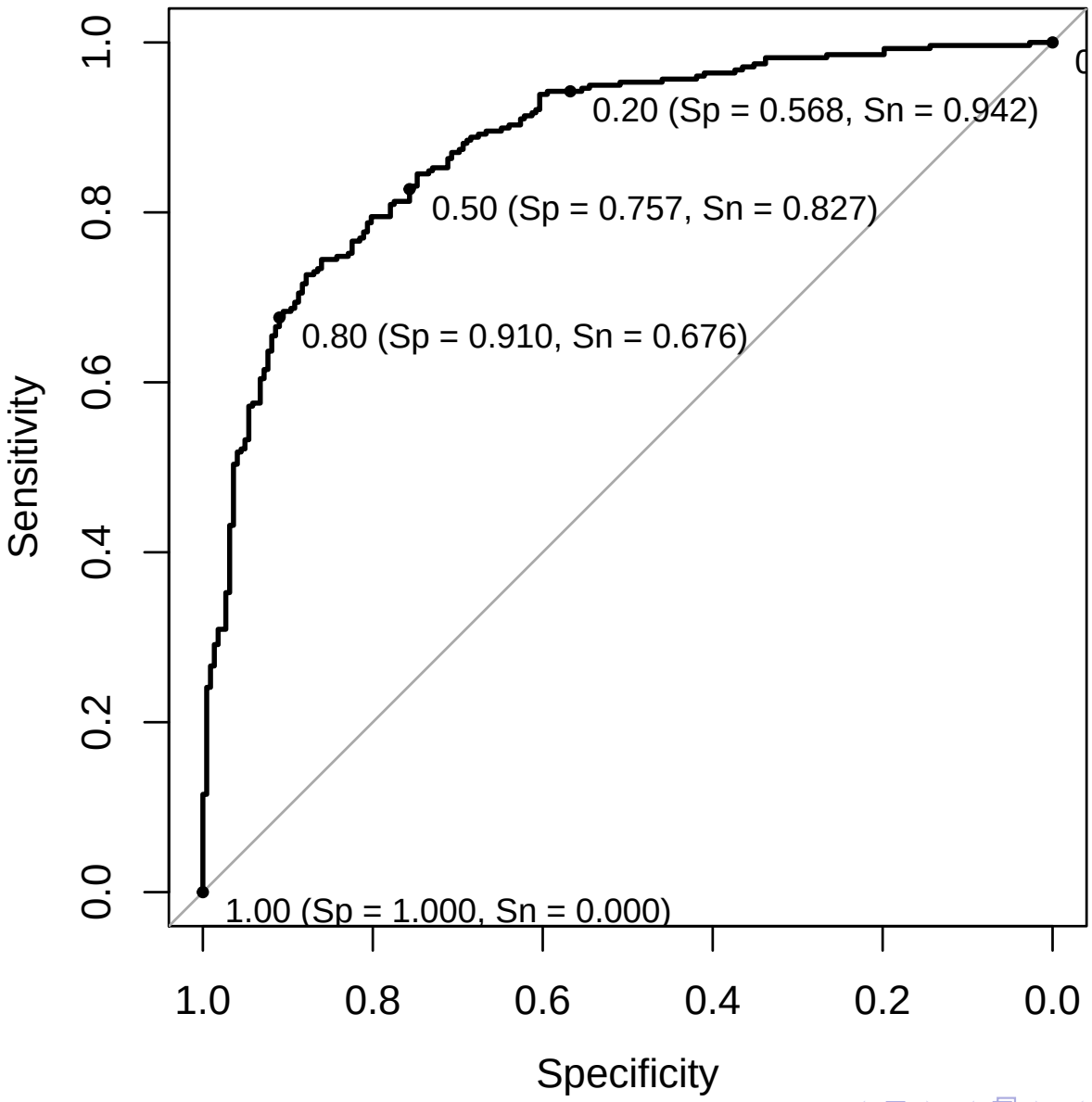
With two classes the Receiver Operating Characteristic (ROC) curve can be used to estimate performance using a combination of sensitivity and specificity.

Here, many alternative cutoffs are evaluated and, for each cutoff, we calculate the sensitivity and specificity.

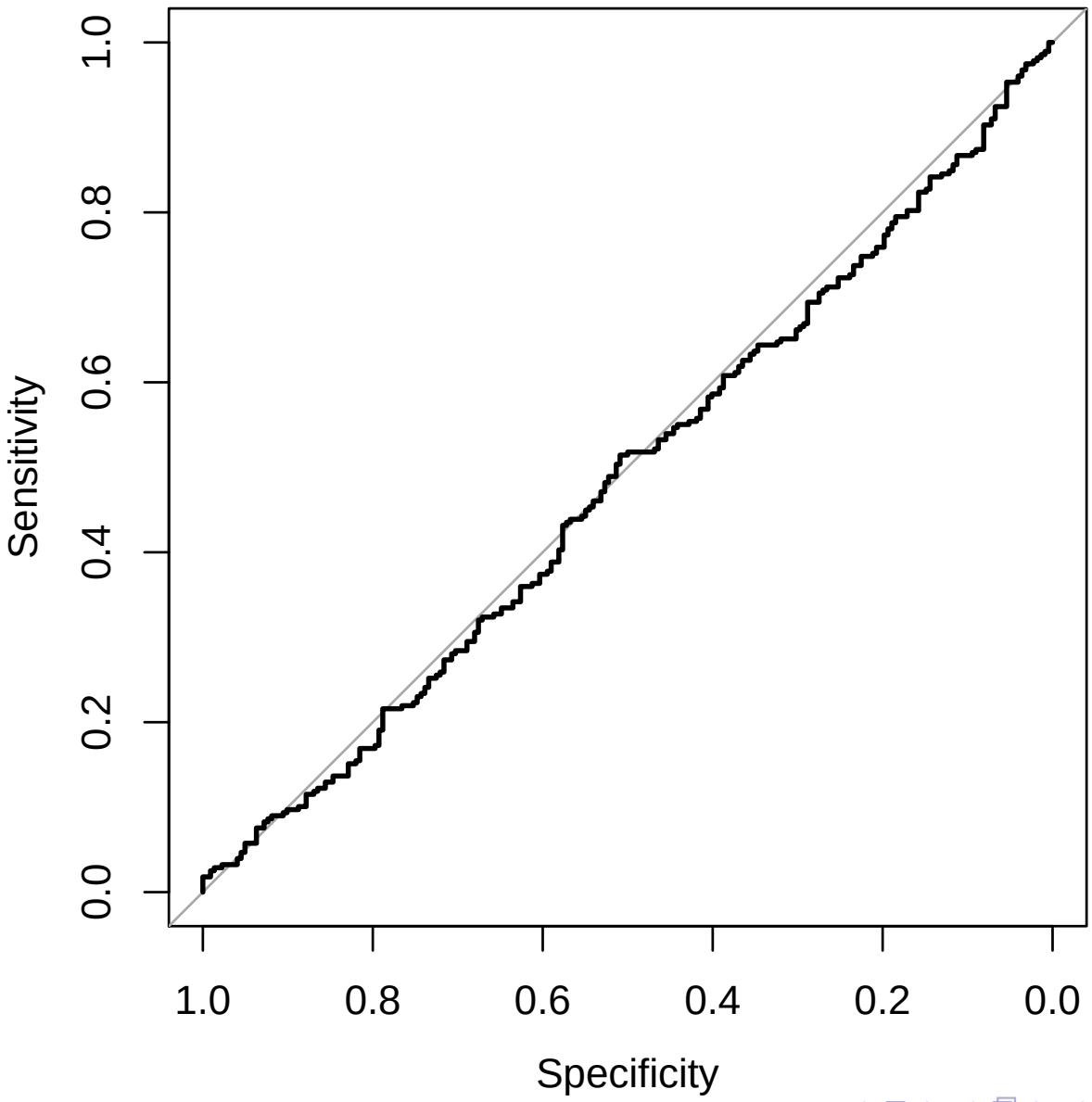
The ROC curve plots the sensitivity (eg. true positive rate) by one minus specificity (eg. the false positive rate).

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

ROC Curve (AUC = 0.88)



Poor ROC Curve (AUC = 0.484)



ROC Curve Packages

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

We'll focus on pROC in later examples.

Over–Fitting and Model Tuning

APM Ch. 4, FES Review Chapter

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

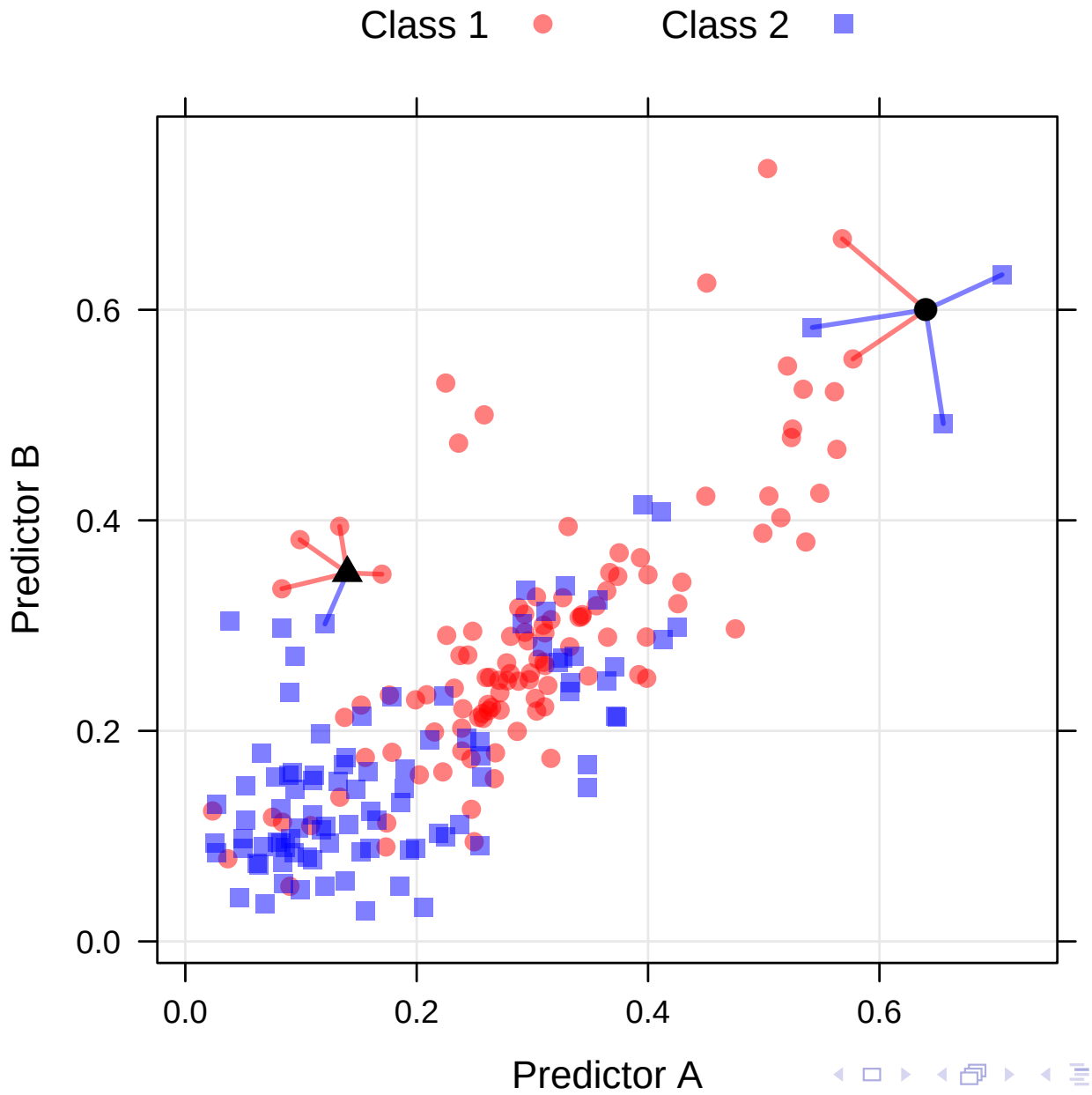
- neighborhood size in nearest neighbor models is an example
- the number if splits in a tree model

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

V–Nearest Neighbors Classification



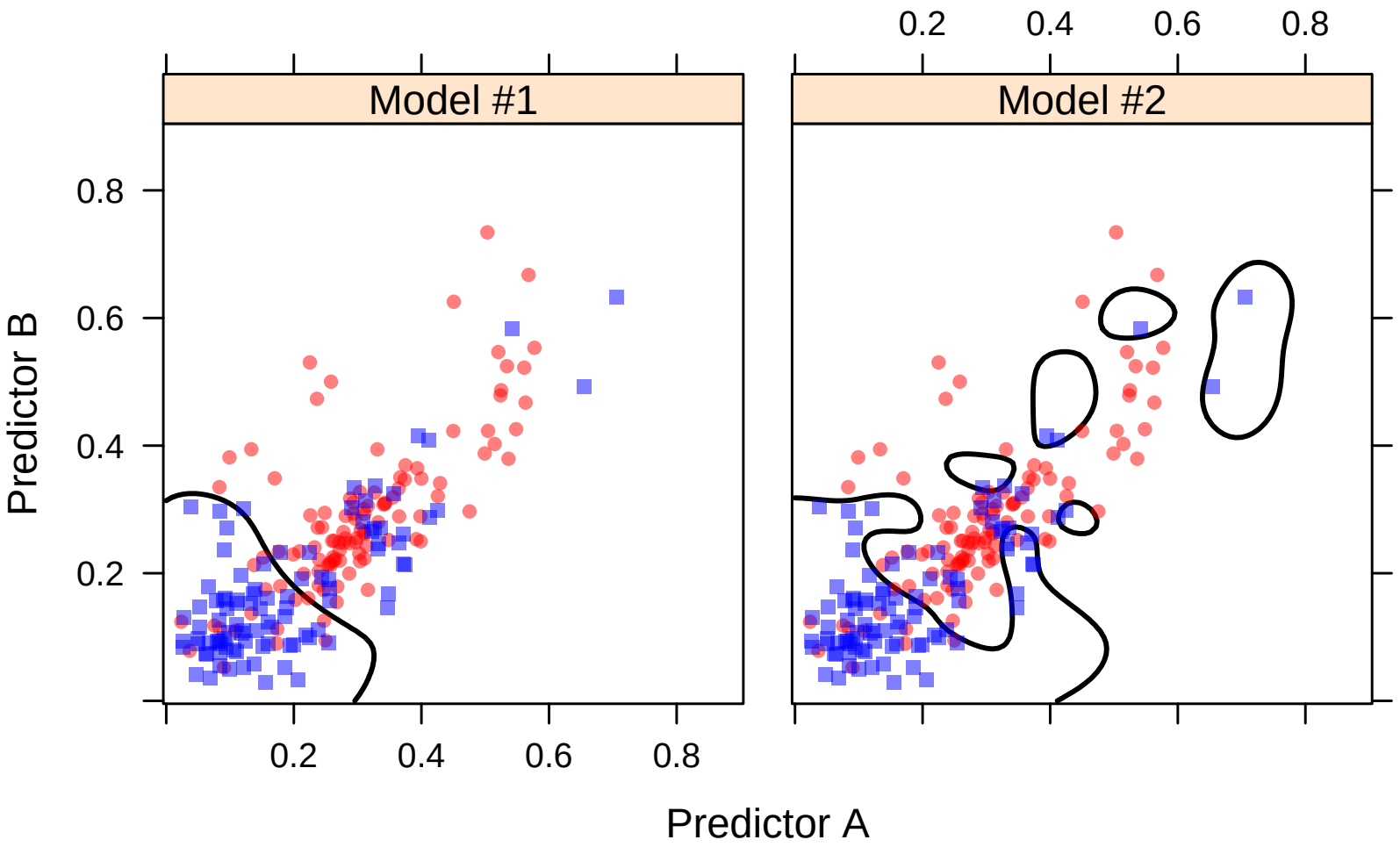
Over–Fitting

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



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.

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.

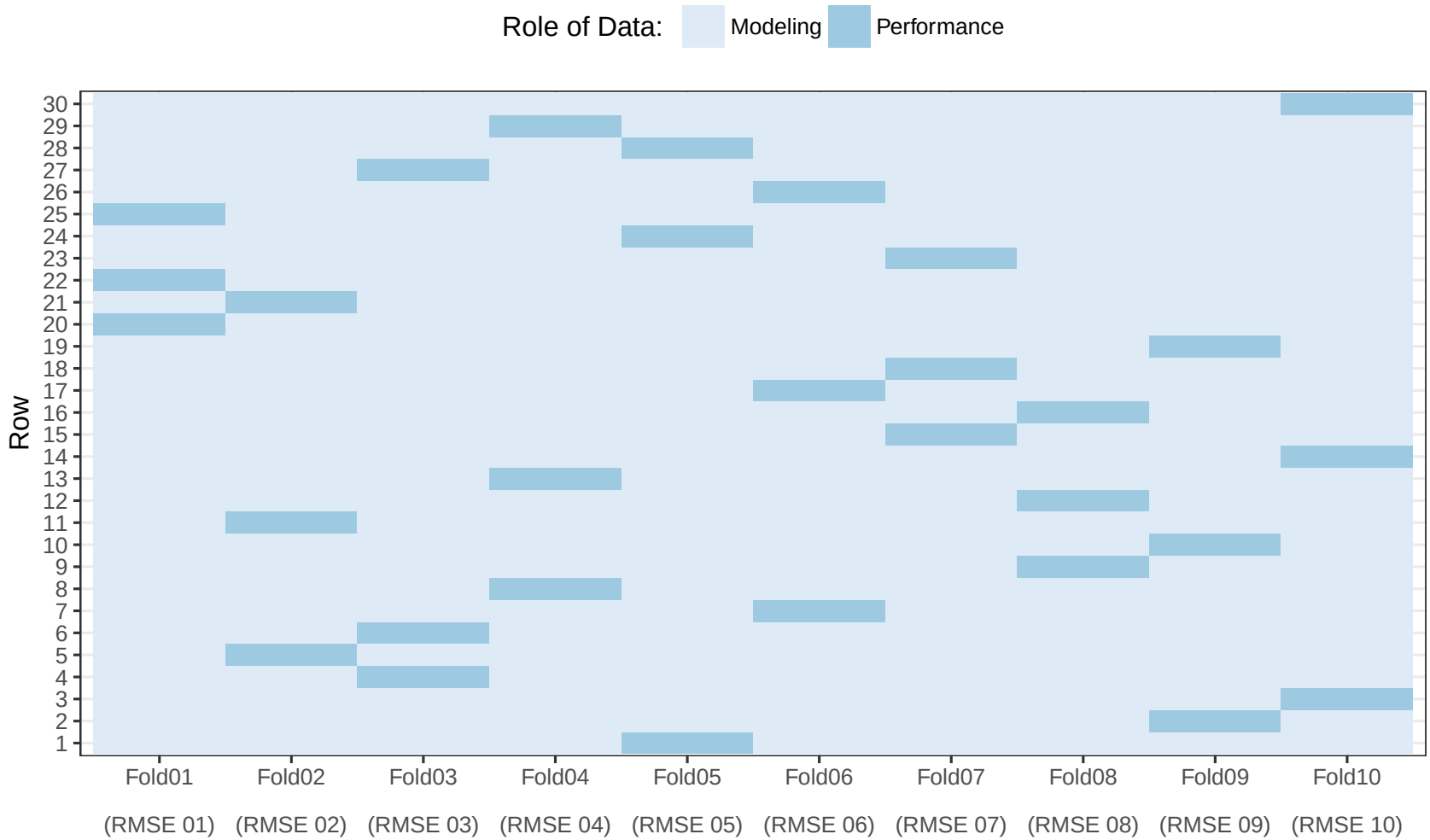
1. We leave out the first block of data and fit a model.
2. This model is used to predict the held-out block
3. We continue this process until we've predicted all V held–out blocks

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)

10–Fold Cross–Validation



In R

Many packages have cross–validation functions, but they are usually limited to 10–fold CV.

caret has a general purpose function called `train` that has many resampling methods for many models (more later).

caret has functions to produce samples splits for V–fold CV (`createFolds`), multiple training/test splits (`createDataPartition`) and bootstrap sampling (`createResample`).

Also, the base **R** function `sample` can be used to create completely random splits or bootstrap samples

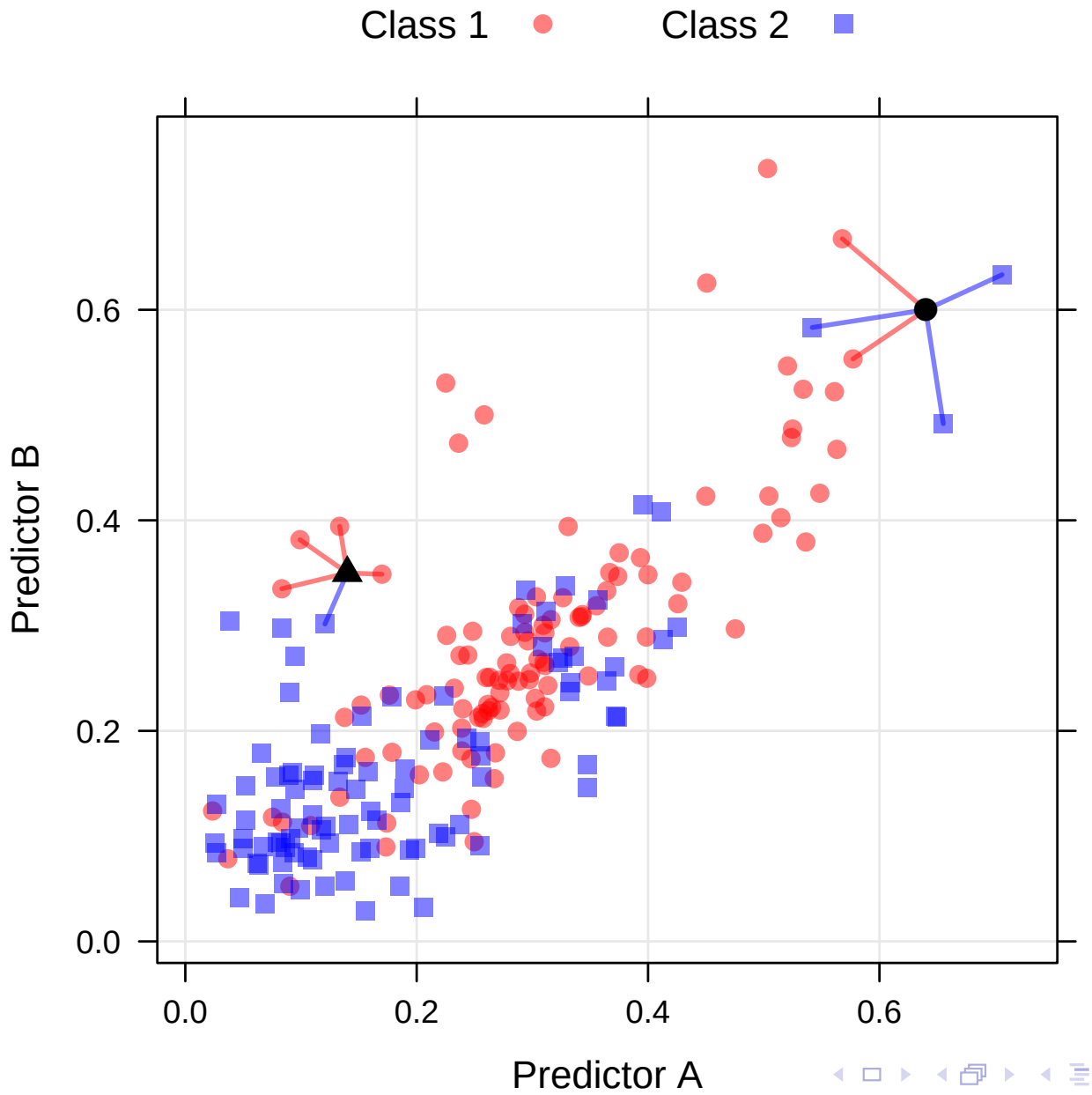
The Big Picture

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

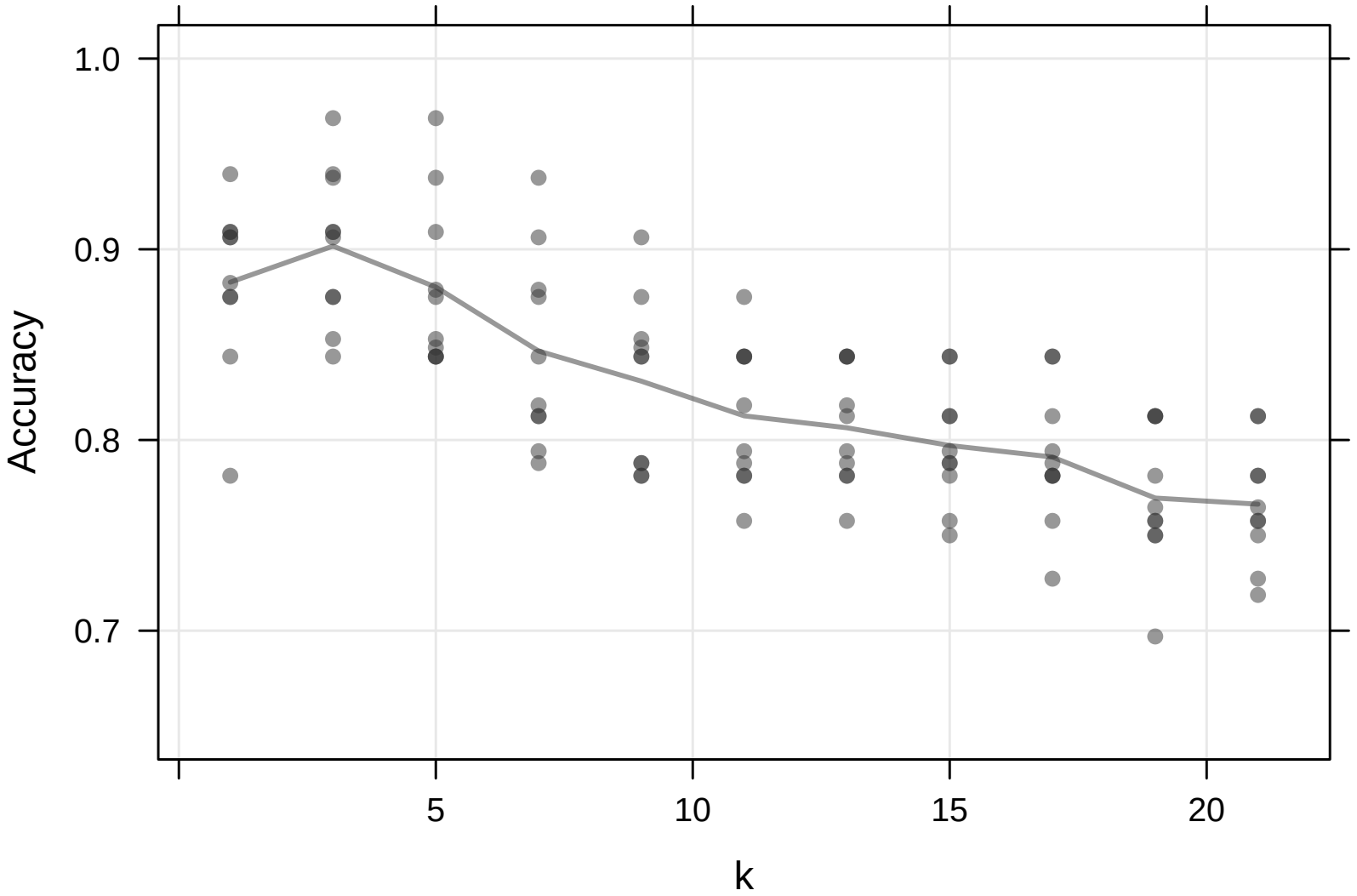


The Big Picture – KNN Example

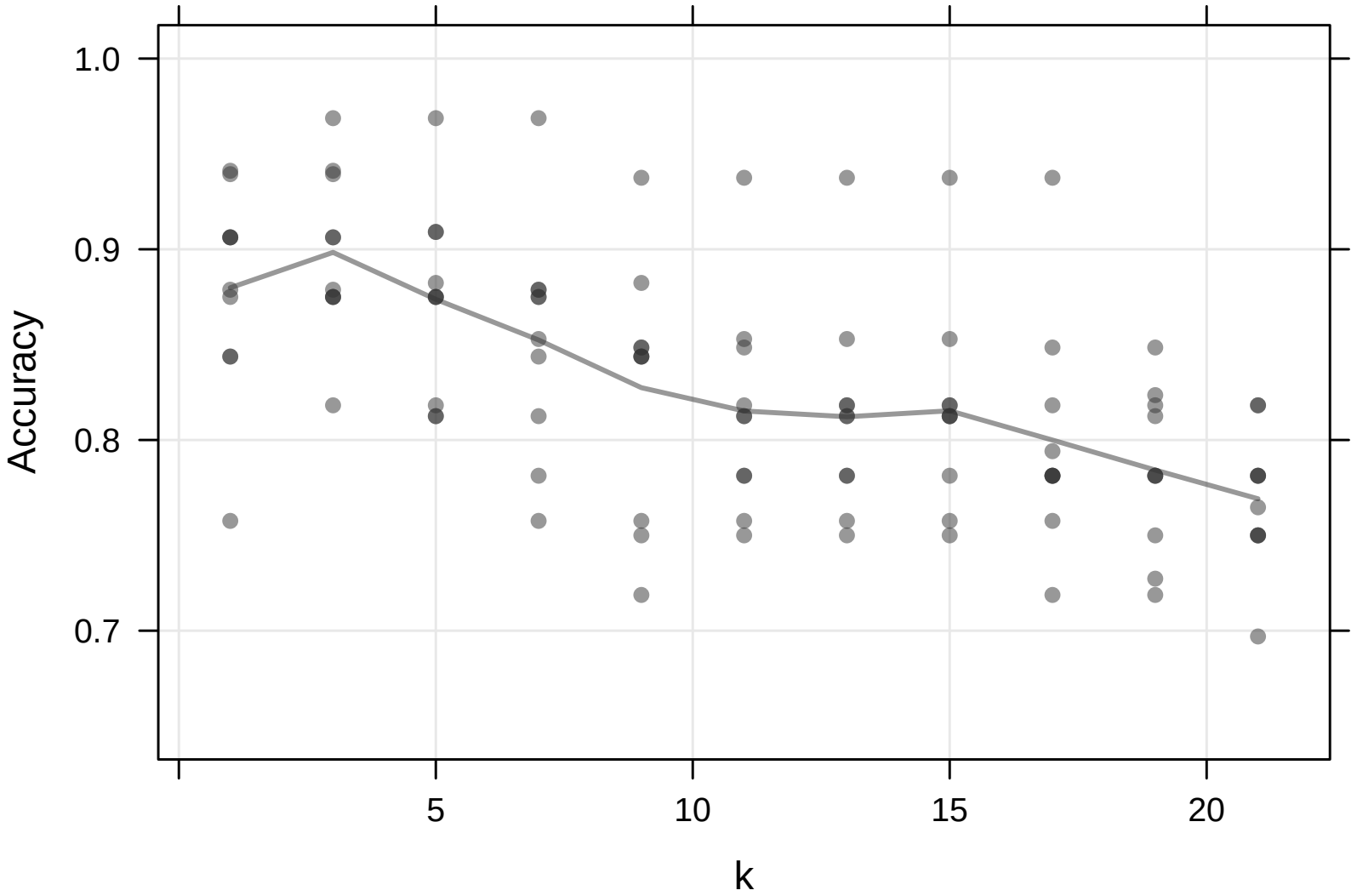
Using k–nearest neighbors as an example:

Randomly put samples into 10 distinct groups;
for $k = 1, 3, 5, \ldots, 21$ **do**
  for $i = 1 \ldots 10$ **do**
    Hold–out block i ;
    Fit the model on the other 90%;
    Predict the i^{th} block and save results;
  end
  Calculate the average accuracy across the 10 hold–out sets of predictions
end
Determine k based on the highest cross–validated accuracy;

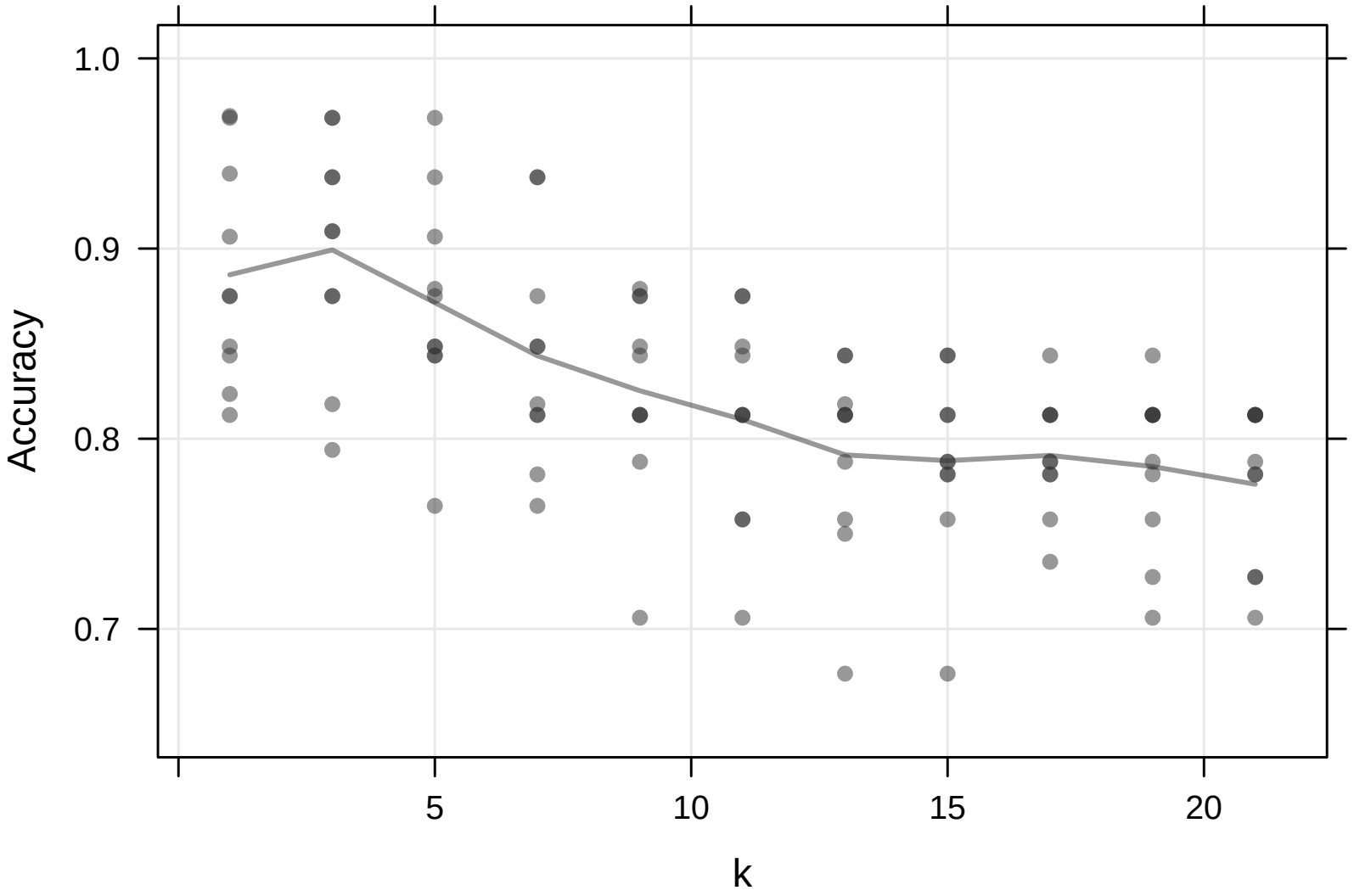
The Big Picture – KNN Example



With a Different Set of Resamples



With a Different Different Set of Resamples



Data Pre–Processing

APM Ch. 3, FES Numeric Predictor and Categorical Predictor chapters

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
- transformations of the groups of predictors, such as the
  - the "spatial–sign" transformation (i.e. $x' = x/||x||$)
  - feature extraction via PCA

Dummy Variables

Before pre-processing the data, there are a few predictors that are categorical in nature.

For these, we would convert the values to binary *dummy variables* prior to using them in numerical computations.

If a categorical predictors has C levels, it would make $C - 1$ variables with values 0 or 1 (one level would be omitted).

The core **R** function `model.matrix` can be used to do this.

Most functions that use the formula interface will automatically create dummy variables from factors.

Dummy Variables

In the data, the catalog predictor has $C = 16$ distinct values: "other", "a," and "b", and so on. For these data, the possible dummy variables are:

	Dummy Variable Columns				
Data Value	other	a	b	···	x
"other"	1	0	0	···	0
"a"	0	1	0	···	0
"b"	0	0	1	···	0
⋮	⋮	⋮	⋮		⋮
"x"	0	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 and the FES Section.

Manually Creating Dummy Variables

Using the formula method for a model will *usually* create the dummy variables automatically (see next slide).

If you have a situation where you need to create them explicitly, the `dummyVars` function can create them:

```
> dummies <- dummyVars( ~ ., data = train_data[, predictors],
+                       fullRank = TRUE)
> train_dummies <- predict(dummies, train_data[, predictors])
> test_dummies  <- predict(dummies, test_data[, predictors])
> colnames(train_dummies)
 [1] "in_us"           "num_trans"       "web"             "male"
 [5] "residence"       "catalog.a"       "catalog.b"       "catalog.c"
 [9] "catalog.d"       "catalog.e"       "catalog.h"       "catalog.m"
[13] "catalog.o"       "catalog.p"       "catalog.r"       "catalog.s"
[17] "catalog.t"       "catalog.u"       "catalog.w"       "catalog.x"
[21] "activity_length"
```

Dummy Variables and Model Functions

Most models are parameterized so that the predictors are *required* to be numeric. Linear regression, for example, doesn't know what to do with a raw value of "green."

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

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.

Centering and Scaling

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:

```
> pp_values <- preProcess(train_data[, predictors],
+                         method = c("center", "scale"))
> pp_values

Created from 1202 samples and 7 variables

Pre-processing:
  - centered (6)
  - ignored (1)
  - scaled (6)
```

Apply them to the training and test sets:

```
> train_scaled <- predict(pp_values, newdata = train_data[, predictors])
> test_scaled  <- predict(pp_values, newdata = test_data[, predictors])
```

Signal Extraction via PCA

Principal component analysis (PCA) can be used to create a (hopefully) small subset of new predictors that capture most of the information in the whole set.

The principal components are linear combinations of each individual predictor and usually have not meaningful interpretation.

The components are created sequentially

- the first captures the larges component of variation in the predictors
- the second does the same for the leftover information, and so on

We can track how much of the variance is explained by the components and select enough to have fidelity to the original data.

Signal Extraction via PCA

The advantages to this approach:

- the components are all uncorrected to one another
- a small number of predictors in the model can sometimes help

However...

- this is not feature selection; all the predictors are still required
- the components may not be correlated to the outcome

Signal Extraction via PCA in R

The base R function `prcomp` can be used to create the components.

`preProcess` can make the transformation and automatically retain the number of components to account for some pre–specified amount of information.

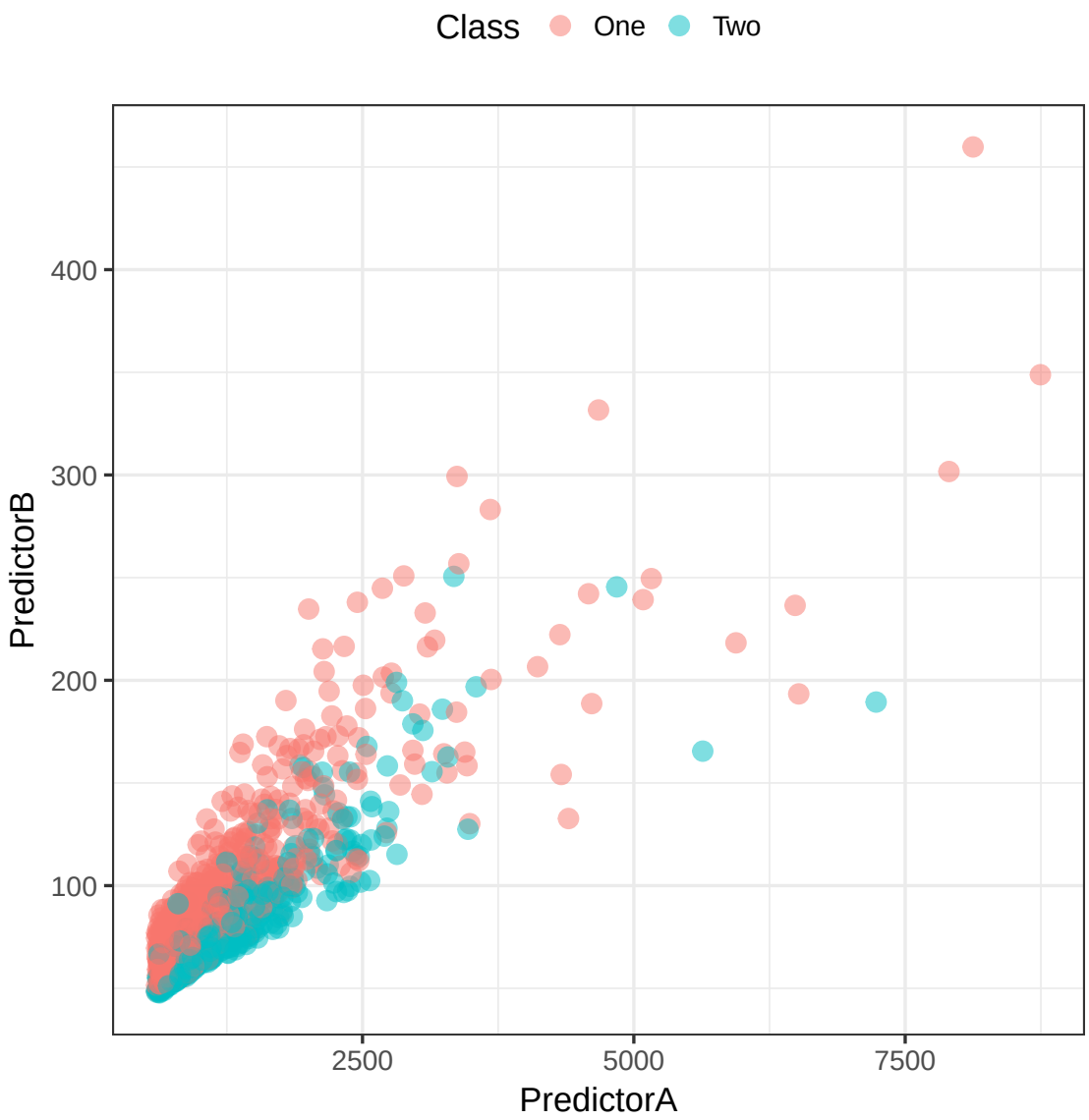
The predictors should be centered and scaled prior the PCA extraction. `preProcess` will automatically do this even if you forget.

An Example

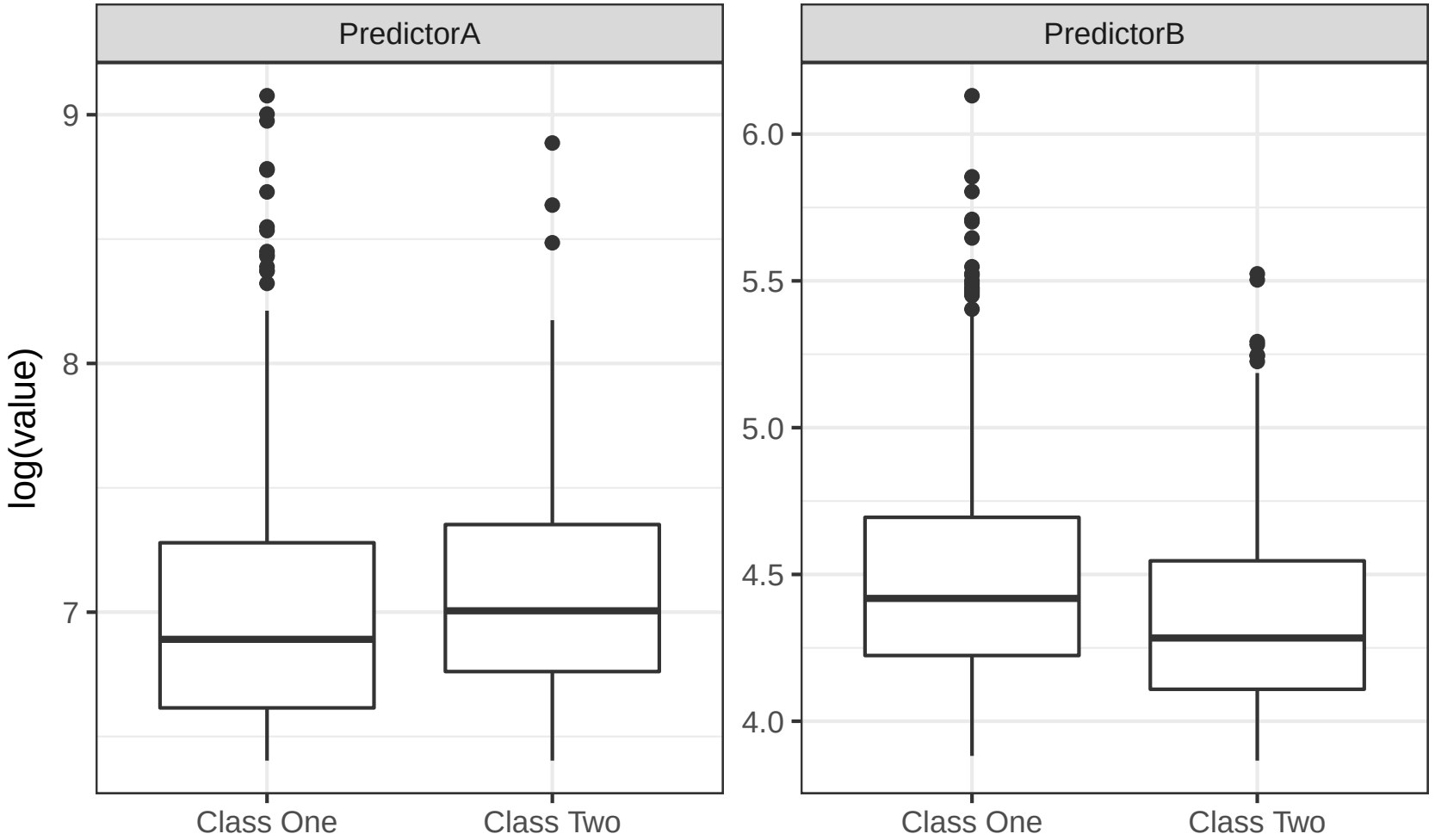
Another data set shows an nice example of PCA. There are two the predictors and two classes:

```
> dim(example_train)

[1] 1009    3

> dim(example_test)

[1] 1010    3

> head(example_train)

   PredictorA PredictorB Class
2        3279      154.9   One
3        1727       84.6   Two
4        1195      101.1   One
12       1027       68.7   Two
15       1036       73.4   One
16       1434       79.5   One
```

Correlated Predictors



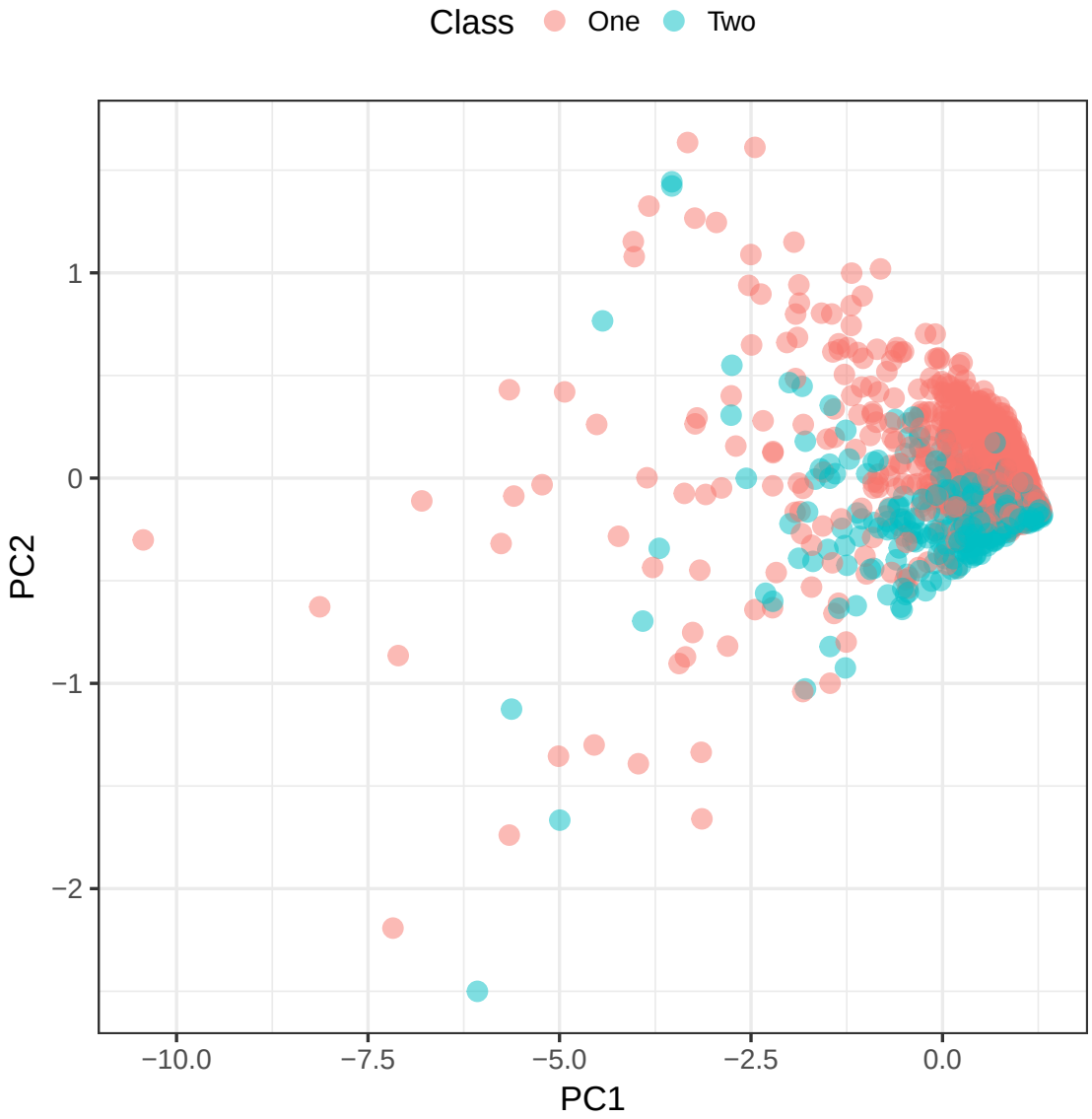
Mildly Predictive of the Classes



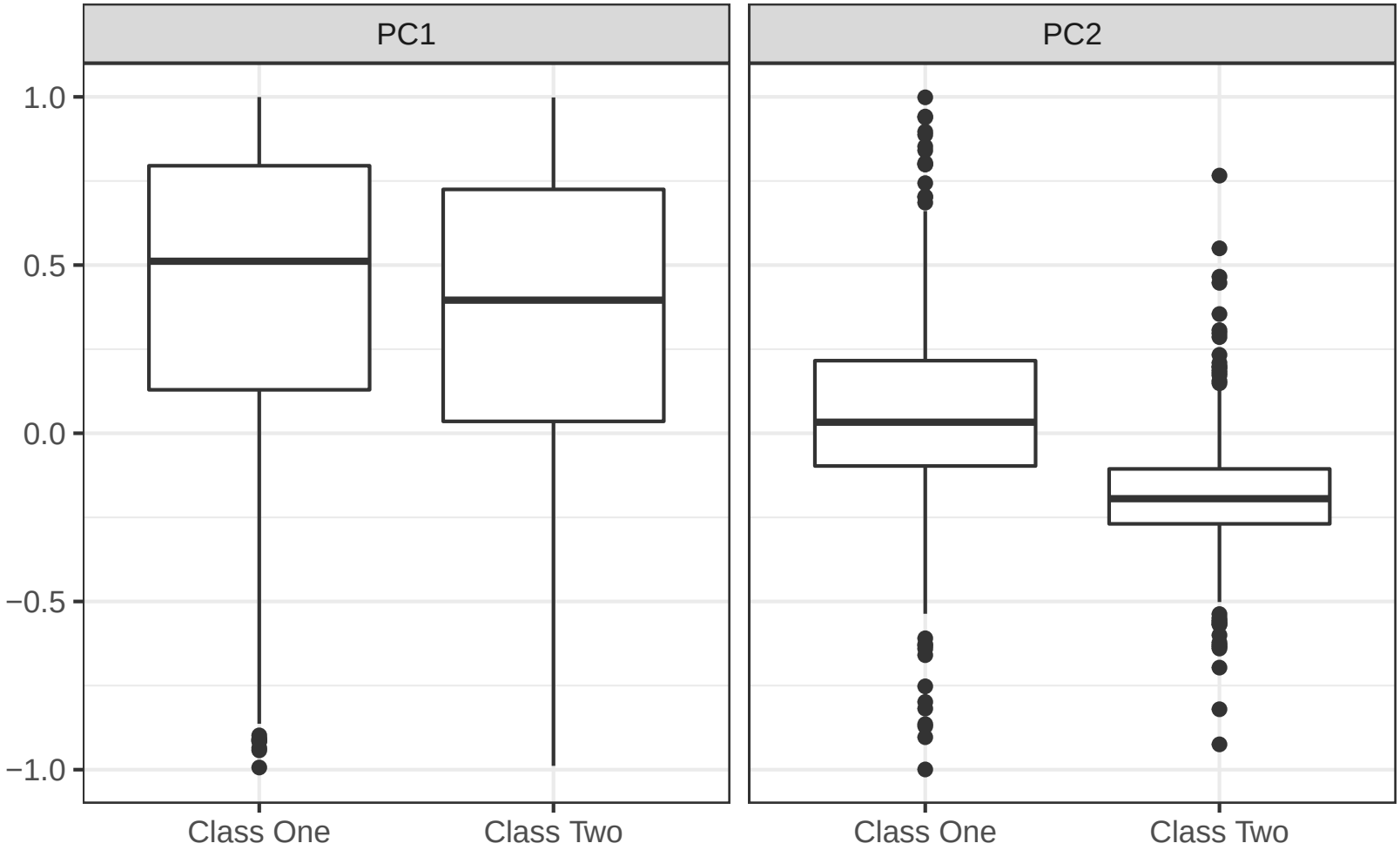
An Example

```
> pca_pp <- preProcess(example_train[, 1:2],
+                      method = "pca") # also added "center" and "scale"
> pca_pp

Created from 1009 samples and 2 variables

Pre-processing:
  - centered (2)
  - ignored (0)
  - principal component signal extraction (2)
  - scaled (2)

PCA needed 2 components to capture 95 percent of the variance

> train_pc <- predict(pca_pp, example_train[, 1:2])
> test_pc <- predict(pca_pp, example_test[, 1:2])
> head(test_pc, 4)

    PC1     PC2
1 0.842  0.0728
5 0.219  0.0457
6 1.207 -0.2104
7 1.179 -0.2098
```

A simple Rotation in 2D



The First PC is Least Important Here



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.

Filtering Problematic Variables

Some models have computational issues if predictors have degenerate distributions. For example, models that use $X'X$ or it's inverse might have issues with

- outliers
- predictors with a single value (aka zero–variance predictors)
- highly unbalanced distributions

Other models are insensitive to the characteristics of the predictor distributions.

The caret functions `findCorrelation` and `nearZeroVar` can be used for unsupervised filtering of predictors.

Feature Engineering

One of the most critical parts of the modeling process is *feature engineering*; how should the predictors enter the model?

For example, two predictors might be more informative if they enter the model as a ratio instead of as two main effects.

This requires an in–depth knowledge of the problem at hand and the nuances of the data.

Also, like pre–processing, this is highly model dependent.

Classification Models

APM Ch. 11-17

Logistic Regression

APM Sect 14.5

Logistic Regression

Logistic regression is a standard classification model that tries to predict the outcome in a manner similar to linear regression, in that there are slopes for each predictor and an intercept.

Instead of modeling the probability of the response (p) directly, it models the *log-odds*:

$$log\left(\frac{p_i}{1-p_i}\right) = \beta_0 + \beta_1 x_{i1} + \ldots \beta_p x_{ip}$$

Parameter estimates are generated by maximizing the binomial likelihood.

Pros and Cons

+ Simple, stable, and interpretable model.
+ Enables prediction as well as inference.
+ No tuning parameters.
– Maximum likelihood $\neq$ maximum accuracy.
– Without significant feature engineering, may not be high performance.
– Suffers from correlated predictors.

glmnet Model

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

- an L_1 penalty (penalty is $\lambda_1 \sum |\beta_j|$) can have the effect of setting coefficients to zero.
- L_2 regularization ($\lambda_2 \sum \beta_j^2$) is basically ridge regression where the magnitude of the coefficients are dampened to avoid overfitting

Regularization is a way to dampen parameters values towards zero, especially if their corresponding predictors do not have a strong relationship with the outcome.

This is especially helpful when predictors are correlated; this leads to unstable and abnormally large parameters.

Regularization in the `glmnet` Model

For a `glmnet` model, we need to determine the total amount of 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).

The predictors require centering and scaling before being used in a `glmnet`, lasso, or ridge regression model.

Technical bits can be found in Statistical Learning with Sparsity.

Regularization in the glmnet Model

The glmnet package can be used to fit these models for a specific values of `alpha`.

Its syntax is somewhat abnormal in comparison to other models in R.

We'll use the `train` function to fit and tune these models over `alpha` and `lambda`.

Let's get introduced to the `train` function and sequentially build up its syntax.

The train Function

The basic syntax for the function is:

```
> train(formula, data, method) # or
> train(x, y, method)
```

Looking at ?train, using method = "glmnet" can be used to tune this model. We can use:

```
> # To automatically create dummy variables,
> # use the formula method:
> train(purch ~ ., data = train_data, method = "glmnet")
```

We'll add a bit of customization too.

The train Function

By default, the function will tune over 3 values of the tuning parameters for this model (e.g 9 combinations).

For glmnet, the train function determines the distinct number of tree depth values for the data.

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

```
> glmn_grid <- expand.grid(alpha = seq(0, 1, by = .25),
+                          lambda = 10^seq(-3, -1, length = 20))
>
> set.seed(1735)
> glmn_tune <- train(purch ~ ., data = train_data,
+                    method = "glmnet",
+                    tuneGrid = glmn_grid)
```

The train Function

We might want to choose the tuning parameters based on the largest 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 called . For example:

```
> twoClassSummary(fakeData)
```

```
   ROC   Sens   Spec
0.5020 0.1145 0.8827
```

The train Function

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

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

```
> glmn_grid <- expand.grid(alpha = seq(0, 1, by = .25),
+                          lambda = 10^seq(-3, -1, length = 20))
>
> cv_ctrl <- trainControl(method = "cv",
+                         summaryFunction = twoClassSummary,
+                         classProbs = TRUE)
>
> set.seed(1735)
> glmn_tune <- train(purch ~ ., data = train_data,
+                    method = "glmnet",
+                    tuneGrid = glmn_grid,
+                    metric = "ROC",
+                    trControl = cv_ctrl)
```

The train Function

Finally, we pre-process the predictors by using the preProc argument to ensure centering and scaling.

```
> glmn_grid <- expand.grid(alpha = seq(0, 1, by = .25),
+                          lambda = 10^seq(-3, -1, length = 20))
>
> cv_ctrl <- trainControl(method = "cv",
+                         summaryFunction = twoClassSummary,
+                         classProbs = TRUE)
>
> set.seed(1735)
> glmn_tune <- train(purch ~ ., data = train_data,
+                    method = "glmnet",
+                    tuneGrid = glmn_grid,
+                    metric = "ROC",
+                    preProc = c("center", "scale"),
+                    trControl = cv_ctrl)
```

glmnet Results

```
> glmn_tune



glmnet

1202 samples
   7 predictor
   2 classes: 'yes', 'no'

Pre-processing: centered (21), scaled (21)
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 1082, 1082, 1082, 1082, 1081, 1082, ...
Resampling results across tuning parameters:

  alpha  lambda   ROC    Sens   Spec
  0.00   0.00100  0.835  0.851  0.6066
  0.00   0.00127  0.835  0.851  0.6066
    :       :       :      :      :
  1.00   0.04833  0.832  0.980  0.2083
  1.00   0.06158  0.822  0.989  0.1417
  1.00   0.07848  0.810  0.992  0.1395
  1.00   0.10000  0.767  0.999  0.0686

ROC was used to select the optimal model using the largest value.
The final values used for the model were alpha = 1 and lambda = 0.0183.
```

Each value in the `ROC` column is the average ROC across all of the resampled holdouts for that parameter combination.

Working With the `train` Object

There are a few methods of interest:

- `plot.train` or `ggplot.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.

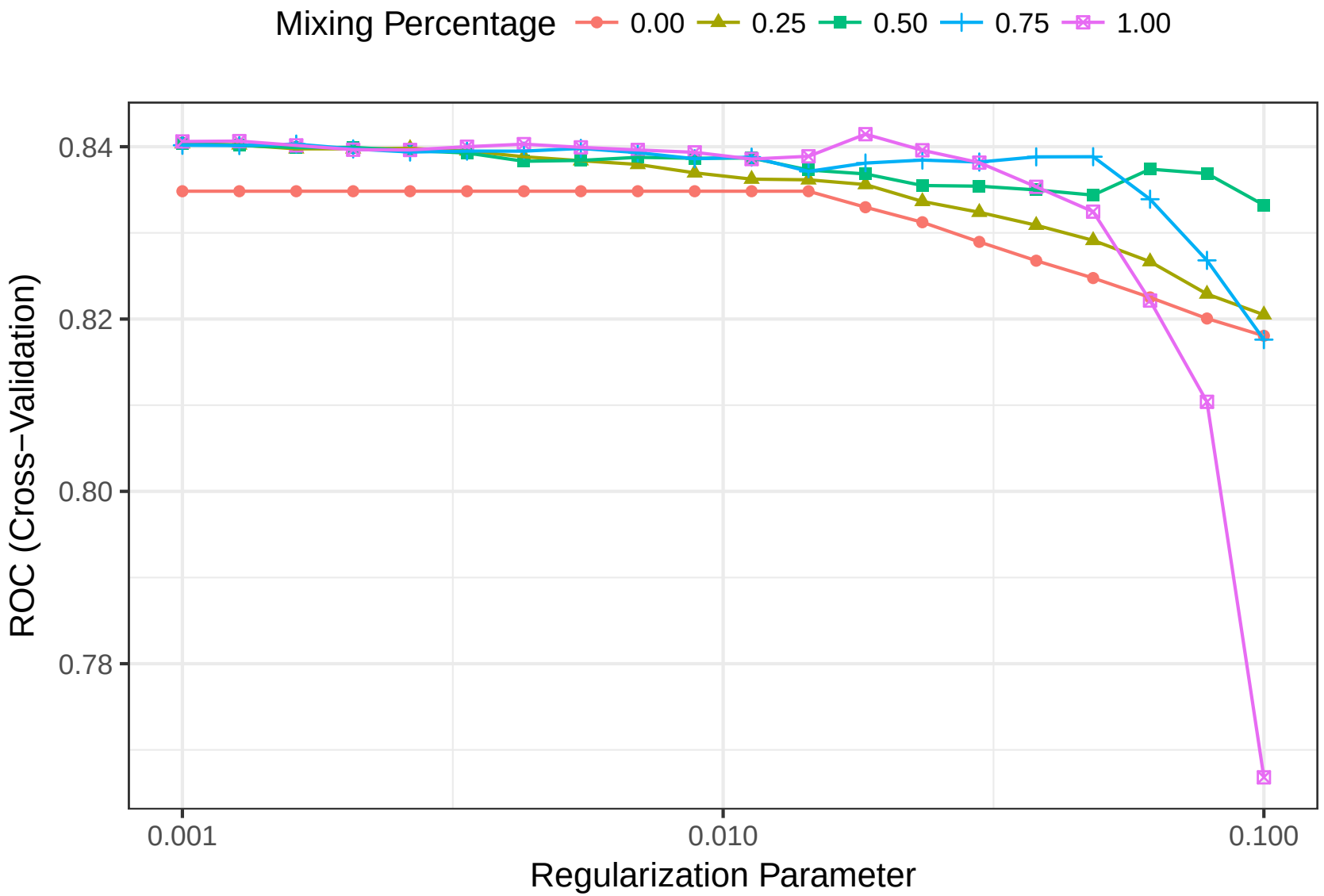
Additionally, the final model fit (i.e. the model with the best resampling results) is in a sub–object called `finalModel`.

So in our example, `glmn_tune` is of class `train` and the object `glmn_tune$finalModel` is of class `glmnet`.

Let's look at what the plot method does.

Resampled ROC Profile

```
ggplot(glmn_tune) + scale_x_log10()
```



Predicting New Samples

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

- `predict(glmn_tune$finalModel, newdata, type = "class")` (**but please don't do this**)
- `predict(glmn_tune, newdata)`

The first method uses `predict.glmnet`, but is not preferred if using `train` and **will give wrong answers**.

`predict.train` does the same thing, but automatically accounts for the centering and scaling.

Test Set Results

```
> glmn_pred <- predict(glmn_tune, newdata = test_data)
> confusionMatrix(glmn_pred, test_data$purch)

Confusion Matrix and Statistics

          Reference
Prediction yes  no
       yes 206  59
       no   44  91

               Accuracy : 0.742
                 95% CI : (0.697, 0.785)
    No Information Rate : 0.625
    P-Value [Acc > NIR] : 4.14e-07

                  Kappa : 0.439
 Mcnemar's Test P-Value : 0.168

            Sensitivity : 0.824
            Specificity : 0.607
         Pos Pred Value : 0.777
         Neg Pred Value : 0.674
             Prevalence : 0.625
         Detection Rate : 0.515
   Detection Prevalence : 0.662
      Balanced Accuracy : 0.715

       'Positive' Class : yes
```

Predicting Class Probabilities

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

```
> glmn_probs <- predict(glmn_tune, newdata = test_data, type = "prob")
> head(glmn_probs, n = 4)
```

```
    yes    no
1 0.131 0.869
2 0.759 0.241
3 0.585 0.415
4 0.423 0.577
```

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.

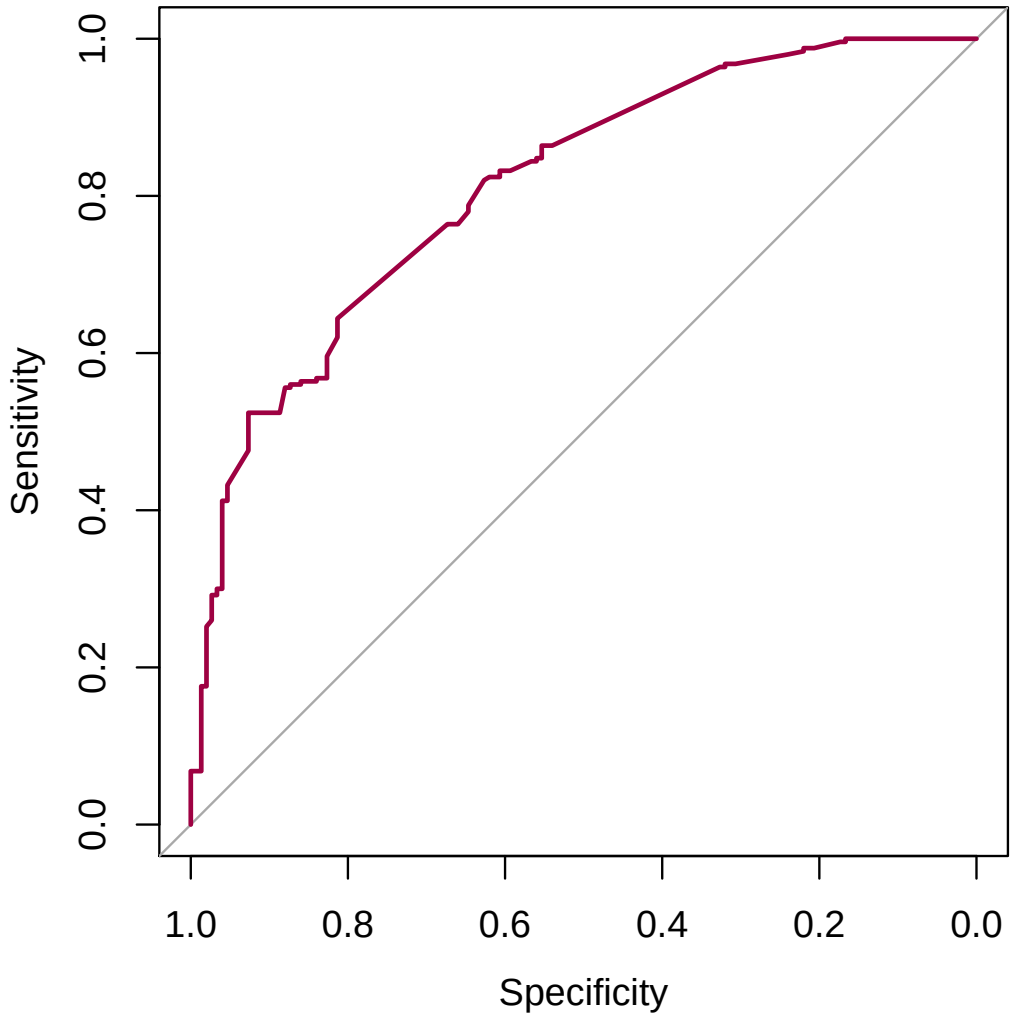
```
> library(pROC)
> ## The roc function assumes the *second* level is the one of
> ## interest, so we use the 'levels' argument to change the order.
> glmn_roc <- roc(response = test_data$purch, predictor = glmn_probs[, "yes"],
+                 levels = rev(levels(test_data$purch)))
```

glmnet also has an `roc` function. To get the right one, we call it using its namespace:

```
> pROC::auc(glmn_roc)
Area under the curve: 0.812
```

glmnet ROC Curve

```
plot(glmn_roc)
```



Variable Importance for the glmnet Model

Since this a fairy simple model, the absolute value of the slope coefficients can be used to measure how much each predictor affects the model.

Note that, for any model with `alpha > 0`, the L_1 penalty may set some coefficients to zero.

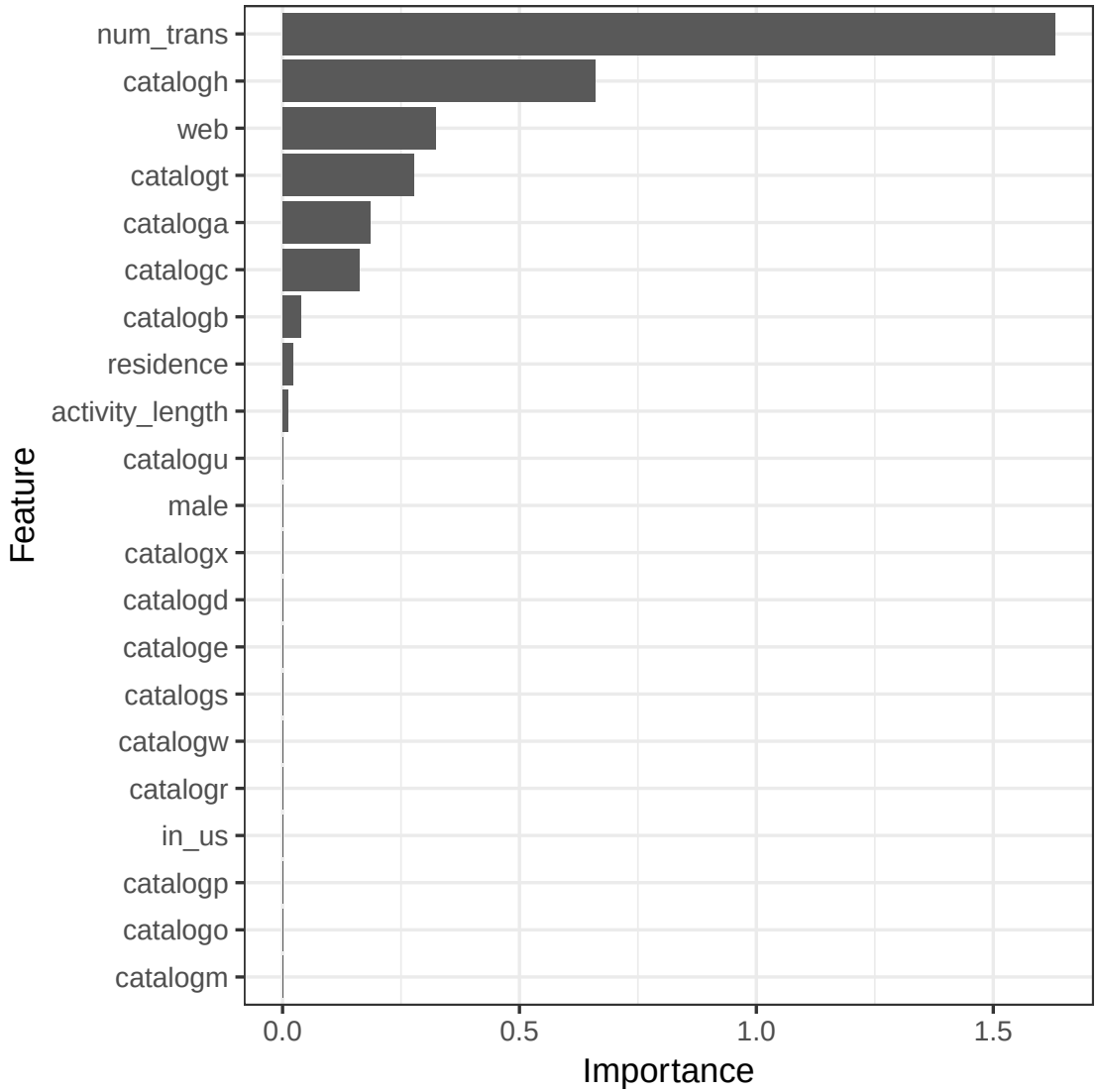
In this way, glmnet can conduct feature selection during model fitting.

This can be measured using caret's `varImp` function and the non-zero predictors are:

```
> predictors(glmn_tune)

 [1] "num_trans"       "web"             "residence"       "cataloga"
 [5] "catalogb"        "catalogc"        "catalogh"        "catalogt"
 [9] "catalogu"        "activity_length"
```

glmnet Variable Importance

```
> ggplot(varImp(glmn_tune, scale = FALSE))
```



Classification Trees

APM Chapter 14

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

There are many tree–based packages in **R**. The main package for fitting single trees are rpart, RWeka, 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(purch ~ .,
+                 data = train_data,
+                 control = rpart.control(maxdepth = 2))
> rpart1
```

```
n= 1202

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

1) root 1202 452 yes (0.624 0.376)
  2) activity_length>=93.5 320  24 yes (0.925 0.075) *
  3) activity_length< 93.5 882 428 yes (0.515 0.485)
    6) catalog=a,d,e,m,p,r,s,t,u,w,x 695 280 yes (0.597 0.403) *
    7) catalog=other,b,c,h 187  39 no (0.209 0.791) *
```

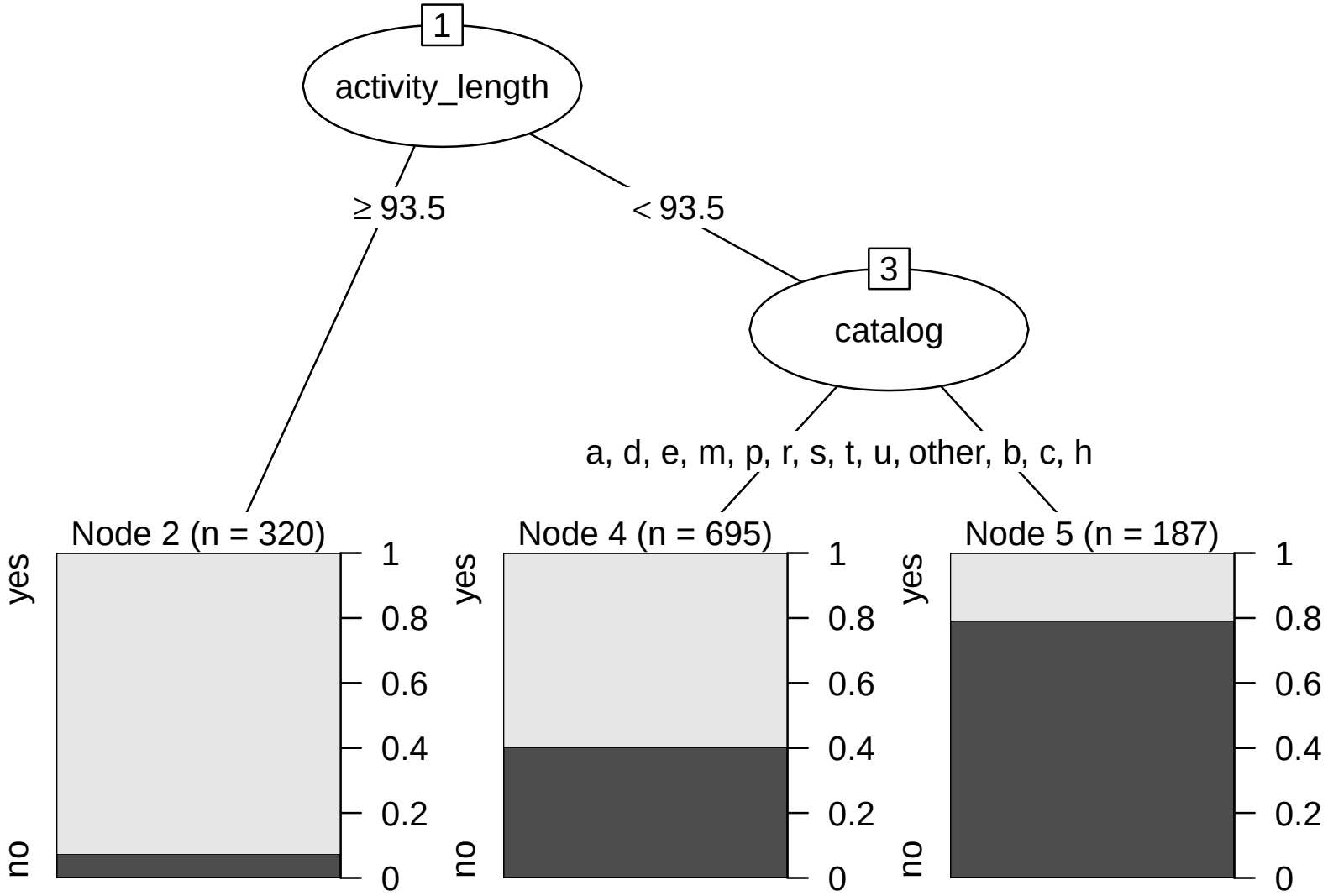
Visualizing the Tree

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

The partykit package 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:

```
> library(partykit)
> rpart1_plot <- as.party(rpart1)
> ## plot(rpart1_plot)
```

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.

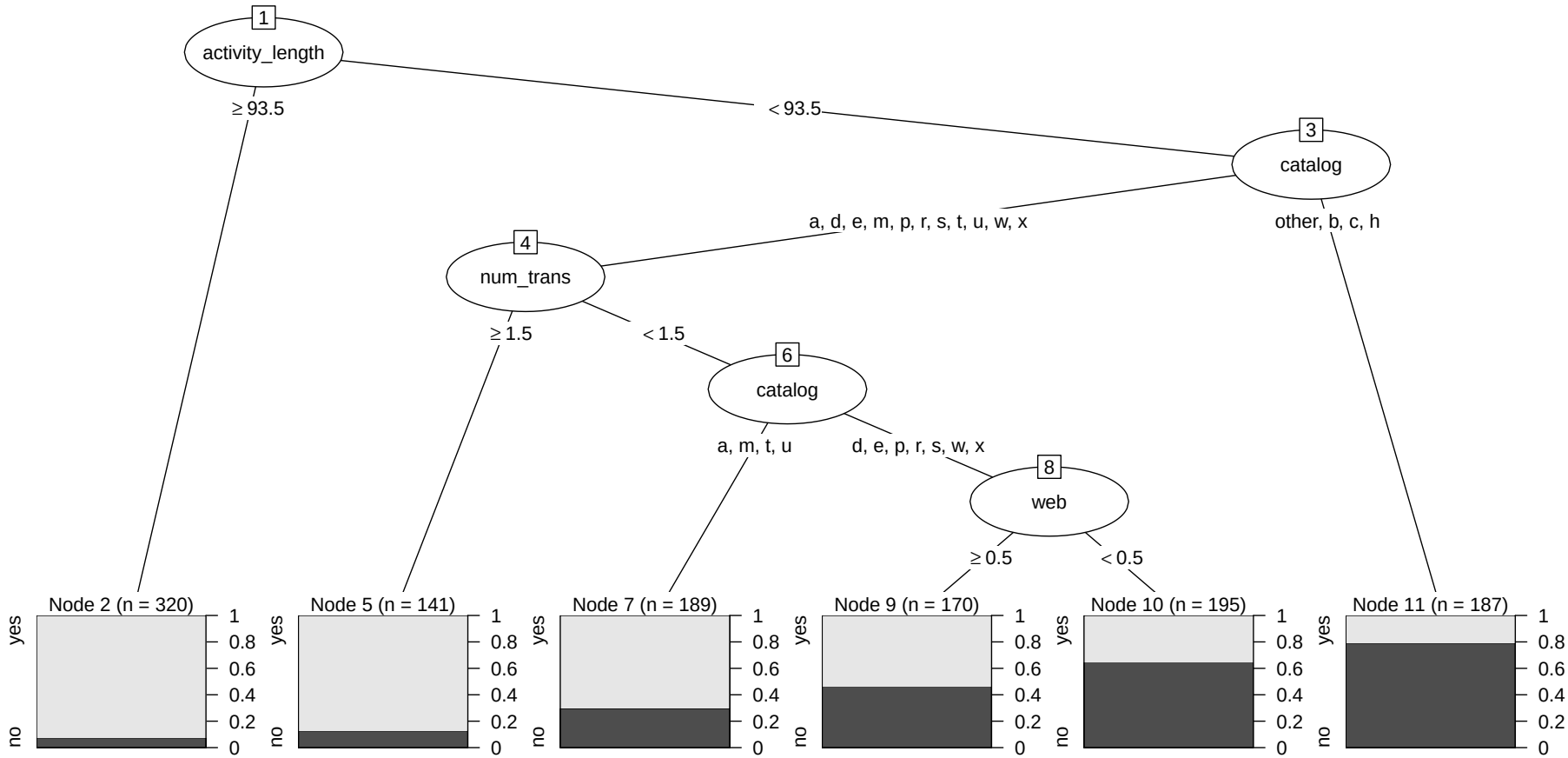
By default, `rpart` will conduct as many splits as possible, then use 10–fold cross–validation to prune the tree.

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.

The Final Tree

```
> rpart_full <- rpart(purch ~ ., data = train_data)
> rpart_full

n= 1202

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

 1) root 1202 452 yes (0.624 0.376)
   2) activity_length>=93.5 320  24 yes (0.925 0.075) *
   3) activity_length< 93.5 882 428 yes (0.515 0.485)
     6) catalog=a,d,e,m,p,r,s,t,u,w,x 695 280 yes (0.597 0.403)
      12) num_trans>=1.5 141  18 yes (0.872 0.128) *
      13) num_trans< 1.5 554 262 yes (0.527 0.473)
        26) catalog=a,m,t,u 189  56 yes (0.704 0.296) *
        27) catalog=d,e,p,r,s,w,x 365 159 no (0.436 0.564)
          54) web>=0.5 170  79 yes (0.535 0.465) *
          55) web< 0.5 195  68 no (0.349 0.651) *
     7) catalog=other,b,c,h 187  39 no (0.209 0.791) *
```

The Final rpart Tree



Test Set Results

```
> rpart_pred <- predict(rpart_full, newdata = test_data, type = "class")
> confusionMatrix(data = rpart_pred, reference = test_data$purch)   # requires 2 factor vectors

Confusion Matrix and Statistics

          Reference
Prediction yes  no
       yes 211  61
       no   39  89

               Accuracy : 0.75
                 95% CI : (0.705, 0.792)
    No Information Rate : 0.625
    P-Value [Acc > NIR] : 7.24e-08

                  Kappa : 0.451
 Mcnemar's Test P-Value : 0.0357

            Sensitivity : 0.844
            Specificity : 0.593
         Pos Pred Value : 0.776
         Neg Pred Value : 0.695
             Prevalence : 0.625
         Detection Rate : 0.527
   Detection Prevalence : 0.680
      Balanced Accuracy : 0.719

       'Positive' Class : yes
```

Creating the ROC Curve

We do the same as before:

```
> class_probs <- predict(rpart_full, newdata = test_data)
> head(class_probs, 3)
```

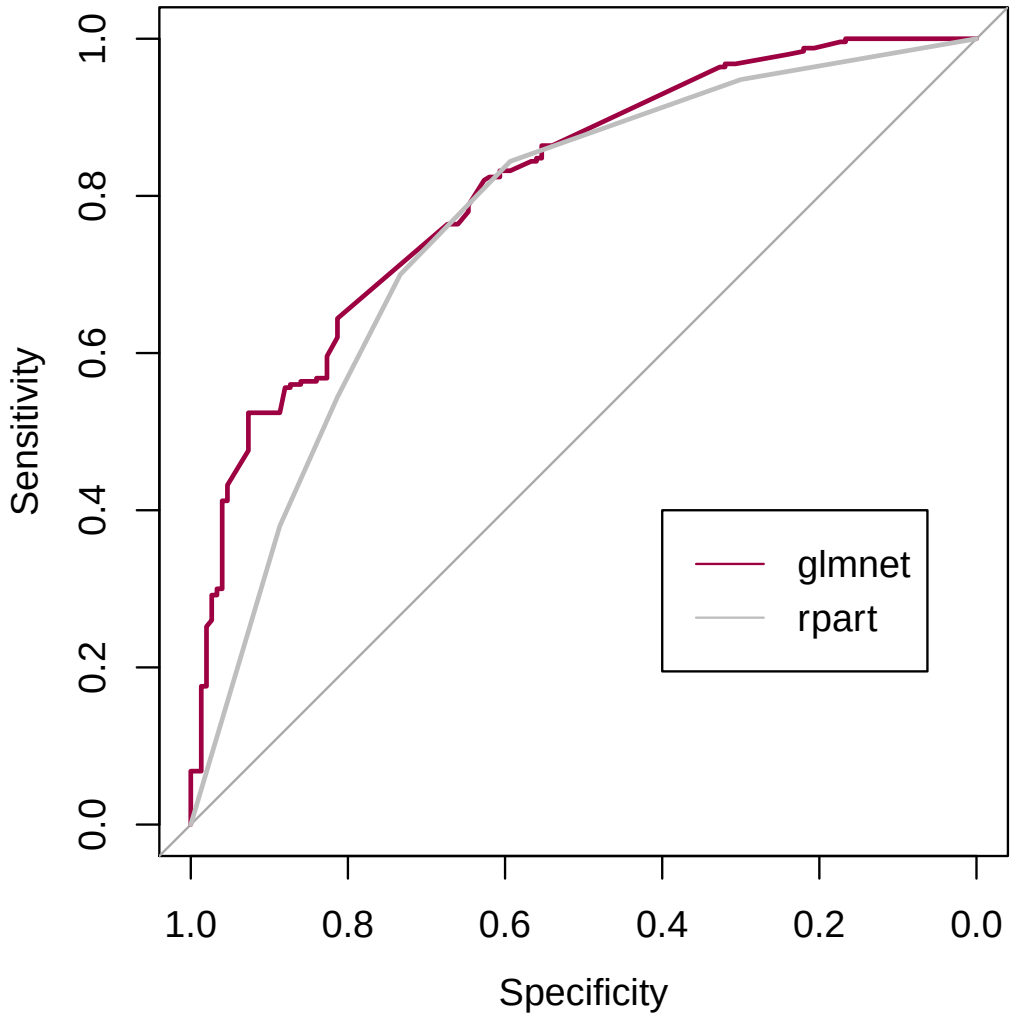
```
     yes    no
12 0.209 0.791
16 0.704 0.296
26 0.925 0.075
```

```
> library(pROC)
> rpart_roc <- roc(response = test_data$purch, predictor = class_probs[, "yes"],
+                  levels = rev(levels(test_data$purch)))
> ## Get the area under the ROC curve
> pROC::auc(rpart_roc)
```

```
Area under the curve: 0.768
```

Classification Tree ROC Curve

```
plot(rpart_roc)
```



Hands–On Break 1

Let's take a minute and make sure that everyone has been able to fit the classification tree model code shown so far.

Your assignment is to refit the `rpart` model using the `rpart` function but modifying the number of samples that are required to make a split. (Hint: see `rpart.control`)

- did performance change?
- is there a difference in the tree depth? Use this function to estimate the number of terminal nodes:

```
num_leaves <- function(x) {
  output <- capture.output(print(x))
  length(grep("\\*$", output))
}
num_leaves(rpart_full)
```

```
[1] 6
```

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.
o The point above 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.

Boosted Trees

APM Sect 14.5

Boosted Trees (Original " adaBoost" Algorithm)

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 adaBoost 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.

Statistical Approaches to Boosting

The original algorithm was developed outside of statistical theory.

Statisticians discovered that the basic boosting algorithm was very similar to gradient–based steepest decent methods, such as Newton's method.

Based on their observations, a new class of boosted tree models were proposed and are generally referred to as "gradient boosting" methods.

Boosted Trees Parameters

Most implementations of boosting have three tuning parameters:

- number of iterations (i.e. trees)
- complexity of the tree (i.e. number of splits)
- learning rate (aka. "shrinkage"): how quickly the algorithm adapts
- the minimum number of samples in a terminal node

Boosting functions for trees in R: `xgboost` in the `xgboost` package, `ada` in `ada`, `blackboost` in `mboost`, `C5.0` in the `C50` package, `gbm` in the `gbm` package, and many others

Using the xgboost Package

The xgboost package can be used to fit the model:

```
> library(xgboost)
> # Requires its own data structure
> train_object <- xgb.DMatrix(train_dummies,
+                             label = ifelse(train_data$purch == "yes", 1, 0))
>
> set.seed(10)
> mod <- xgb.train(data = train_object,
+                  nrounds = 50,                    # Boosting iterations
+                  max_depth = 2,                   # How many splits in each tree
+                  eta = 0.01,                      # Learning rate
+                  silent = 1, nthread = 1,
+                  objective = "binary:logistic") # For classification
```

xgboost Predictions

The predict method requires the same data format for new samples.

Also, it does not predict the actual class. We'll get the class probability and do the conversion.

```
> test_object <- xgb.DMatrix(test_dummies)
> xgb_pred <- predict(mod, newdata = test_object)
> head(xgb_pred)

[1] 0.362 0.546 0.626 0.487 0.673 0.525

> xgb_pred <- factor(ifelse(xgb_pred > .5, "yes", "no"),
+                    levels = c("yes", "no"))
> head(xgb_pred)

[1] no  yes yes no  yes yes
Levels: yes no
```

Test Set Results

```
> confusionMatrix(xgb_pred, test_data$purch)

Confusion Matrix and Statistics

          Reference
Prediction yes  no
       yes 207  65
       no   43  85

               Accuracy : 0.73
                 95% CI : (0.684, 0.773)
    No Information Rate : 0.625
    P-Value [Acc > NIR] : 5.87e-06

                  Kappa : 0.407
 Mcnemar's Test P-Value : 0.0433

            Sensitivity : 0.828
            Specificity : 0.567
         Pos Pred Value : 0.761
         Neg Pred Value : 0.664
             Prevalence : 0.625
         Detection Rate : 0.517
   Detection Prevalence : 0.680
      Balanced Accuracy : 0.697

       'Positive' Class : yes
```

Model Tuning using `train`

We can fit a series of boosted trees with different specifications and use resampling to understand which one is most appropriate.

For example, we can define a grid of 152 tuning parameter combinations:

- number of trees: between 10 to 1000
- tree depth: 1 to 7 by 2
- learning rate: 0.001 and 0.01

In **R**:

```
> xgb_grid <- expand.grid(
+   max_depth = seq(1, 7, by = 2),
+   nrounds = seq(100, 1000, by = 50),
+   eta = c(0.01, 0.1),
+   # Other parameters. See ?xgboost
+   gamma = 0,
+   colsample_bytree = 1,
+   min_child_weight = 1,
+   subsample = 0.5
+ )
```

We can use this grid to define exactly what models are evaluated by `train`. A list of model parameter names can be found using `?models`.

Model Tuning using `train`

```
> set.seed(1735)
> xgb_tune <- train(
+   purch ~ ., data = train_data,
+   method = "xgbTree",
+   # Use a custom grid of tuning parameters
+   tuneGrid = xgb_grid,
+   trControl = cv_ctrl,
+   metric = "ROC",
+   # Remember the 'three dots' discussed in the bootcamp?
+   # This options is directly passed to the xgb.train function.
+   silent = 1
+ )
```

Digression – 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 and this blog post.

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.

foreach and caret

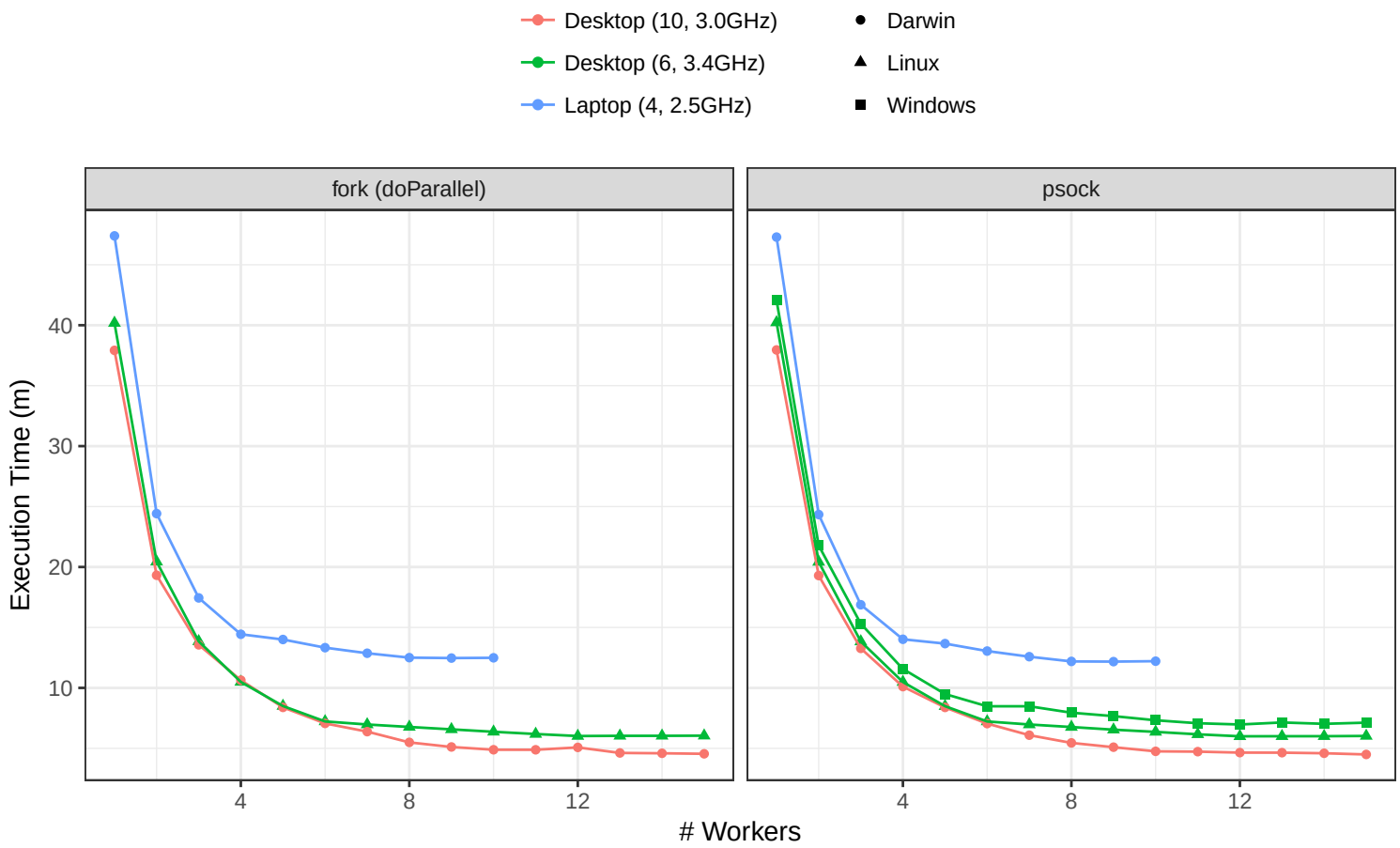
To use parallel processing in caret, no changes are needed when calling `train`.

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

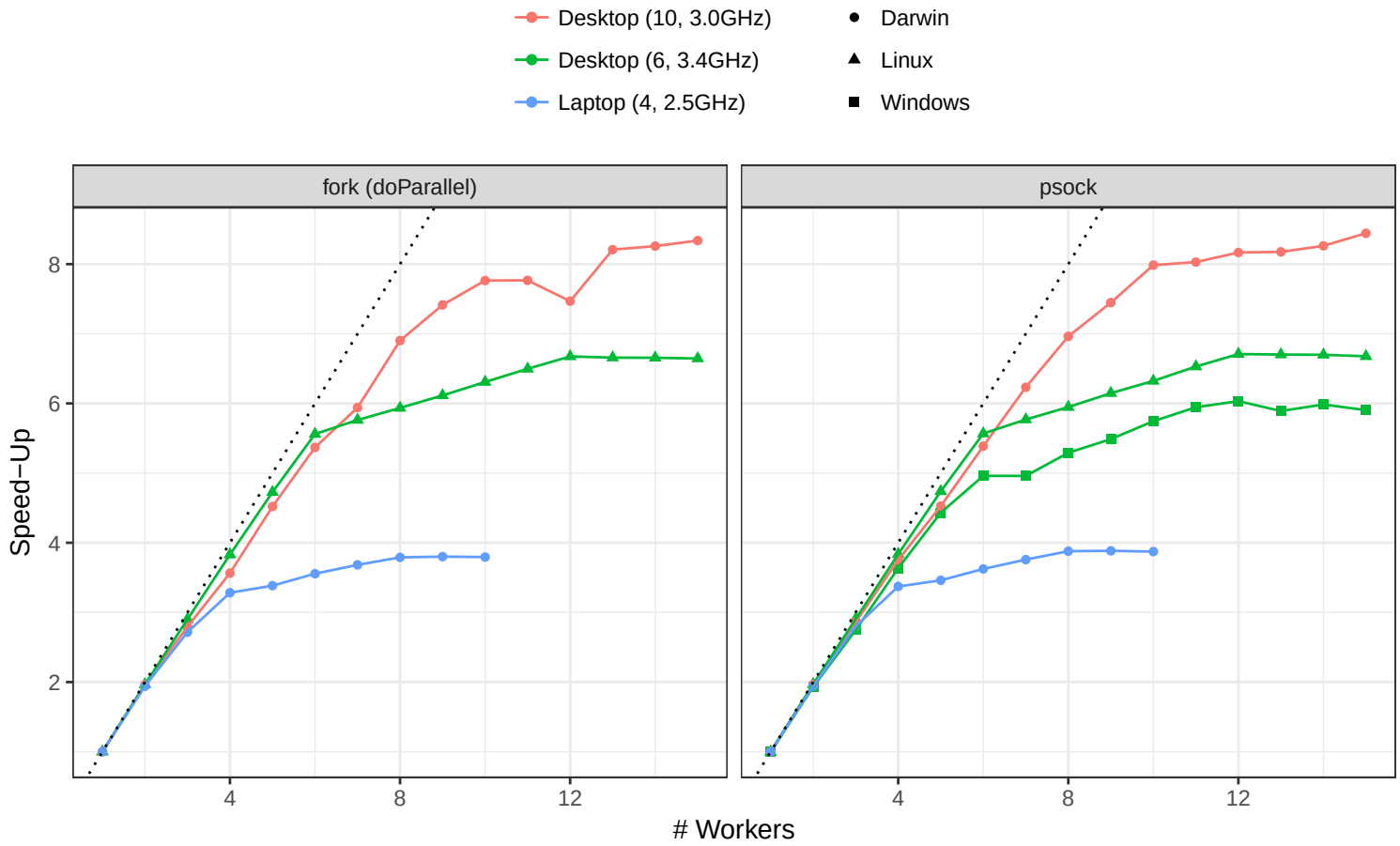
```
> library(doMC)           # on unix, linux or OS X
> ## library(doParallel) # windows and others
> registerDoMC(cores = 2)
>
> # or
>
> library(doParallel)
> cl <- makePSOCKcluster(parallel::detectCores(logical = FALSE))
> registerDoParallel(cl)
```

Training Time (min)

Example for a simulated data set.



Speed–Up

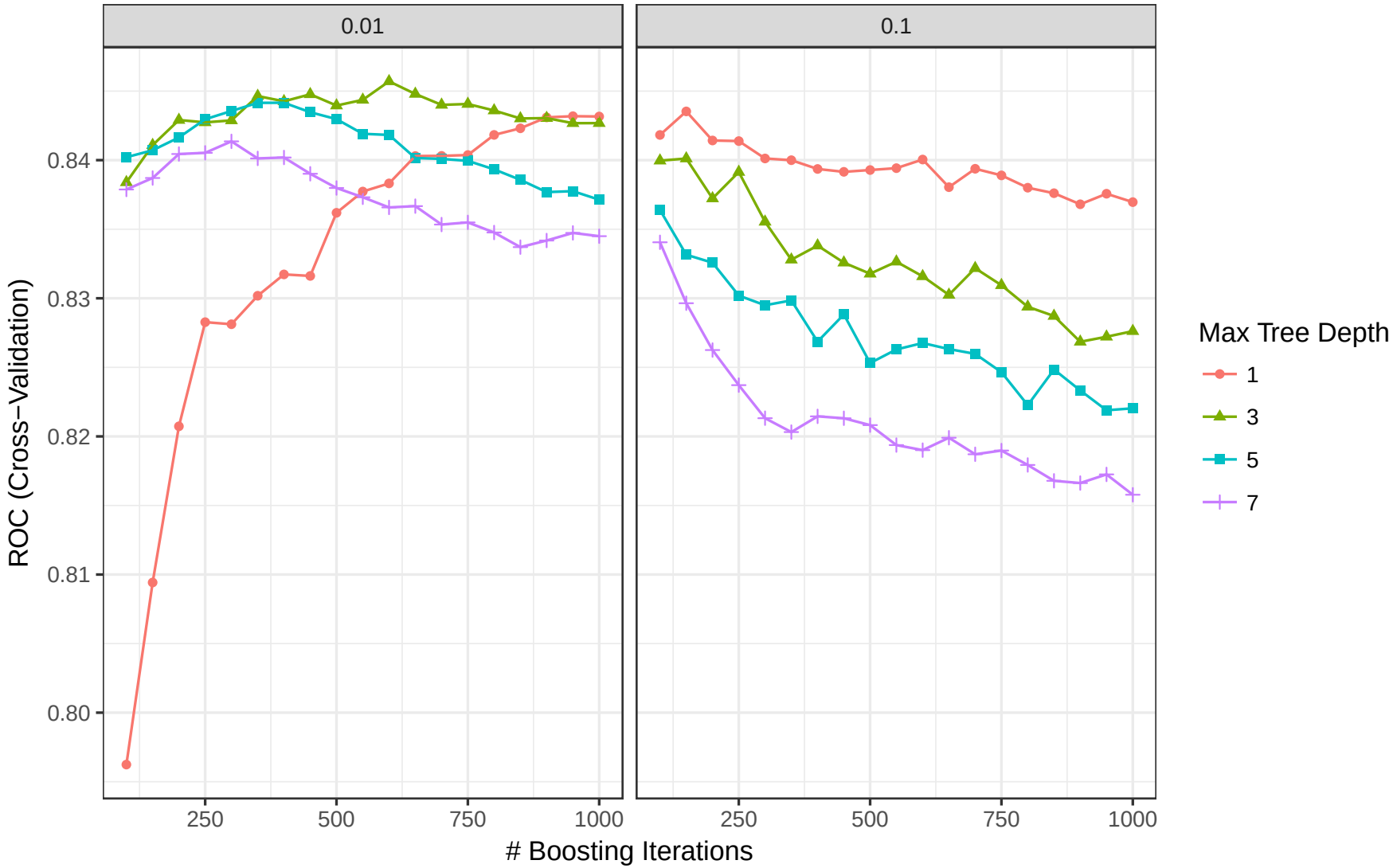

Desktop (10, 3.0GHz)
Desktop (6, 3.4GHz)
Laptop (4, 2.5GHz)
Darwin
Linux
Windows
fork (doParallel)
psock
Speed–Up
Workers
2
4
6
8
4
8
12

Boosted Tree Results

```
> xgb_tune


eXtreme Gradient Boosting

1202 samples
   7 predictor
   2 classes: 'yes', 'no'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 1082, 1082, 1082, 1082, 1081, 1082, ...
Resampling results across tuning parameters:

  eta   max_depth  nrounds  ROC    Sens   Spec
  0.01  1           100     0.796  0.997  0.122
  0.01  1           150     0.809  0.996  0.126
    :   :            :       :      :      :
  0.10  7           950     0.817  0.808  0.640
  0.10  7          1000     0.816  0.808  0.646

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

Tuning parameter 'min_child_weight' was held constant at a value of 1

Tuning parameter 'subsample' was held constant at a value of 0.5
ROC was used to select the optimal model using the largest value.
The final values used for the model were nrounds = 600, max_depth = 3, eta
 = 0.01, gamma = 0, colsample_bytree = 1, min_child_weight = 1 and subsample
 = 0.5.
```

Boosted Tree ROC Profile

```
ggplot(xgb_tune)
```



Test Set Results

```
> xgb_pred <- predict(xgb_tune, newdata = test_data[, predictors]) # Magic!
> confusionMatrix(xgb_pred, test_data$purch)

Confusion Matrix and Statistics

          Reference
Prediction yes  no
       yes 199  55
       no   51  95

               Accuracy : 0.735
                 95% CI : (0.689, 0.778)
    No Information Rate : 0.625
    P-Value [Acc > NIR] : 2.11e-06

                  Kappa : 0.432
 Mcnemar's Test P-Value : 0.771

            Sensitivity : 0.796
            Specificity : 0.633
         Pos Pred Value : 0.783
         Neg Pred Value : 0.651
             Prevalence : 0.625
         Detection Rate : 0.497
   Detection Prevalence : 0.635
      Balanced Accuracy : 0.715

       'Positive' Class : yes
```

xgb Probabilities

```
> xgb_probs <- predict(xgb_tune,
+                      newdata = test_data[, predictors],
+                      type = "prob")
> head(xgb_probs)
```

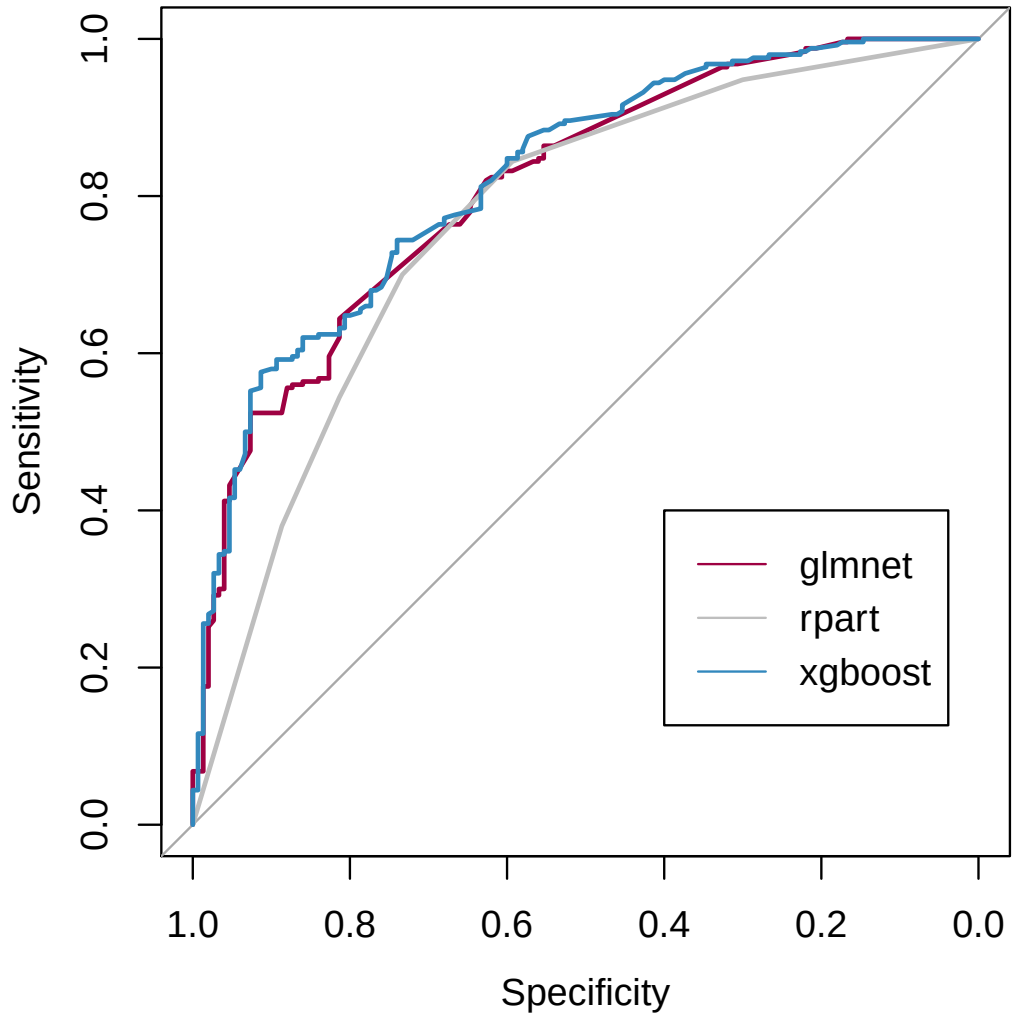
```
     yes    no
1 0.0447 0.955
2 0.7592 0.241
3 0.8151 0.185
4 0.3461 0.654
5 0.8803 0.120
6 0.5490 0.451
```

Boosted Tree ROC Curve

```
> xgb_roc <- roc(response = test_data$purch, predictor = xgb_probs[, "yes"],
+                levels = rev(levels(test_data$purch)))
> pROC::auc(glmn_roc)

Area under the curve: 0.812

> pROC::auc(rpart_roc)

Area under the curve: 0.768

> pROC::auc(xgb_roc)

Area under the curve: 0.823
```

```
> plot(glmn_roc, col = "#9E0142")
> plot(rpart_roc, col = "grey", legacy.axes = FALSE, add = TRUE)
> plot(xgb_roc,   col = "#3288BD", legacy.axes = FALSE, add = TRUE)
> legend(.4, .4, legend = c("glmnet", "rpart", "xgboost"),
+        lty = rep(1, 3),
+        col = c("#9E0142", "grey", "#3288BD"))
```

Boosted Tree ROC Curve



Hands–On Break 2

Use the `varImp` function on the `train` object to see the importance of each of the predictors.

Feature Selection

Feature Selection

There are some predictive models that can be sensitive to non–informative predictors in the model.

For this reason, and the general idea of model parsimony, we might want to eliminate such predictors.

- Some models, such as trees, intrinsically do feature selection during training.
- Others require additional computational approaches.

One such approach is the *wrapper* method that treats feature selection like a optimization routine:

maximize model performance such that a minimum number of predictors are retained

This is *supervised feature selection*.

Feature Selection

There are many optimization routines that can be used to search over the predictor space

- genetic algorithms
- simulated annealing
- forward selection
- backward selection (called recursive feature elimination, or RFE)
- ...

It is very important to understand that feature selection is part of the model building process.

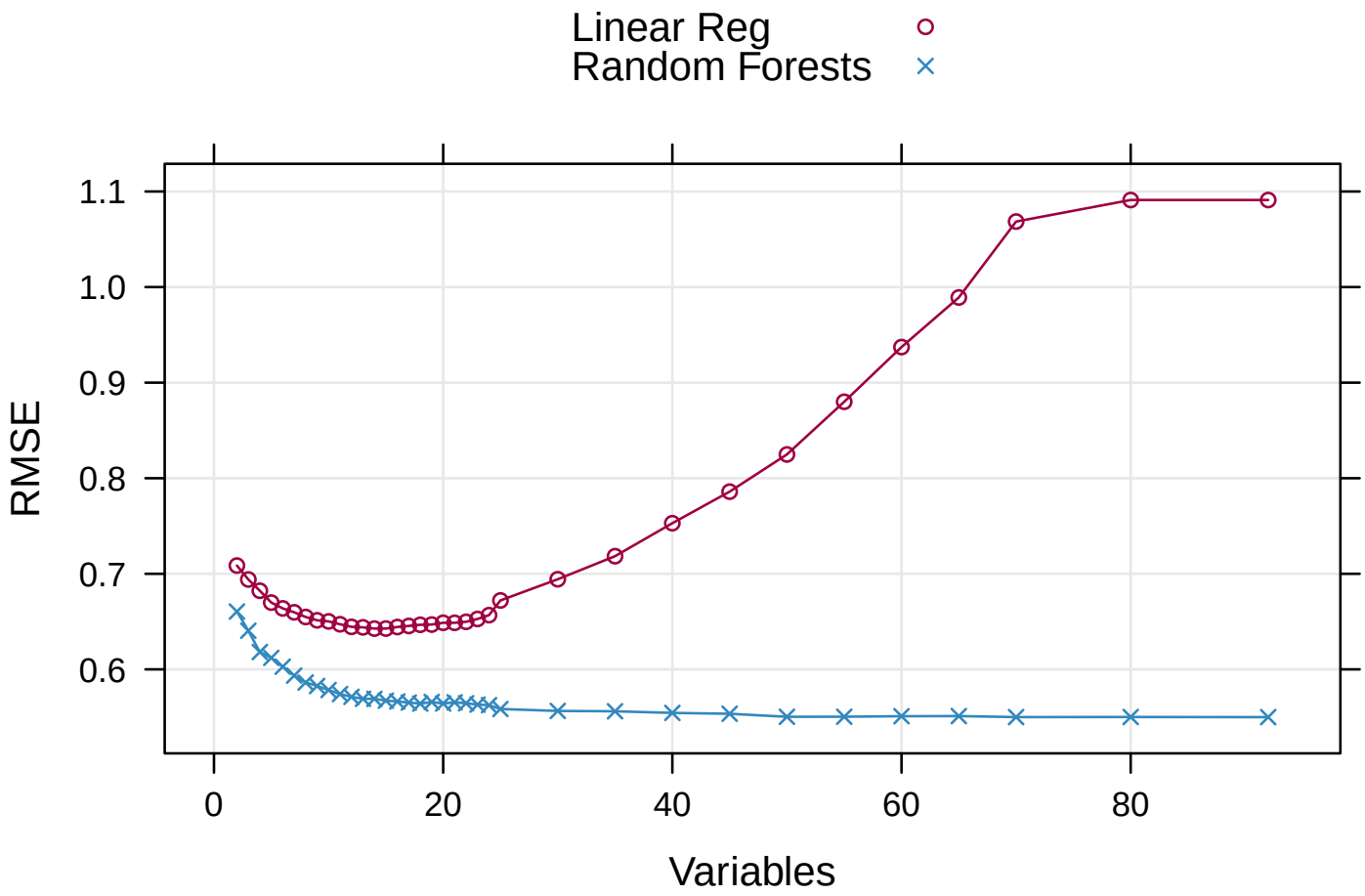
As such, it must be included in the validation process to get honest estimates of model performance.

Ambroise and McLachlan (2002) show an example where RFE was excluded from the resampling process with serious results

Recursive Feature Elimination

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

An Example RFE Profile



Using `rfe`

The `rfe` is structured so that any model or filter can be "plugged–in" to the function.

There are a few modules for specific models in the package (see `?rfeControl` and package website).

We will fit an logistic regression model across different subset sizes to see if removing features helps.

For this model, variable importance is based on the absolute value of the regression coefficients.

Using rfe

caret contains a list of modules called lrFuncs that can be used to define the various steps in the RFE algorithm:

```
> names(lrFuncs)

[1] "summary"    "fit"        "pred"       "rank"       "selectSize"
[6] "selectVar"

> lrFuncs$fit

function (x, y, first, last, ...)
{
    tmp <- if (is.data.frame(x))
        x
    else as.data.frame(x)
    tmp$Class <- y
    glm(Class ~ ., data = tmp, family = "binomial")
}
<environment: namespace:caret>
```

The rfeControl Function

```
> ## No tuning parameters for this model, so avoid an inner resampling loop
> inner_ctrl <- trainControl(method = "none",
+                            classProbs = TRUE,
+                            allowParallel = FALSE)
>
> ## Now the rfe control function. First, to get the ROC value
> ## we change the summary function as we did with train()
> ourFunctions <- lrFuncs
> ourFunctions$summary <- twoClassSummary
>
> rfeCtrl <- rfeControl(functions = ourFunctions,
+                       method = "cv",
+                       verbose = TRUE)
```

The rfe Call

To avoid numerical issues, we use the predictor set with no near zero variance predictors and low correlations.

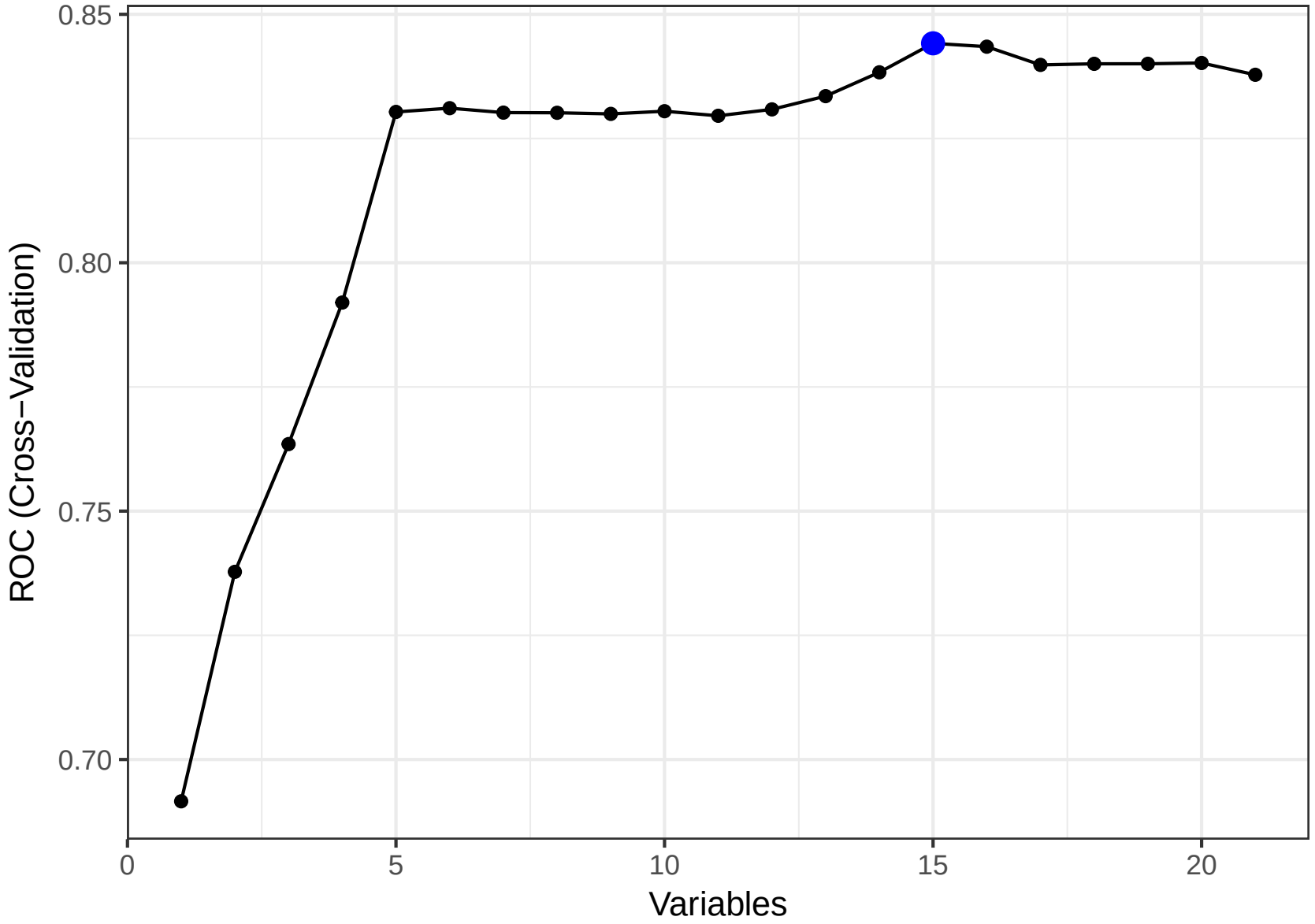
```
> ## We can pass the same options to train() through rfe()
> set.seed(1735)
> logistic_rfe <- rfe(
+   x = train_dummies,
+   y = train_data$purch,
+   sizes = 1:ncol(train_dummies),
+   rfeControl = rfeCtrl,
+   ## now the options that pass to train()
+   method = "glm",
+   preProc = c("center", "scale"),
+   trControl = inner_ctrl,
+   metric = "ROC"
+   )
```

Results

```
> logistic_rfe
```

```
Recursive feature selection

Outer resampling method: Cross-Validated (10 fold)

Resampling performance over subset size:

 Variables   ROC  Sens  Spec  ROCSD SensSD SpecSD Selected
         1 0.692 0.547 0.790 0.0359 0.0603 0.0688
         2 0.738 0.805 0.506 0.0316 0.0394 0.0541
         3 0.763 0.837 0.482 0.0286 0.0486 0.0604
         :   :     :     :      :      :      :
        13 0.834 0.809 0.661 0.0324 0.0281 0.0897
        14 0.838 0.823 0.648 0.0311 0.0339 0.0968
        15 0.844 0.837 0.641 0.0302 0.0365 0.0935        *
        16 0.843 0.835 0.641 0.0328 0.0383 0.0905
        17 0.840 0.841 0.648 0.0312 0.0323 0.0885
        18 0.840 0.844 0.644 0.0313 0.0321 0.0865
        19 0.840 0.843 0.648 0.0310 0.0319 0.0853
        20 0.840 0.843 0.650 0.0313 0.0319 0.0829
        21 0.838 0.843 0.644 0.0332 0.0319 0.0854

The top 5 variables (out of 15):
   num_trans, web, catalog.t, catalog.h, catalog.a
```

Resampling Profile



RFE Test Results

Resampling estimated the area under the ROC curve to be 0.844.

Using the test set:

```
> rfe_test <- predict(logistic_rfe, test_dummies)
> head(rfe_test)

      yes    no pred
12 0.0457 0.954   no
16 0.8264 0.174  yes
26 0.5838 0.416  yes
39 0.3359 0.664   no
45 0.8510 0.149  yes
50 0.6203 0.380  yes
```

```
> roc(response = test_data$purch, predictor = rfe_test$yes,
+     levels = rev(levels(test_data$purch)))


Call:
roc.default(response = test_data$purch, predictor = rfe_test$yes,     levels = rev(

Data: rfe_test$yes in 150 controls (test_data$purch no) < 250 cases (test_data$purc
Area under the curve: 0.813
```

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 resampling 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.

The tidyposterior package has a more modern approach based on Benavoli et al (2017).

Collecting Results With `resamples`

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

```
> cv_values <- resamples(
+   list(glmnet = glmn_tune,
+        xgboost = xgb_tune, logistic = logistic_rfe)
+  )
```

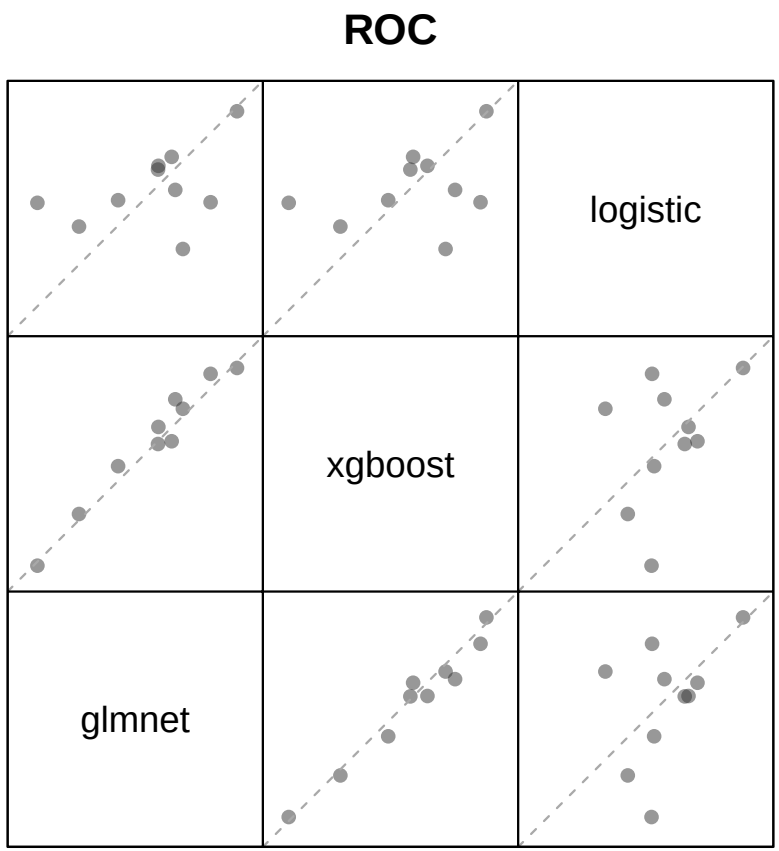
Collecting Results With resamples

```
> summary(cv_values, metric = "ROC")


Call:
summary.resamples(object = cv_values, metric = "ROC")

Models: glmnet, xgboost, logistic
Number of resamples: 10

ROC
          Min. 1st Qu. Median  Mean 3rd Qu.  Max. NA's
glmnet   0.751   0.822  0.851 0.841   0.863 0.907    0
xgboost  0.748   0.830  0.851 0.846   0.876 0.903    0
logistic 0.796   0.832  0.838 0.844   0.860 0.904    0

> glmn_tune$times$everything[3]/60

elapsed
 0.0403

> xgb_tune$times$everything[3]/60

elapsed
  0.435
```

Visualizing the Resamples

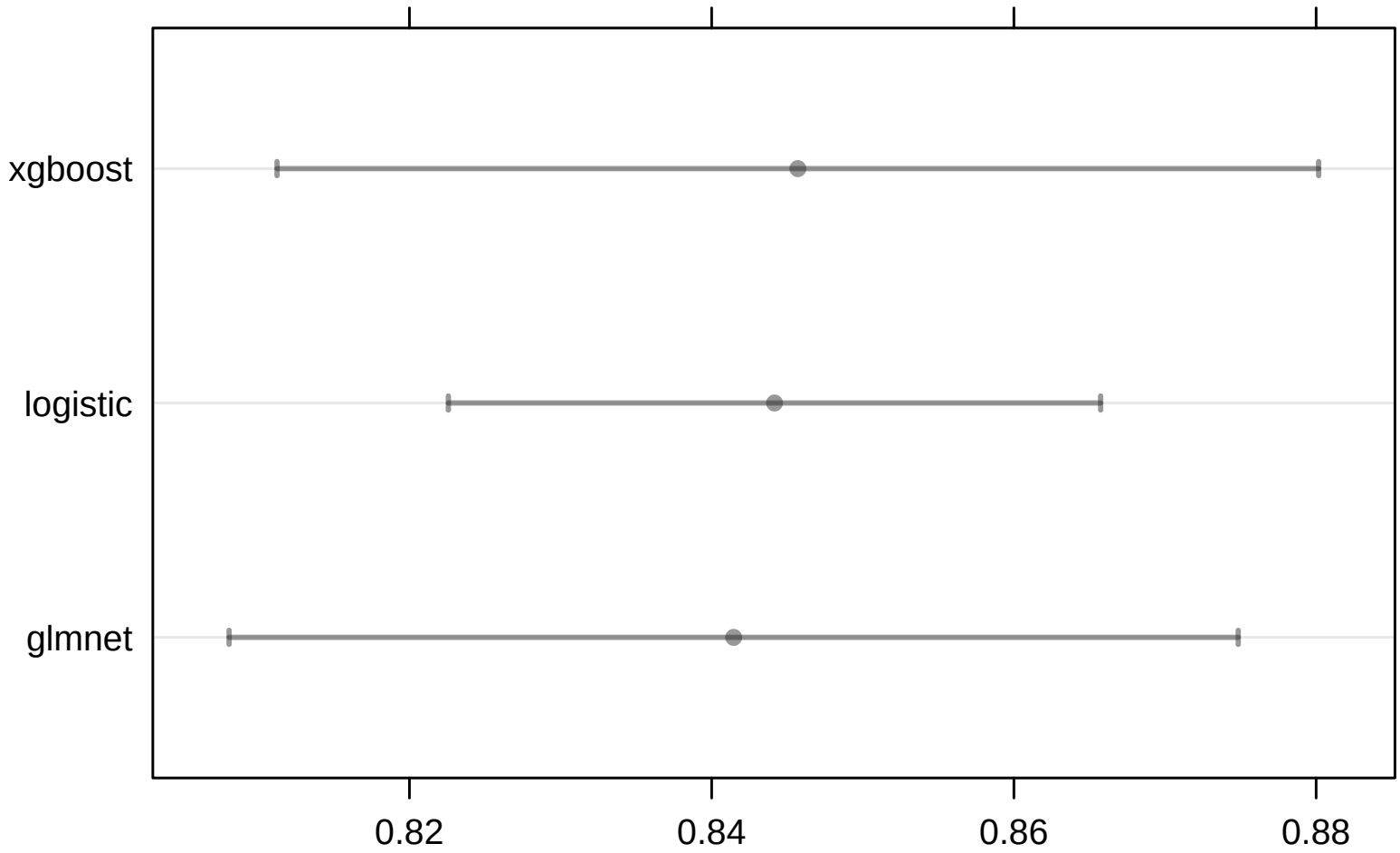
There are a number of lattice plot methods to display the results: `bwplot`, `dotplot`, `parallelplot`, `xyplot`, `splom`. For example:

```
> splom(cv_values, metric = "ROC", pscales = 0)
```



Visualizing the Resamples

```
> dotplot(cv_values, metric = "ROC")
```



Comparing Models

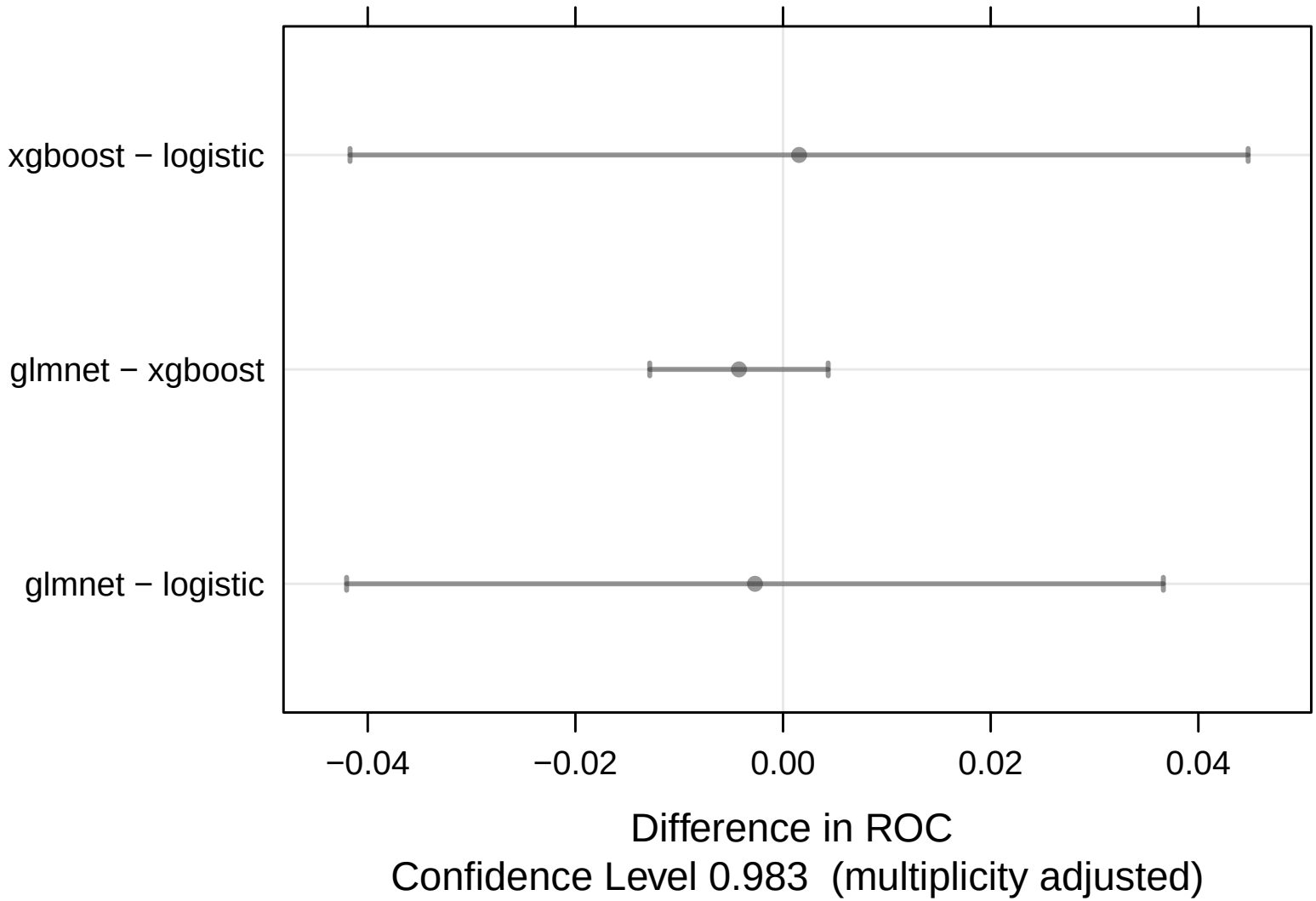
We can also test to see if there are differences between the models:

```
> roc_diffs <- diff(cv_values, metric = "ROC")
> summary(roc_diffs)


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

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

ROC
         glmnet xgboost  logistic
glmnet           -0.00425 -0.00270
xgboost  0.543             0.00155
logistic 1.000  1.000
```

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

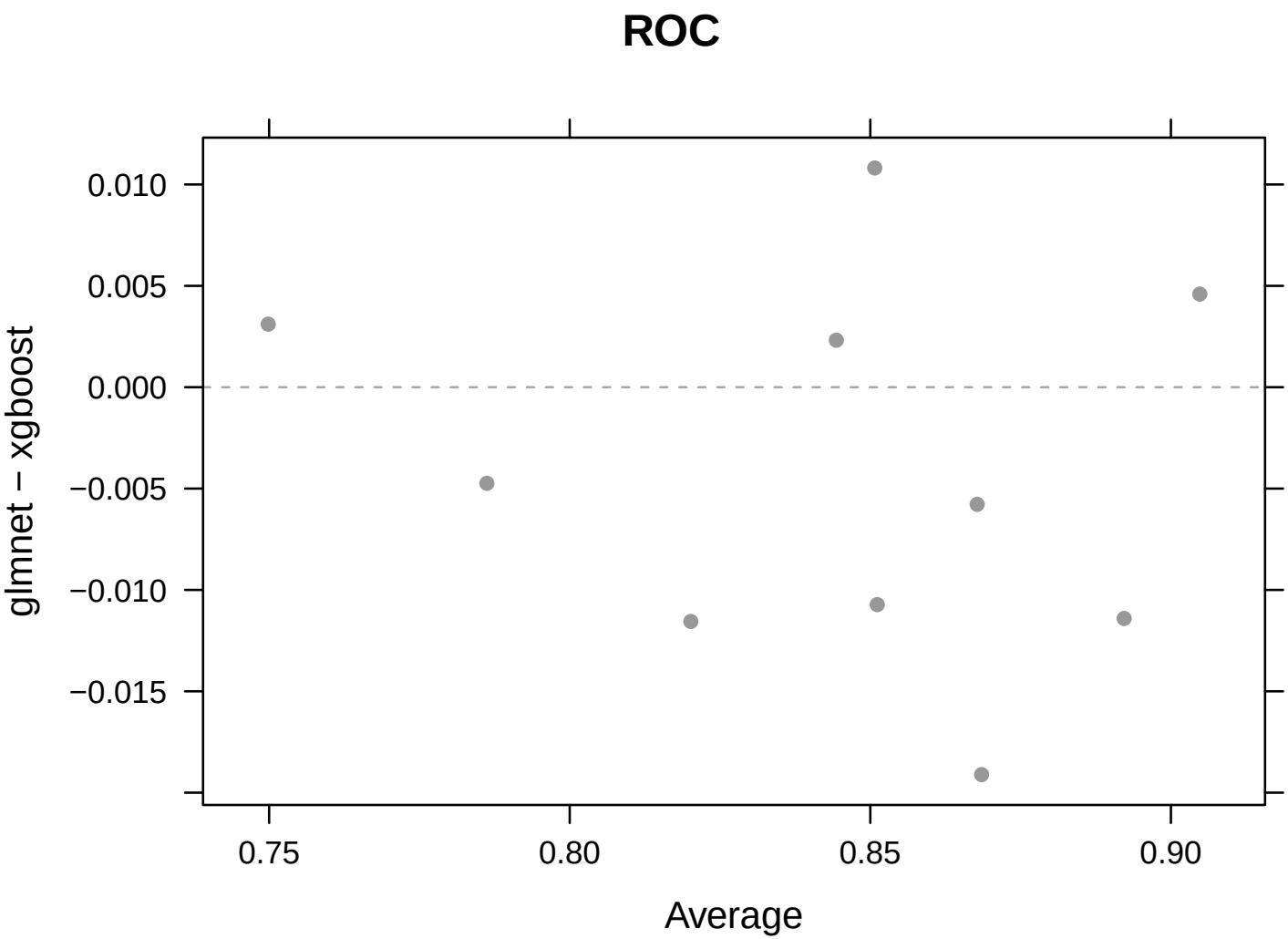
Visualizing the Differences

```
> dotplot(roc_diffs, metric = "ROC")
```



A "Bland-Altman" Plot Comparing Two Models

```
> xyplot(cv_values, metric = "ROC", models = c("glmnet", "xgboost"),
+        what = "BlandAltman")
```



Backup/Extra Slides

Other Functions and Classes

- `bag`: a general bagging function
- `upSample`, `downSample`: functions for class imbalances
- `predictors`: class for determining which predictors are included in the prediction equations (e.g. `rpart`, `earth`, `lars` models)
- `varImp`: classes for assessing the aggregate effect of a predictor on the model equations
- `lift`: creating lift/gain charts

Other Functions and Classes

- `knnreg`: nearest–neighbor regression
- `plsda`, `splsda`: PLS discriminant analysis
- `icr`: independent component regression
- `pcaNNet`: `nnet:::nnet` with automatic PCA pre–processing step
- `bagEarth`, `bagFDA`: bagging with MARS and FDA models
- `maxDissim`: a function for maximum dissimilarity sampling
- `rfe`, `sbf`, `gabf`, `sabf`: classes/frameworks for recursive feature selection (RFE), univariate filters, genetic algrithms, simulated annealing feature selection methods
- `featurePlot`: a wrapper for several `lattice` functions

Example: Encoding Time and Date Data

Some applications have date or time data as predictors. How should we encode this?

- numerical day of the year along with the year?
- categorical or ordinal factors for the day of the week, week, month, or season, etc?
- number of days from some reference date?

The answer depends on the type of model and the nature of the data.

Example: Encoding Time and Date Data

I have found the lubridate package to be invaluable in these cases.

Let's load some example dates from an existing RData file:

```
> day_values <- c("2015-05-10", "1970-11-04", "2002-03-04", "2006-01-13")
> class(day_values)

[1] "character"

> library(lubridate)
> days <- ymd(day_values)
> str(days)

 Date[1:4], format: "2015-05-10" "1970-11-04" "2002-03-04" "2006-01-13"
```

Example: Encoding Time and Date Data

```
> day_of_week <- wday(days, label = TRUE)
> day_of_week

[1] Sun Wed Mon Fri
Levels: Sun < Mon < Tue < Wed < Thu < Fri < Sat

> year(days)

[1] 2015 1970 2002 2006

> week(days)

[1] 19 44  9  2

> month(days, label = TRUE)

[1] May Nov Mar Jan
12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec

> yday(days)

[1] 130 308  63  13
```

Lift Curve (aka "Gain Chart")

Lift curves are graphs that show the relationship between the percentage or events found versus the number of items evaluated.

Suppose a data set has an overall event rate of π.

If a model is completely non–informative, we would expect to capture π events in a sample.

If a model is predictive, the lift would be the ratio of samples found to π, i.e. a lift of 2 means that the model correctly predicted 2π events.

Lift Curve

Operationally:

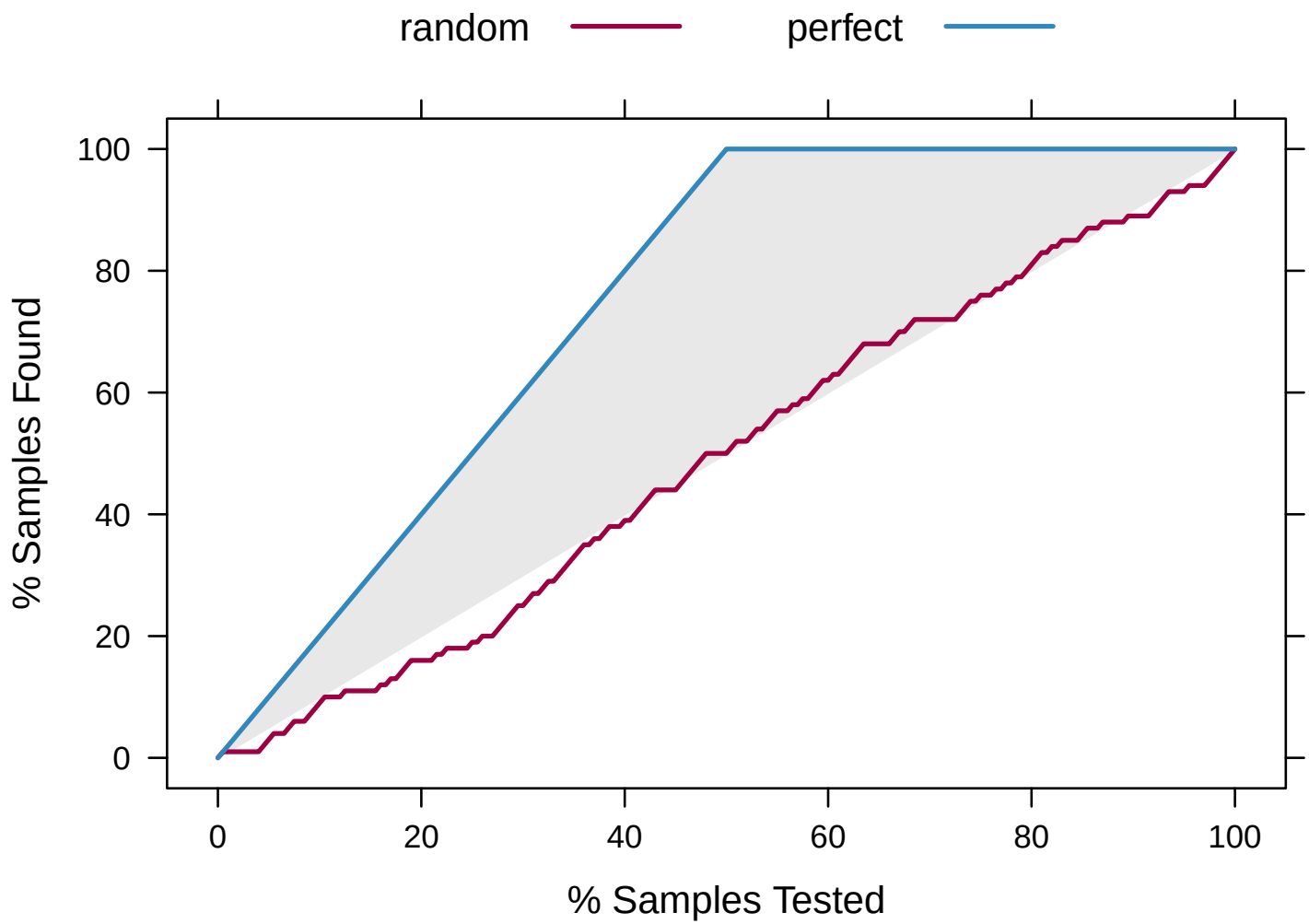
1. determine the baseline event rate in a sample (requires the true outcomes)
2. score the items using some quantitative measure of confidence, such as a class probability
3. sort the samples from most likely to be events to least likely
4. for each unique class probability, determine the percentage of events in the data with probability values less than the probability cutoff
5. the lift is the percentage of events in the group versus the overall percentage of events

Lift Curve

With a good model there should be more "yes" values at the top of the list:

```
     Purchase Prob Customer Truth
 73           100%       73    no
129           100%      129   yes
159           100%      159   yes
160           100%      160   yes
180           100%      180   yes
189           100%      189   yes
             :       :           :
     Purchase Prob Customer Truth
 684             6%      684    no
 784             6%      784    no
1248             6%     1248    no
1620             6%     1620    no
1637             6%     1637    no
1695             6%     1695    no
```

Example Lift Curves



Lift Charts

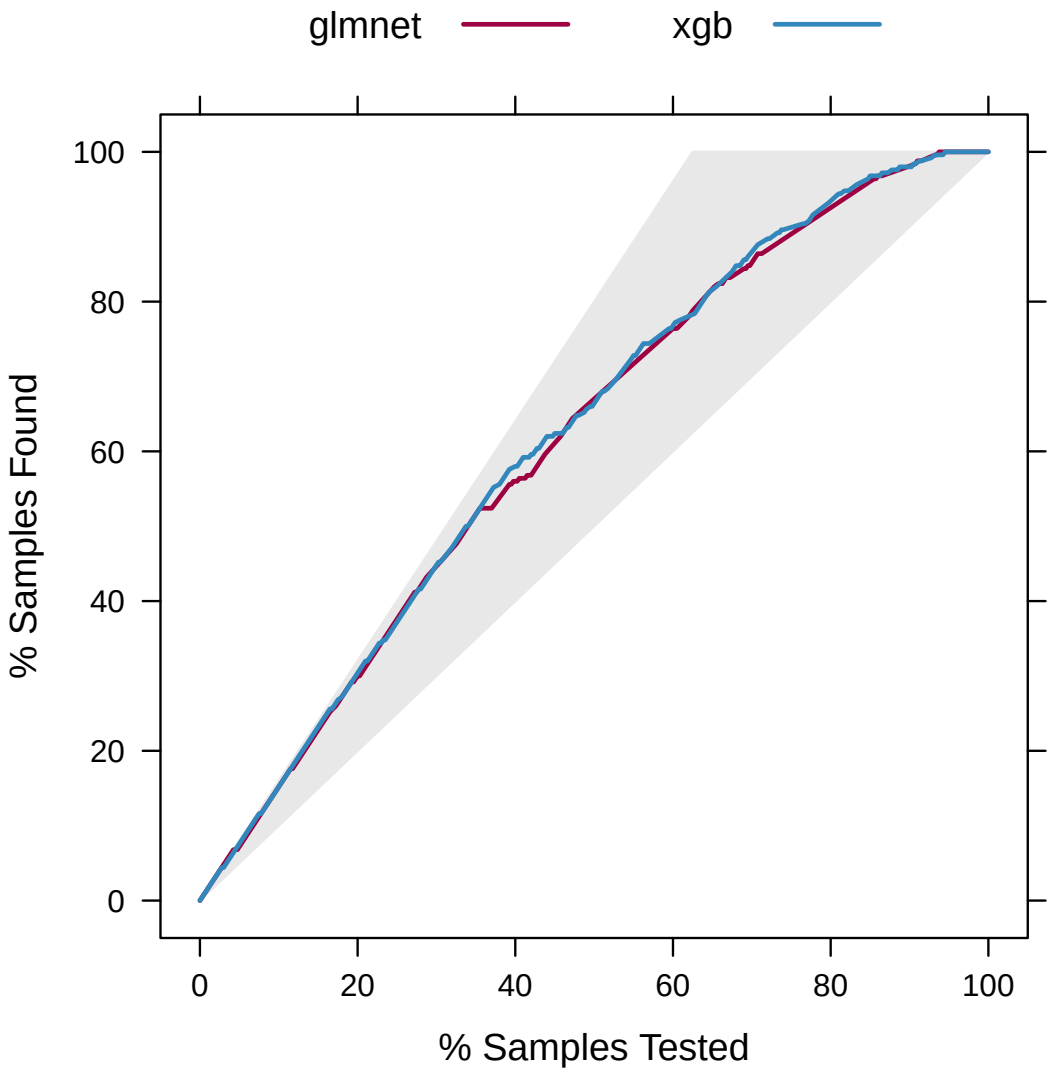
```
> test_probs <- data.frame(purch = test_data$purch,
+                          xgb = xgb_probs[, "yes"],
+                          glmnet = glmn_probs[, "yes"])
>
> head(test_probs)
```

```
  purch    xgb glmnet
1    no 0.0447  0.131
2    no 0.7592  0.759
3   yes 0.8151  0.585
4    no 0.3461  0.423
5   yes 0.8803  0.712
6    no 0.5490  0.584
```

```
> lift_obj <- caret::lift(purch ~ glmnet + xgb, data = test_probs)
> lift_obj


Call:
lift.formula(x = purch ~ glmnet + xgb, data = test_probs)

Models: glmnet, xgb
Event: yes (62.5%)
```

```
> ## plot(lift_obj, auto.key = list(lines = TRUE, points = FALSE))
```

Lift Charts



Session Info

- R version 3.3.3 (2017-03-06), `x86_64-apple-darwin13.4.0`
- Base packages: base, datasets, graphics, grDevices, grid, methods, parallel, stats, tcltk, utils
- Other packages: AppliedPredictiveModeling 1.1-6, BiocInstaller 1.24.0, C50 0.1.2, caret 6.0-79, class 7.3-14, ctv 0.8-4, doMC 1.3.4, foreach 1.4.4, Formula 1.2-2, ggplot2 2.2.1, Hmisc 4.1-1, iterators 1.0.9, kernlab 0.9-25, knitr 1.20, lattice 0.20-35, lubridate 1.7.4, mlbench 2.1-1, odfWeave 0.8.4, partykit 1.0-5, plyr 1.8.4, pROC 1.11.0, reshape2 1.4.3, rpart 4.1-13, survival 2.41-3, vcd 1.4-4, xgboost 0.6.4.1, XML 3.98-1.9
- Loaded via a namespace (and not attached): abind 1.4-5, acepack 1.4.1, assertthat 0.2.0, backports 1.1.2, base64enc 0.1-3, bindr 0.1.1, bindrcpp 0.2.2, broom 0.4.4, checkmate 1.8.5, cluster 2.0.6, codetools 0.2-15, colorspace 1.3-2, compiler 3.3.3, COREIearn 1.52.1, Cubist 0.2.2, CVST 0.2-1, data.table 1.10.4-3, ddalpha 1.3.2, DEoptimR 1.0-8, digest 0.6.15, dimRed 0.1.0.9001, dplyr 0.7.4, DRR 0.0.3, e1071 1.6-8, evaluate 0.10.1, foreign 0.8-69, geometry 0.3-6, glmnet 2.0-16, glue 1.2.0.9000, gower 0.1.2, gridExtra 2.3, gtable 0.2.0, highr 0.6, htmlTable 1.11.2, htmltools 0.3.6, htmlwidgets 1.0, ipred 0.9-6, labeling 0.3, latticeExtra 0.6-28, lava 1.6.1, lazyeval 0.2.1, lmtest 0.9-36, magic 1.5-8, magrittr 1.5, MASS 7.3-49, Matrix 1.2-12, mnormt 1.5-5, ModelMetrics 1.1.0, munsell 0.4.3, nlme 3.1-131, nnet 7.3-12, pillar 1.2.1, pkgconfig 2.0.1, prodlim 1.6.1, psych 1.8.3.3, purrr 0.2.4.9000, R6 2.2.2, RColorBrewer 1.1-2,