

# Applied Predictive Modeling

## Central Iowa R Users Group

Max Kuhn

Pfizer R&D

# “Predictive Modeling”

# Define That!

Rather than saying that method  $X$  is a predictive model, I would say:

## Predictive Modeling

is the process of creating a model whose *primary* goal is to achieve high levels of accuracy.

In other words, a situation where we are concerned with making the best possible prediction on an individual data instance.

(aka pattern recognition)(aka machine learning)

# Models

So, in theory, a linear or logistic regression model is a predictive model?

Yes.

As will be emphasized during this talk:

- the quality of the prediction is the focus
- model interpretability and inferential capacity are not as important

# Examples

- **spam detection**: we want the most accurate prediction that minimizes false positives and eliminates spam
- **plane delays, travel time**, etc.
- **customer volume**
- **sentiment analysis** of text
- **sale price of a property**

For example, does anyone care *why* an email or SMS is labeled as spam?

# Quantitative Structure Activity Relationships (QSAR)

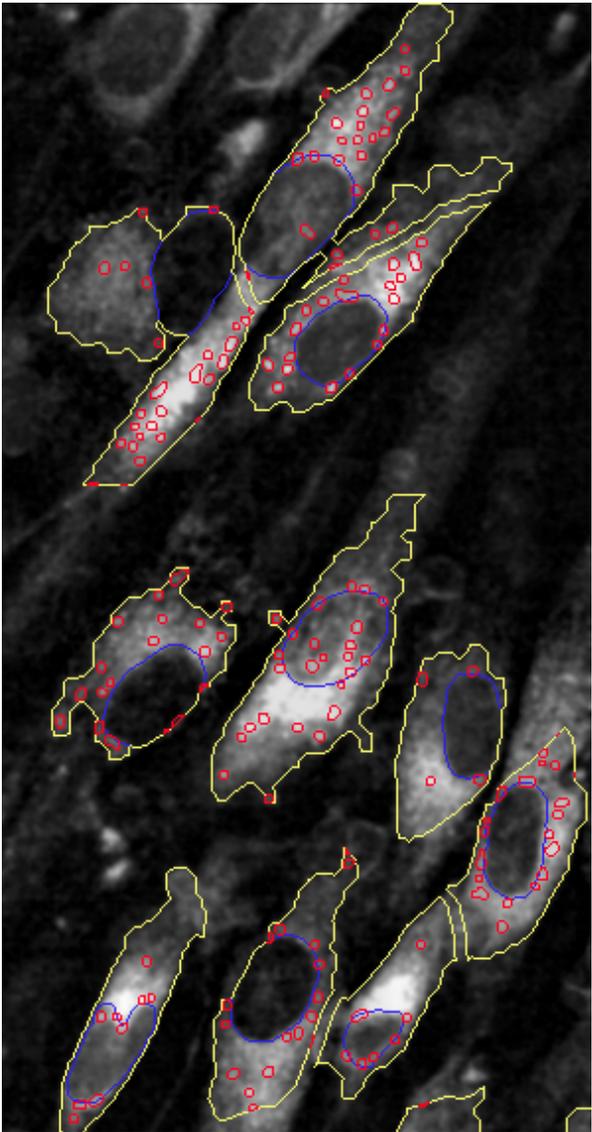
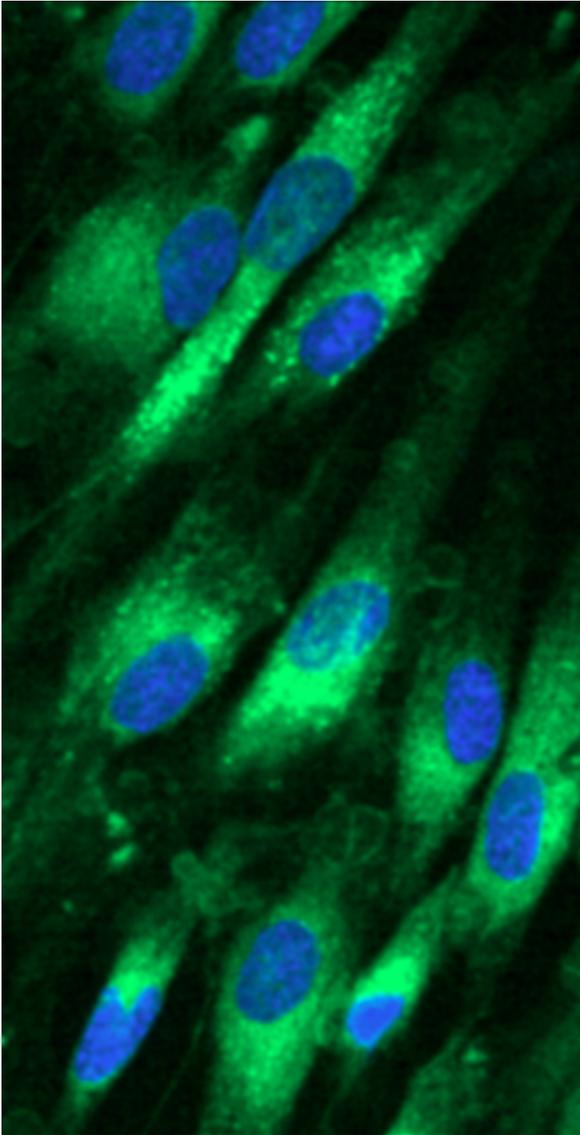
Pharmaceutical companies screen millions of molecules to see if they have good properties, such as:

- biologically potent
- safe
- soluble, permeable, drug-like, etc

We synthesize many molecules and run lab tests (“assays”) to estimate the characteristics listed above.

When a medicinal chemist designs a new molecule, he/she would like a prediction to help assess whether we should synthesized it.

# Cell Segmentation



# Cell Segmentation

Individual cell results are aggregated so that decisions can be made about specific compounds

Improperly segmented objects might compromise the quality of the data so an algorithmic filter is needed.

In this application, we have measurements on the size, intensity, shape or several parts of the cell (e.g. nucleus, cell body, cytoskeleton).

Can these measurements be used to predict poorly segmentation using a set of manually labeled cells?

# The Data

Hill *et al* (2007) scored 2019 cells into these two bins: well-segmented (WS) or poorly-segmented (PS). The data are in the `caret` package.

There are 58 measurements in each cell that can be used as predictors.

```
> library(caret)
> data(segmentationData)
> dim(segmentationData)
```

```
[1] 2019  61
```

```
> str(segmentationData[, 1:9])
```

```
'data.frame': 2019 obs. of  9 variables:
 $ Cell      : int  207827637 207932307 207932463 207932470 207932455 207827656 20
 $ Case      : Factor w/ 2 levels "Test","Train": 1 2 2 2 1 1 1 1 1 1 ...
 $ Class     : Factor w/ 2 levels "PS","WS": 1 1 2 1 1 2 2 1 2 2 ...
 $ AngleCh1  : num  143.25 133.75 106.65 69.15 2.89 ...
 $ AreaCh1   : int  185 819 431 298 285 172 177 251 495 384 ...
 $ AvgIntenCh1: num  15.7 31.9 28 19.5 24.3 ...
 $ AvgIntenCh2: num  4.95 206.88 116.32 102.29 112.42 ...
 $ AvgIntenCh3: num  9.55 69.92 63.94 28.22 20.47 ...
 $ AvgIntenCh4: num  2.21 164.15 106.7 31.03 40.58 ...
```

# 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

# Model Building Steps

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

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

# Spending Our Data

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

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

# Spending Our Data

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 these models emphasize the need for an untouched set of samples to evaluate performance.

The authors designated a training set ( $n = 1009$ ) and a test set ( $n = 1010$ ). We'll use these:

```
> ## remove the cell identifier
> segmentationData$Cell <- NULL
> seg_train <- subset(segmentationData, Case == "Train")
> seg_test  <- subset(segmentationData, Case == "Test")
>
> seg_train$Case <- NULL
> seg_test$Case <- NULL
```

# Spending Our Data

*If* you wanted to do a random 50/50 split of the data, there is a function in `caret` that can be used:

```
> ## NOT executed...
> ## make a balanced random split
> in_train <- createDataPartition(segmentationData$Class, p = 0.5, list = FALSE)
>
> ## `in_train` is set of row indices that are selected to go
> ## into the training set
> train_data <- segmentationData[ in_train,]
> test_data  <- segmentationData[-in_train,]
```

# 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 of splits in a tree model

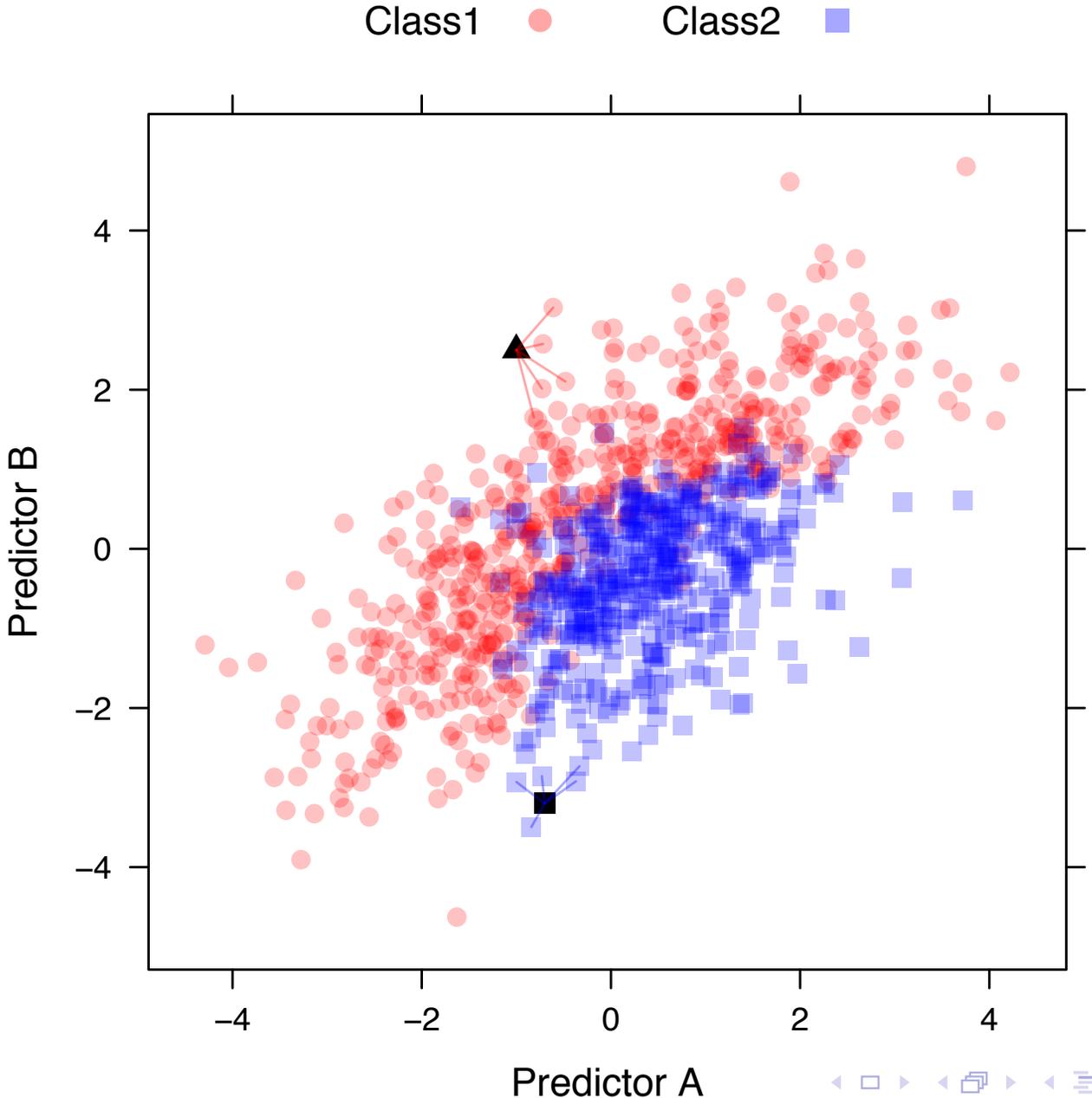
# Over-Fitting

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.

# $K$ -Nearest Neighbors Classification



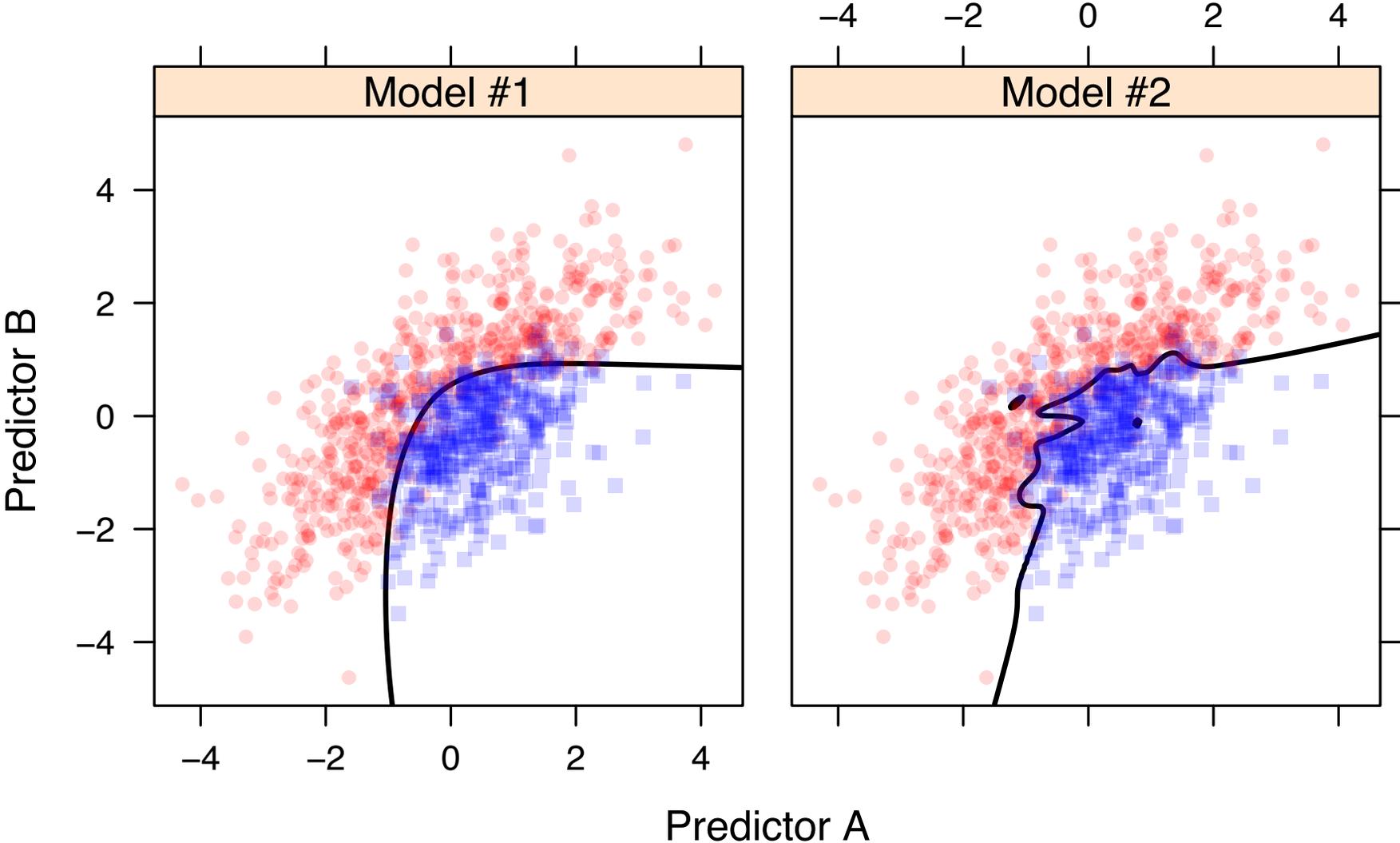
# Over-Fitting

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



# 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).

Examples are cross–validation (in many varieties) and the bootstrap.

These procedures repeated split the *training data* into subsets used for modeling and performance evaluation.

# *K*-Fold Cross-Validation

Original Data



*Build Model With*

CV Group #1



CV Group #2



CV Group #3



*Predict On*



# The Big Picture

We think that resampling will give us honest estimates of future performance, but there is still the issue of which sub-model to select (e.g. 5 or 10 NN).

One algorithm to select sub-models:

Define sets of model parameter values to evaluate;

**for** *each parameter set* **do**

**for** *each resampling iteration* **do**

        Hold-out specific samples ;

        [optional] pre-process the predictor data;

        Fit the model on the remainder;

        Predict the hold-out samples;

**end**

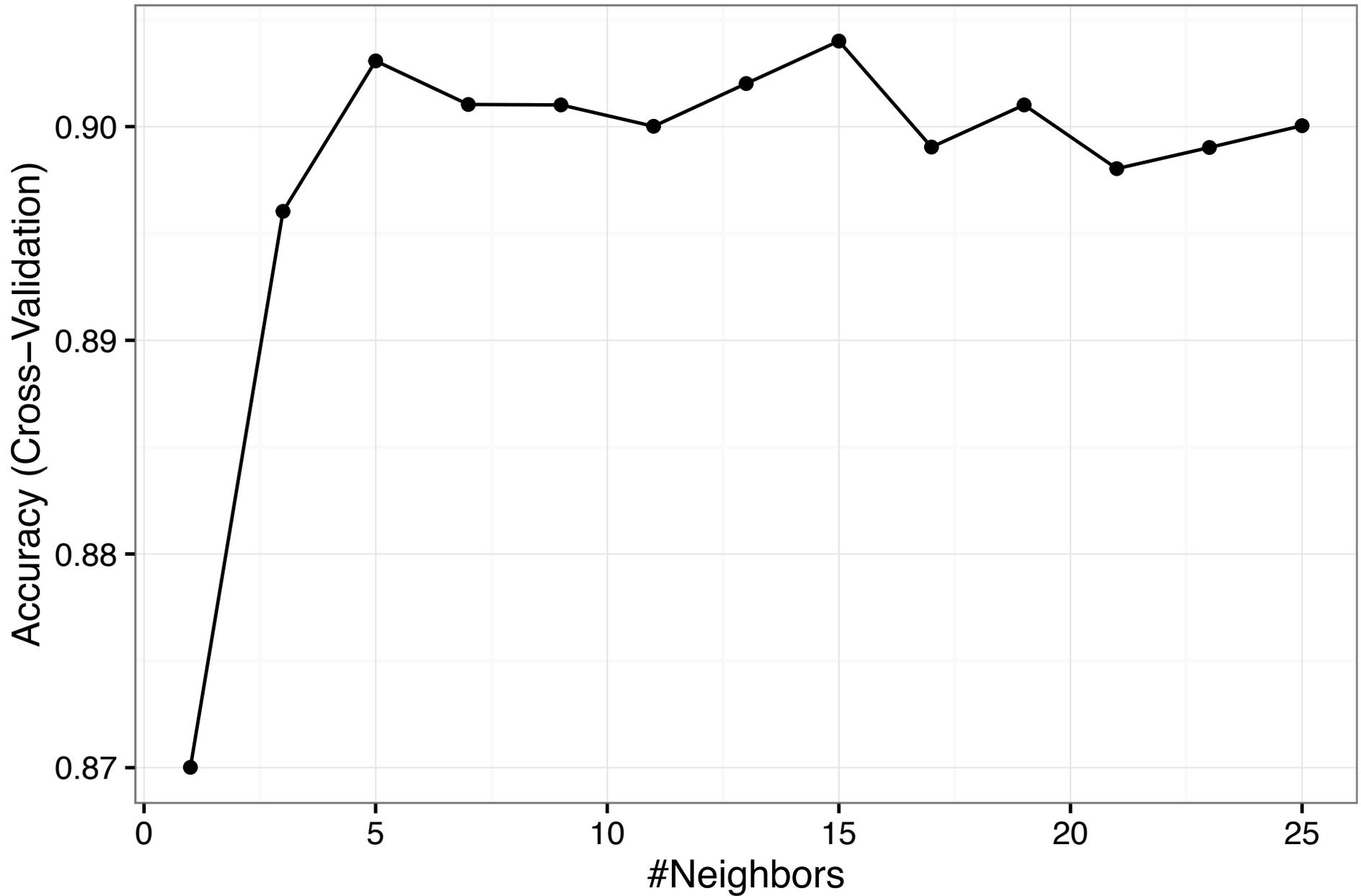
    Calculate the average performance across hold-out predictions

**end**

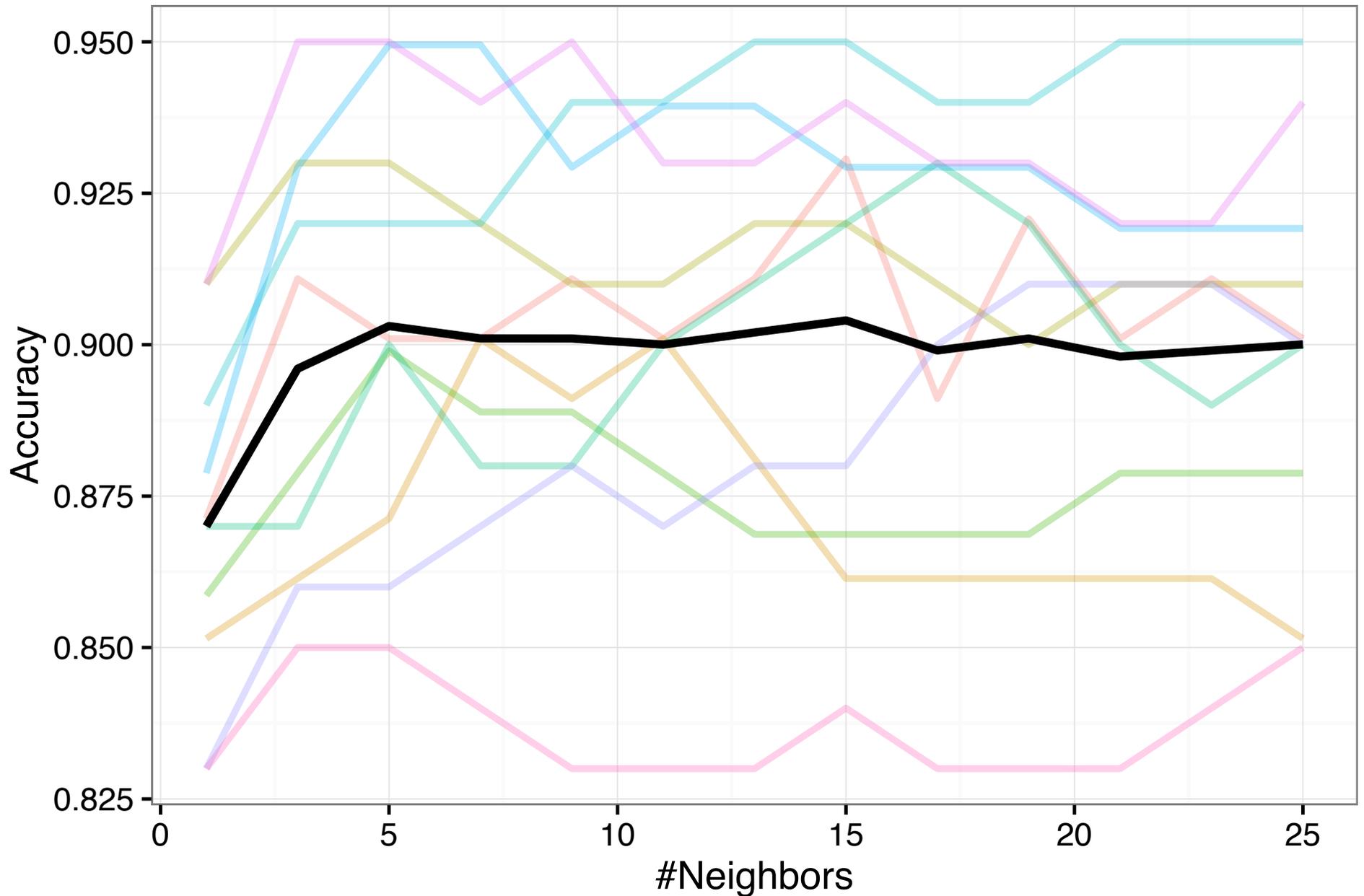
Determine the optimal parameter value;

Create final model with entire training set and optimal parameter value;

# $K$ -Nearest Neighbors Tuning



# $K$ -Nearest Neighbors Tuning – Individual Resamples



# Typical Process for Model Building

Now that we know how to evaluate models on the training set, we can try different techniques (including pre-processing) and try to optimize model performance.

Performance might not be the only consideration. Others might include:

- simplicity of prediction
- reducing the number of predictors (aka features) in the model to reduce cost or complexity
- smoothness of the prediction equation
- robustness of the solution

Once we have 1-2 candidate models, we can evaluate the results on the test set.

# Linear Discriminant Analysis

A simple model for fitting linear class boundaries to these data is linear discriminant analysis (LDA).

The model computes the mean vector for the data within each class and a common covariance matrix across the entire training set then uses the differences (discriminant functions):

$$D(\mathbf{u}) = \mathbf{u}'S^{-1}(\bar{\mathbf{x}}_P - \bar{\mathbf{x}}_W)$$

A few packages have this model, but we'll use the `lda` function in the **MASS** package:

```
> library(MASS)
> lda_fit <- lda(Class ~ ., data = seg_train, tol = 1.0e-15)
```

# Linear Discriminant Analysis Predictions

This object doesn't provide any resampled estimate of performance but we can get predictions:

```
> predict(lda_fit, newdata = seg_train[1:3,])
```

```
$class
```

```
[1] WS WS PS
```

```
Levels: PS WS
```

```
$posterior
```

	PS	WS
2	0.26358158	0.73641842
3	0.05269107	0.94730893
4	0.95044135	0.04955865

```
$x
```

	LD1	LD2
2	1.055130	-1.50625016
3	2.015937	-0.26134111
4	-1.013353	-0.07737231

# 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.

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

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

# 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 **poorly segmented cell**:

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

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

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 that a cell is poorly segmented is  $\geq 50\%$ , they would be labelled as a poorly segmented.

What happens when you change the cutoff?

Increasing it makes it harder to be called PS  $\rightarrow$  fewer predicted events, sensitivity  $\uparrow$ , specificity  $\downarrow$

Decreasing the cutoff makes it easier to be called PS  $\rightarrow$  more predicted events, sensitivity  $\downarrow$ , specificity  $\uparrow$

# ROC Curve

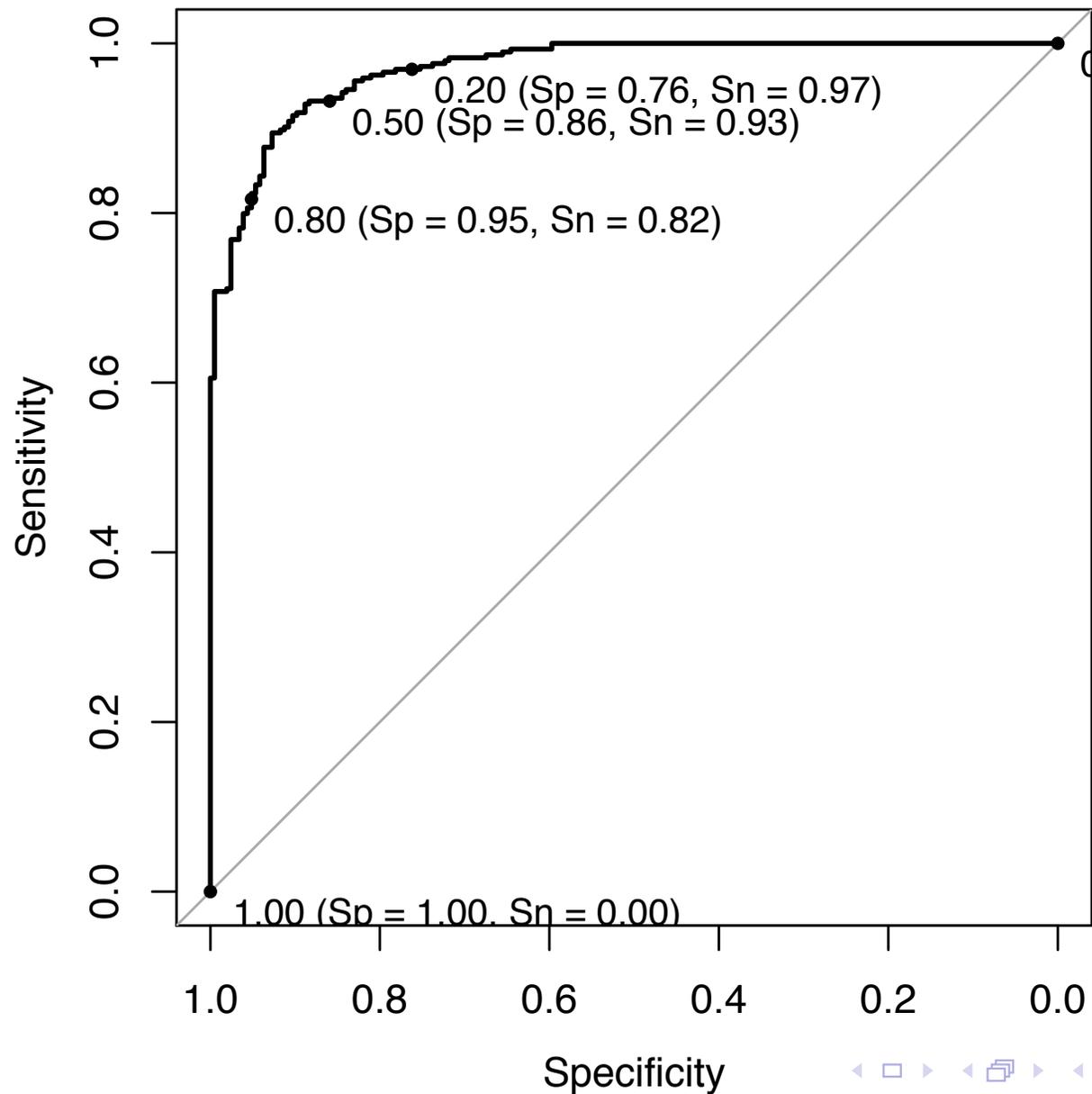
With two classes the Receiver Operating Characteristic (ROC) curve can be used to estimate performance using a combination of sensitivity and specificity.

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.

# Example ROC Curve



# Creating the ROC Curve

We'll use the `pROC` function to compute and plot the ROC curve.

First, we need a set of predicted class probabilities and then we use the `roc` function

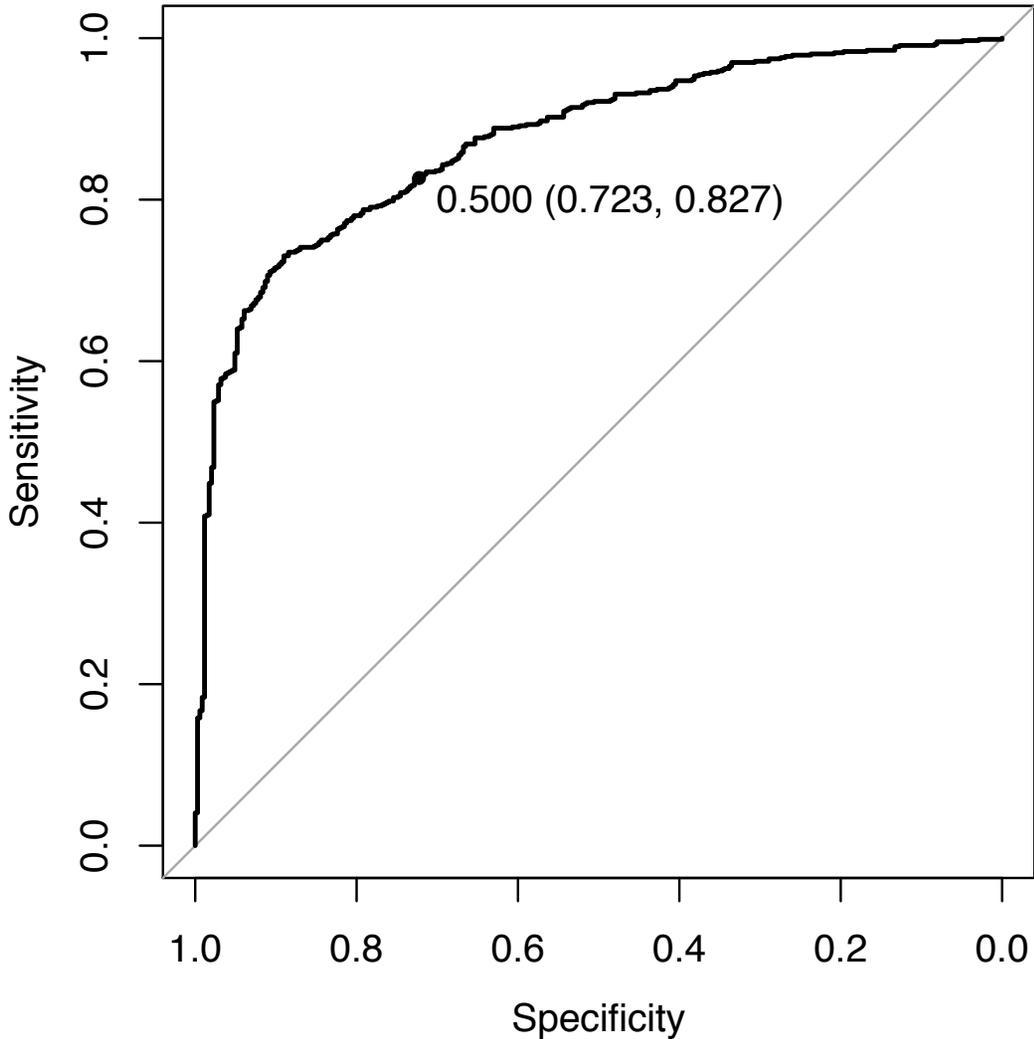
```
> lda_test_pred <- predict(lda_fit, newdata = seg_test)
> library(pROC)
>
> lda_roc <- roc(response = seg_test$Class,
+               predictor = lda_test_pred$posterior[, "PS"],
+               ## we need to tell the function that the _first_ level
+               ## is our event of interest
+               levels = rev(levels(seg_test$Class)))
> lda_roc
```

Call:  
roc.default(response = seg\_test\$Class, predictor = lda\_test\_pred\$posterior[, "P

Data: lda\_test\_pred\$posterior[, "PS"] in 346 controls (seg\_test\$Class WS) < 664 cas  
Area under the curve: 0.874

```
> # plot(exRoc print.thres = .5)
```

# LDA ROC Curve



# LDA Confusion Matrix

```
> confusionMatrix(data = lda_test_pred$class, reference = seg_test$Class)
```

Confusion Matrix and Statistics

```
      Reference
Prediction PS  WS
PS      549  96
WS      115 250
```

Accuracy : 0.7911

95% CI : (0.7647, 0.8158)

No Information Rate : 0.6574

P-Value [Acc > NIR] : <2e-16

Kappa : 0.5422

McNemar's Test P-Value : 0.2153

Sensitivity : 0.8268

Specificity : 0.7225

Pos Pred Value : 0.8512

Neg Pred Value : 0.6849

Prevalence : 0.6574

Detection Rate : 0.5436

Detection Prevalence : 0.6386

Balanced Accuracy : 0.7747

'Positive' Class : PS

# 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	<code>predict(obj)</code> (no options needed)
glm	stats	<code>predict(obj, type = "response")</code>
gbm	gbm	<code>predict(obj, type = "response", n.trees)</code>
mda	mda	<code>predict(obj, type = "posterior")</code>
rpart	rpart	<code>predict(obj, type = "prob")</code>
Weka	RWeka	<code>predict(obj, type = "probability")</code>
LogitBoost	caTools	<code>predict(obj, type = "raw", nIter)</code>

# The caret Package

The `caret` package was developed to:

- create a unified interface for modeling and prediction (interfaces to 216 models – up from 112 a year ago)
- 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

## Next Steps

How can we get resampled estimates of the area under the ROC curve for the LDA model (without going to the test set)?

Let's use five repeats of 10-fold cross-validation to assess the area under the ROC curve with the LDA model.

First, we need to specify the model terms and what type of technique that we are using:

```
> ## setting the seed before calling `train` controls the resamples
> set.seed(20792)
> lda_mod <- train(Class ~ ., data = seg_train, method = "lda")
```

`train` can use the formula and the non-formula method. The two interfaces may lead to different results for *some* models that do not need dummy variable conversions of factors.

# The `train` Function

The default resampling scheme is the bootstrap. Let's use five repeats of 10-fold cross-validation instead.

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

To use five repeats of 10-fold cross-validation, we would use

```
> ctrl <- trainControl(method = "repeatedcv", repeats = 5)
>
> set.seed(20792)
> lda_mod <- train(Class ~ ., data = seg_train, method = "lda",
+                 trControl = ctrl)
```

# The `train` Function

By classification, the default performance metrics that are computed are accuracy and the kappa statistic. For regression, they are RMSE and  $R^2$ .

Instead, let's measure the area under the ROC curve, sensitivity, and specificity.

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

```
> twoClassSummary(fakeData)
```

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

# The `train` Function

We can pass the `twoClassSummary` function in through `trainControl`.

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

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

```
> ctrl <- trainControl(method = "repeatedcv", repeats = 5,  
+                       classProbs = TRUE,  
+                       summaryFunction = twoClassSummary)  
>  
> set.seed(20792)  
> lda_mod <- train(Class ~ ., data = seg_train,  
+                 method = "lda",  
+                 ## Add the metric argument  
+                 trControl = ctrl, metric = "ROC",  
+                 ## Also pass in options to `lda` using `...`  
+                 tol = 1.0e-15)
```

# 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

## 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)
```

# LDA Resampling Results

```
> lda_mod
```

```
Linear Discriminant Analysis
```

```
1009 samples
```

```
58 predictor
```

```
2 classes: 'PS', 'WS'
```

```
No pre-processing
```

```
Resampling: Cross-Validated (10 fold, repeated 5 times)
```

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

```
Resampling results
```

ROC	Sens	Spec	ROC SD	Sens SD	Spec SD
0.8735494	0.8576141	0.7124609	0.04039168	0.05127234	0.08085324

The value 0.8735 is the average of the 50 resamples. The test set estimate was 0.874.

# Other Models

`train` provides wrappers to a lot of different models/packages. A list can be found on the project homepage and by checking `?models`.

Many of these models have tuning parameters. There are two ways to specify the tuning parameters:

- `tuneLength`: `train` has methods for determining grids of the parameters. Using this argument let's the user prescribe how many candidate values to evaluate.
- `tuneGrid`: you can also create a data frame where the columns are the tuning parameters and the rows are the combinations that should be evaluated.

# Tuning the Number of Neighbors

Another new argument that we can pass to `train` is `preProc`. This applies different types of pre-processing to the predictors and is done within resamples. It is also automatically applied when predicting too.

We will center and scale the predictors so that the distance metric isn't biased by scale.

```
> ## The same resamples are used
> set.seed(20792)
> knn_mod <- train(Class ~ ., data = seg_train,
+                 method = "knn",
+                 trControl = ctrl,
+                 ## tuning parameter values to evaluate
+                 tuneGrid = data.frame(k = seq(1, 25, by = 2)),
+                 preProc = c("center", "scale"),
+                 metric = "ROC")
```

# Tuning the Number of Neighbors

```
> knn_mod
```

```
k-Nearest Neighbors
```

```
1009 samples
```

```
 58 predictor
```

```
 2 classes: 'PS', 'WS'
```

```
Pre-processing: centered (58), scaled (58)
```

```
Resampling: Cross-Validated (10 fold, repeated 5 times)
```

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

```
Resampling results across tuning parameters:
```

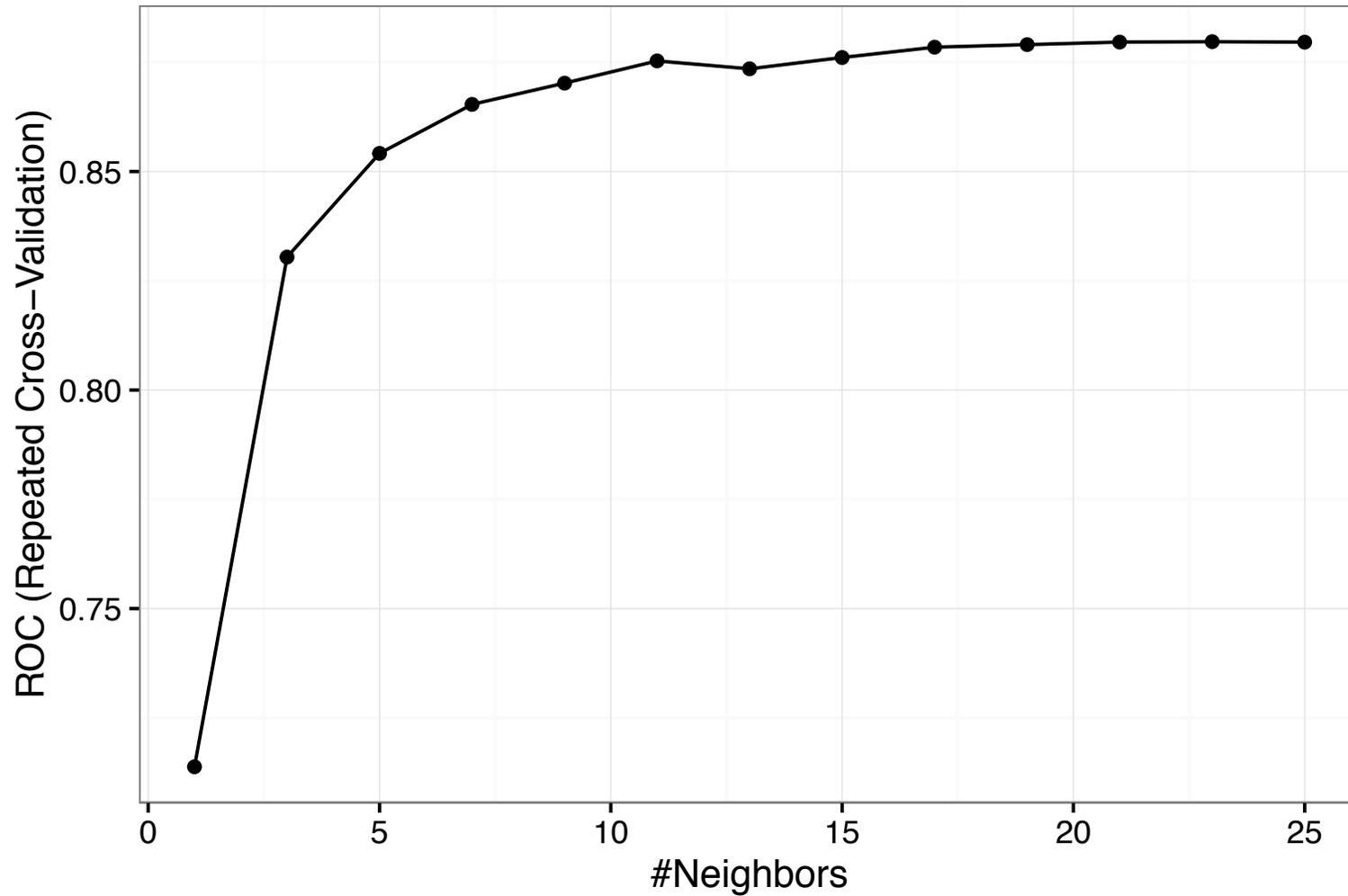
k	ROC	Sens	Spec	ROC SD	Sens SD	Spec SD
1	0.7137900	0.7783730	0.6600853	0.06125686	0.05111556	0.07329513
3	0.8304147	0.8236855	0.7151778	0.03680898	0.05055039	0.07470409
5	0.8541445	0.8218056	0.7355903	0.03665873	0.04970615	0.07092892
7	0.8653454	0.8299851	0.7398720	0.03600259	0.05286398	0.07515553
9	0.8702121	0.8318552	0.7452774	0.03582426	0.04954339	0.07204114
11	0.8752995	0.8372024	0.7505405	0.03614511	0.04751353	0.06814953
13	0.8734974	0.8441022	0.7408108	0.03792648	0.04780407	0.07110998
15	0.8760790	0.8412698	0.7402845	0.03739929	0.05258230	0.07126982
17	0.8784485	0.8453472	0.7494310	0.03790634	0.05157438	0.07089234
19	0.8790282	0.8459871	0.7494737	0.03722205	0.05210655	0.06577087
21	0.8796156	0.8469196	0.7558464	0.03824543	0.05187434	0.06394297
23	0.8797082	0.8494345	0.7521195	0.03825331	0.04987768	0.07036613
25	0.8796015	0.8465923	0.7462589	0.03845505	0.05034432	0.07717937

```
ROC was used to select the optimal model using the largest value.
```

```
The final value used for the model was k = 23.
```

# $K$ -Nearest Neighbors Tuning

```
> ggplot(knn_mod)
```



# Predicting New Samples

```
> ## to get the classes:  
> predict(knn_mod, newdata = head(seg_test))
```

```
[1] PS PS WS WS PS WS  
Levels: PS WS
```

```
> ## We choose `prob` to get class probabilities:  
> predict(knn_mod, newdata = head(seg_test), type = "prob")
```

	PS	WS
1	0.9565217	0.04347826
2	0.8695652	0.13043478
3	0.2173913	0.78260870
4	0.3043478	0.69565217
5	1.0000000	0.00000000
6	0.4782609	0.52173913

# Comparing Models

Many of the predictors are skewed. Would transforming them via the Yeo–Johnson transformation help?

```
> ## The same resamples are used
> set.seed(20792)
> knn_yj_mod <- train(Class ~ ., data = seg_train,
+                     method = "knn",
+                     trControl = ctrl,
+                     tuneGrid = data.frame(k = seq(1, 25, by = 2)),
+                     preProc = c("center", "scale", "YeoJohnson"),
+                     metric = "ROC")
>
> ## What was the best area under the ROC curve?
> getTrainPerf(knn_yj_mod)
```

```
      TrainROC TrainSens TrainSpec method
1 0.8888078 0.8522768 0.7586344      knn
```

# Comparing Models

```
> ## Conduct o a paired t-test on the resampled AUC values to control for  
> ## resample-to-resample variability:  
> compare_models(knn_yj_mod, knn_mod, metric = "ROC")
```

One Sample t-test

```
data: x  
t = 4.4595, df = 49, p-value = 4.796e-05  
alternative hypothesis: true mean is not equal to 0  
95 percent confidence interval:  
 0.004999086 0.013200217  
sample estimates:  
 mean of x  
0.009099651
```

```
> ## Yes, but not by much
```

# Some Notes on Tuning

- `train` attempts to fit as few models as possible. In quite a few places, we use a “sub-model trick” to get predictions for some sub-models without refitting
- *random search*, where the tuning parameters are randomly selected, can be used when you want to try a larger range of parameters
- *adaptive resampling* (aka racing) can be used to reduce the time to tuning the models based on interim analyses to discard candidate sub-models
- the `resamples` class can be used to visualize and compare models on a larger scale than `compare_models` function
- `train` allows for much more flexibility to customize the tuning process

# Final Notes

There's a lot more to tell about predictive modeling in R and what **caret** can do.

There are many functions for feature selection in the package. The website has more information on this and other aspects.

Thanks for listening!