Never Tell Me the Odds! Machine Learning with Class Imbalances

Max Kuhn

Pfizer R&D

June 27, 2016

Outline

- Description of the problem with class imbalances (with illustrative data)
- A short refresher on predictive models, parameter tuning and resampling
- Measuring performance with imbalances
- An even shorter description of tree-based classification models (single models and ensembles)
- Sampling methods for combating class imbalances
- Cost-sensitive learning methods (trees, SVM)

Outline

The slides and code for this presentation are in the github repository https://github.com/topepo/useR2016.

The following R packages are used (and their dependencies): AppliedPredictiveModeling, caret, C50, DMwR, ROSE, kernlab, pROC, randomForest, and rpart.

Some section headers reference specific chapters in *Applied Predictive Modeling*.

Statistical Issues with Class Imbalances

First and foremost, the objective function used by many models to estimate parameters is to minimize error rates (or similar functions).

Severe class imbalances negatively affect this. For example, for a data set with a 5% event rate, a non-informative model would easily achieve 95% accuracy.

Since the *no information rate* is not near $1/C$, where C is the number of classes, assessing model performance can be tricky.

Also, for some models, a large class imbalance can acutely skew the distribution of the class probabilities. As a consequence, they are poorly calibrated and the default rules for determining an event (e.g. 50% cutoff) often produce artificially pessimistic performance values.

Here, we demonstrate some methods to produce better models fits as well as better metrics for evaluating models.

A short refresher on predictive models, parameter tuning and resampling

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)

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

(APM Chapter 4)

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

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

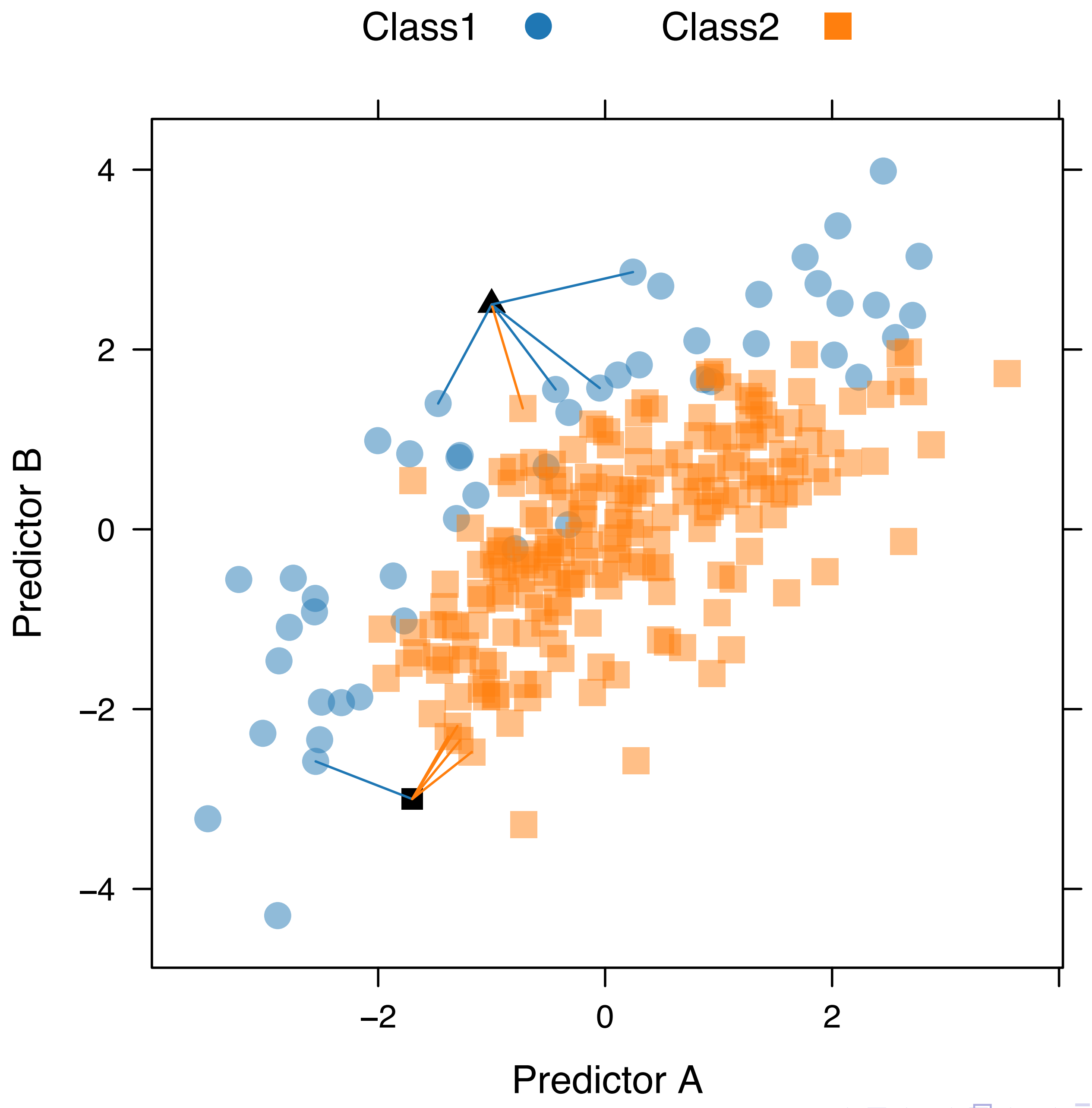
Over–Fitting

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



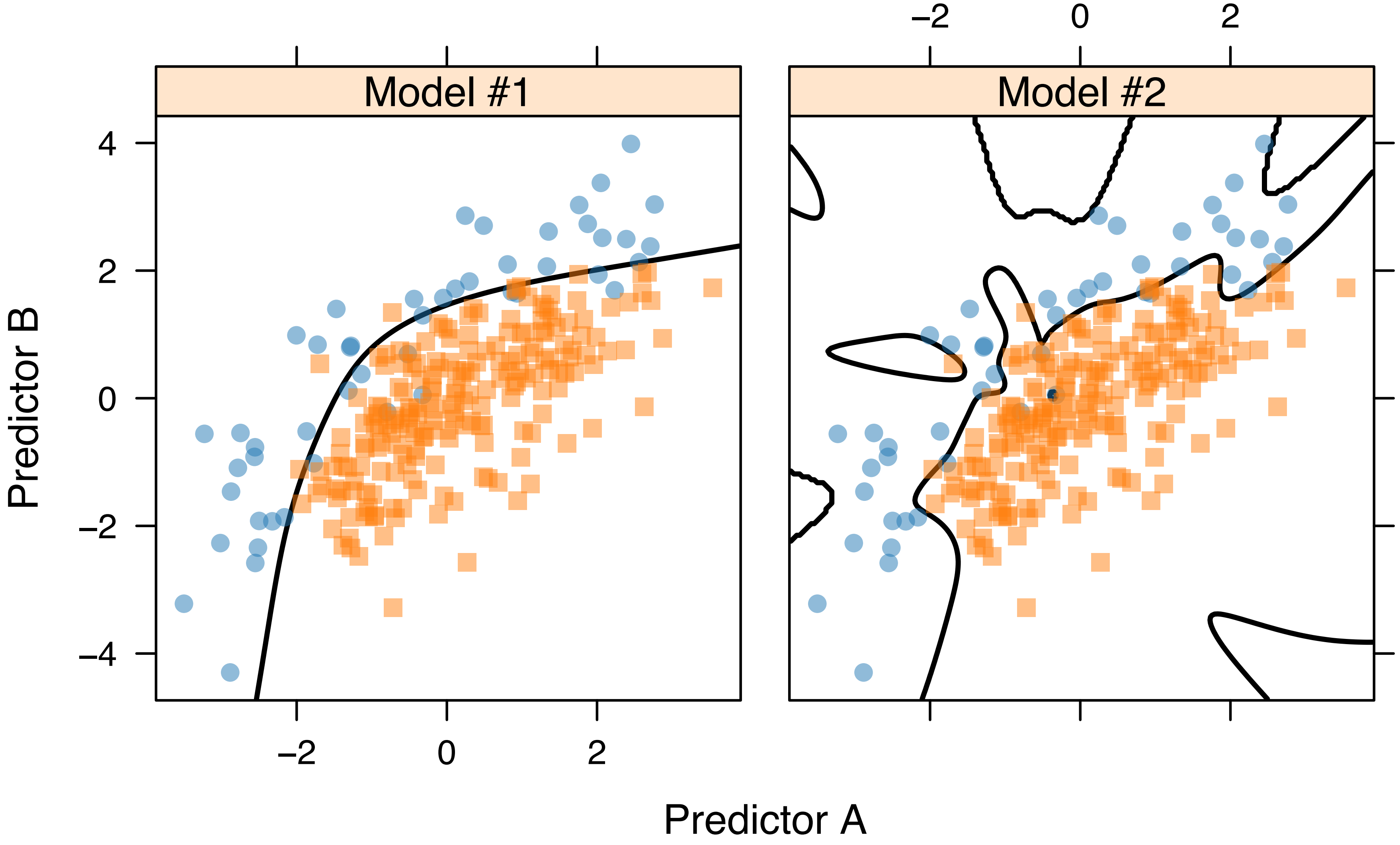
Over–Fitting

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.

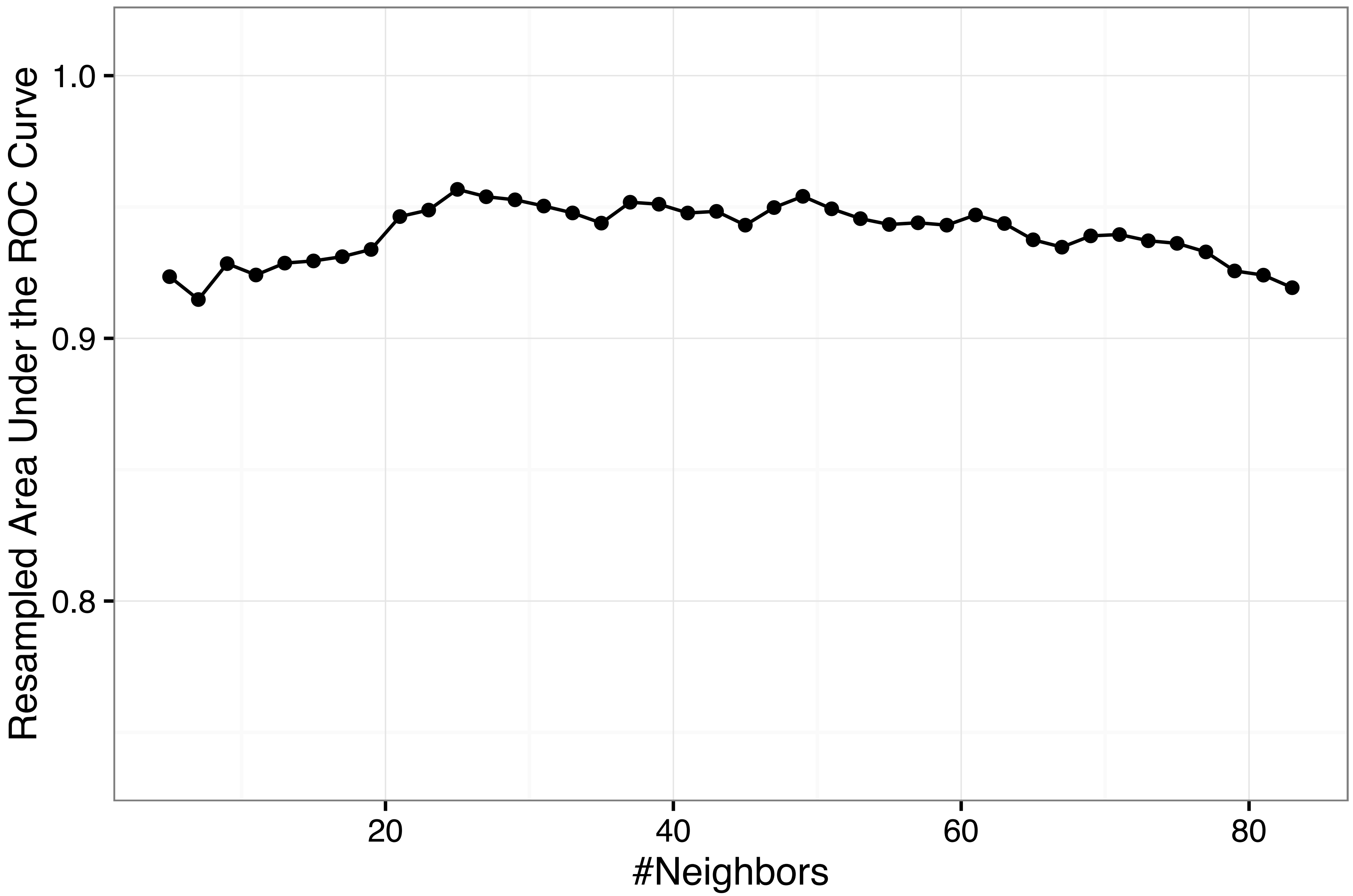
The Big Picture

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).

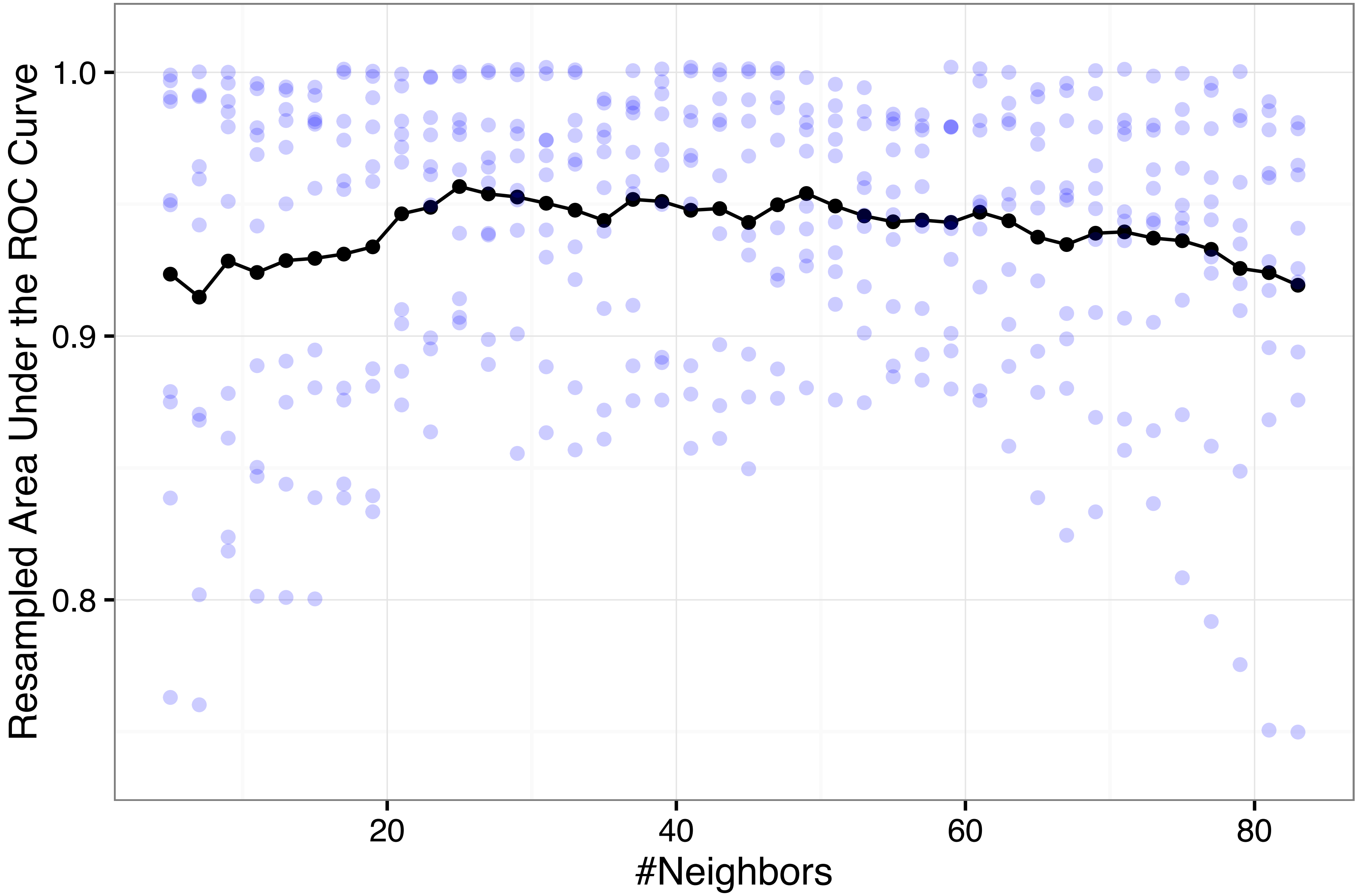
One algorithm to select sub–models:

```
Define sets of model parameter values to evaluate;
for each parameter set do
    for each resampling iteration do
        Hold–out specific samples ;
        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



Example Data Sets

Example Data – Electronic Medical Records

We have a blinded set of 9518 subjects from electronic medical records. Each is classified as either an event ($n = 1518$) or a non–event ($n = 8000$) yielding a prevalence of 15.95$\%$.

For each record, 38 predictors were collected. Of these, 32 are integers that indicate the incidence or severity of a number of `Conditions`. 6 others reflect `Demographics`.

We will use a stratified random split to put 2/3 of these data into the training set.

Data are originally on this github repository.

Example Data – Electronic Medical Records

```
> str(emr, list.len = 20)
```

```
'data.frame': 9518 obs. of  39 variables:
 $ Class        : Factor w/ 2 levels "event","noevent": 2 2 2 2 2 2 2 2 2 2 ...
 $ Condition1   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition2   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition3   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition4   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition5   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition6   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition7   : int  0 0 1 0 0 0 0 0 0 0 ...
 $ Condition8   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition9   : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition10 : int  1 0 0 0 0 0 0 0 0 0 ...
 $ Condition11 : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition12 : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition13 : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition14 : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition15 : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition16 : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition17 : int  0 1 1 0 0 0 0 0 1 0 ...
 $ Condition18 : int  0 0 0 0 0 0 0 0 0 0 ...
 $ Condition19 : int  0 0 0 0 0 1 0 0 0 0 ...
  [list output truncated]
```

Example Data – Electronic Medical Records

```
> library(caret)
>
> set.seed(1732)
> emr_ind <- createDataPartition(emr$Class, p = 2/3, list = FALSE)
> emr_train <- emr[ emr_ind,]
> emr_test  <- emr[-emr_ind,]
>
> mean(emr_train$Class == "event")

[1] 0.1594705

> mean(emr_test$Class == "event")

[1] 0.1595208

> table(emr_train$Class)


  event noevent
   1012    5334

> table(emr_test$Class)


  event noevent
    506    2666
```

Example Data – OKCupid

Kim and Escobedo–Land (2015) describe an anonymized data set from the OKCupid dating website of 51747 profiles from the San Francisco area between 2011 and 2012. The data are available on their github repo.

The data has fields for user's characteristics, including gender, age, ethnicity, spoken languages, etc. There are also free-form essays which are not used in this analysis.

For our purposes, we will try to predict if the user's occupation is in the STEM field (science, technology, engineering, and mathematics). These users account for 18.5% of the data.

Code to assemble the data are in the github repository for this workshop. The final data are in an object called `okc`.

Example Data – OKCupid

```
> str(okc, list.len = 20, vec.len = 2)
```

```
'data.frame': 51747 obs. of  332 variables:
 $ age                   : int  22 35 23 29 29 ...
 $ body_type             : Factor w/ 13 levels "missing","a_little_extra",..: 2 4 1
 $ diet                  : Factor w/ 19 levels "missing","anything",..: 12 8 19 1 5
 $ drinks                : Factor w/ 7 levels "missing","desperately",..: 6 4 6 6 6
 $ drugs                 : Factor w/ 4 levels "missing","never",..: 2 4 1 2 1 ...
 $ education             : Factor w/ 33 levels "missing","college_university",..: 2
 $ height                : int  75 70 71 66 67 ...
 $ income                : Factor w/ 13 levels "missing","inc20000",..: 1 8 2 1 1 .
 $ last_online           : num  3 2 3 4 2 ...
 $ offspring             : Factor w/ 16 levels "missing","doesnt_have_kids",..: 4 4
 $ orientation           : Factor w/ 4 levels "missing","bisexual",..: 4 4 4 4 4 ..
 $ pets                  : Factor w/ 16 levels "missing","dislikes_cats",..: 16 16
 $ religion              : Factor w/ 10 levels "missing","agnosticism",..: 2 2 1 1
 $ sex                   : Factor w/ 3 levels "missing","f",..: 3 3 3 3 3 ...
 $ sign                  : Factor w/ 13 levels "missing","aquarius",..: 6 4 9 2 12
 $ smokes                : Factor w/ 6 levels "missing","no",..: 3 2 2 2 2 ...
 $ status                : Factor w/ 6 levels "missing","available",..: 5 5 5 5 5 .
 $ where_state           : Factor w/ 41 levels "arizona","california",..: 2 2 2 2 2
 $ where_town            : Factor w/ 51 levels "alameda","albany",..: 48 29 6 41 41
 $ religion_modifer      : Factor w/ 5 levels "missing","and_laughing_about_it",..:
  [list output truncated]
```

Example Data – OKCupid

```
> set.seed(1732)
> okc_ind <- createDataPartition(okc$Class, p = 2/3, list = FALSE)
> okc_train <- okc[ okc_ind,]
> okc_test  <- okc[-okc_ind,]
>
> mean(okc_train$Class == "stem")

[1] 0.184701

> mean(okc_test$Class == "stem")

[1] 0.1846591
```

Measuring Performance with Imbalances

(APM Section 14.1)

Using Predicted Classes

The first aspect of statistical analysis of imbalanced data that should be reassessed is the method for characterizing *model performance*

As previously said, overall accuracy is problematic for at least two reasons.

First, the baseline event rate is not taken into account. One method for doing this when *class predictions* are made is to use the Kappa statistic:

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

where O is the observed accuracy, E is the expected accuracy under chance agreement, and $\kappa \in [-1, 1]$

For example, if $O = 0.96$ and $E = 0.95$, $\kappa = 0.20$

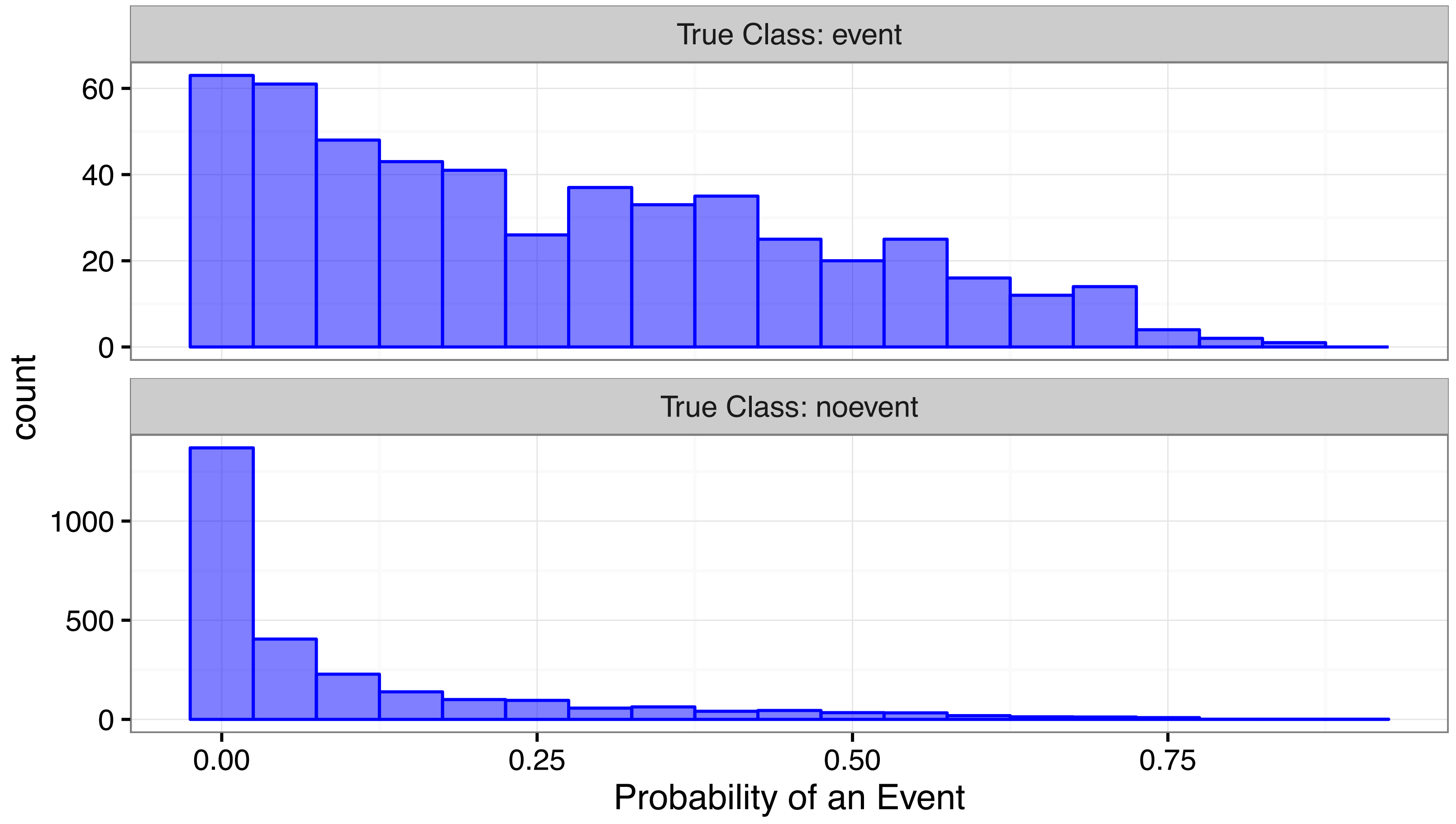
Using Predicted Classes

The second issue with accuracy, or any metric based on class predictions, is that the default $1/C$ cutoff for class probabilities (C is the number of classes) is often a poor choice.

Many predictive models will overfit to the majority class and, as such, their class probability distributions are highly skewed.

For example, here are some test set predictions for the EMR data conditioned on the true results

Class Probabilities



Estimating Performance For Classification

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

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

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

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

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

Estimating Performance For Classification

For our example, let's choose the event to be the **a patient having an event**:

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

$$\text{Specificity} = \frac{\#\text{ truly not events predicted to be non– events}}{\#\text{ true non–events}}$$

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.

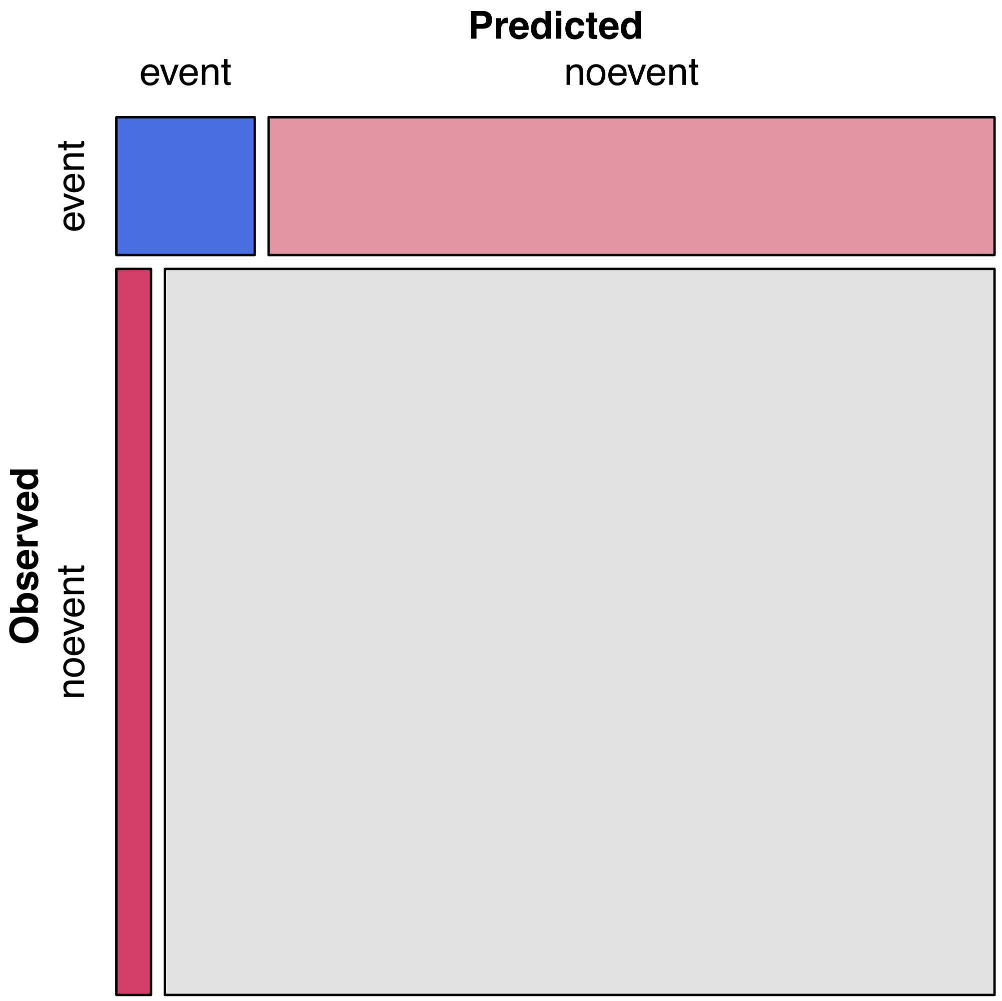
For two classes, the 50% cutoff is customary; if the probability that a patient has an event is $\geq 50\%$, they would be labelled as an event.

What happens when you change the cutoff?

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

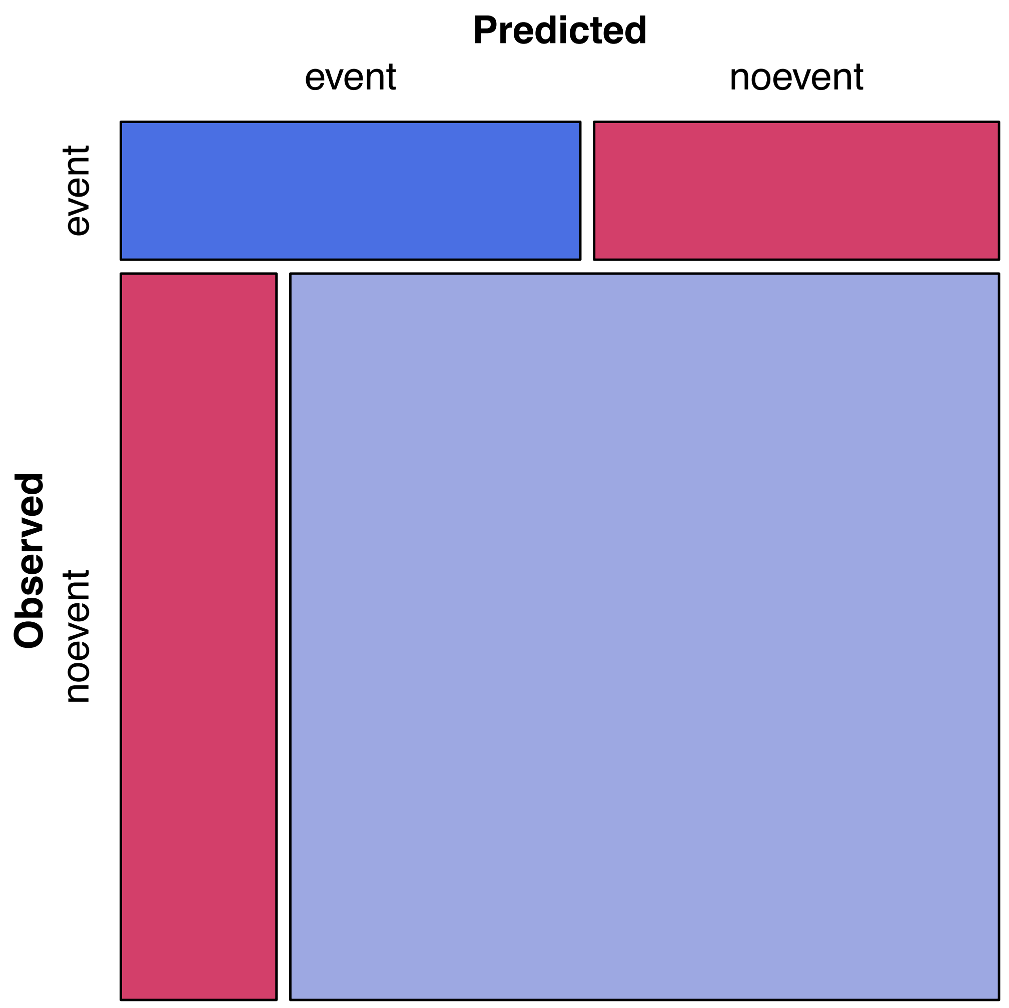
Decreasing the cutoff makes it easier to be called an event $\rightarrow$ more predicted events, sensitivity $\downarrow$, specificity $\uparrow$

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



Sensitivity $= 16\%$, Specificity $= 96\%$, $\kappa = 0.161\%$

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



Sensitivity $= 53.2\%$, Specificity $= 82\%$, $\kappa = 0.294\%$,

ROC Curve

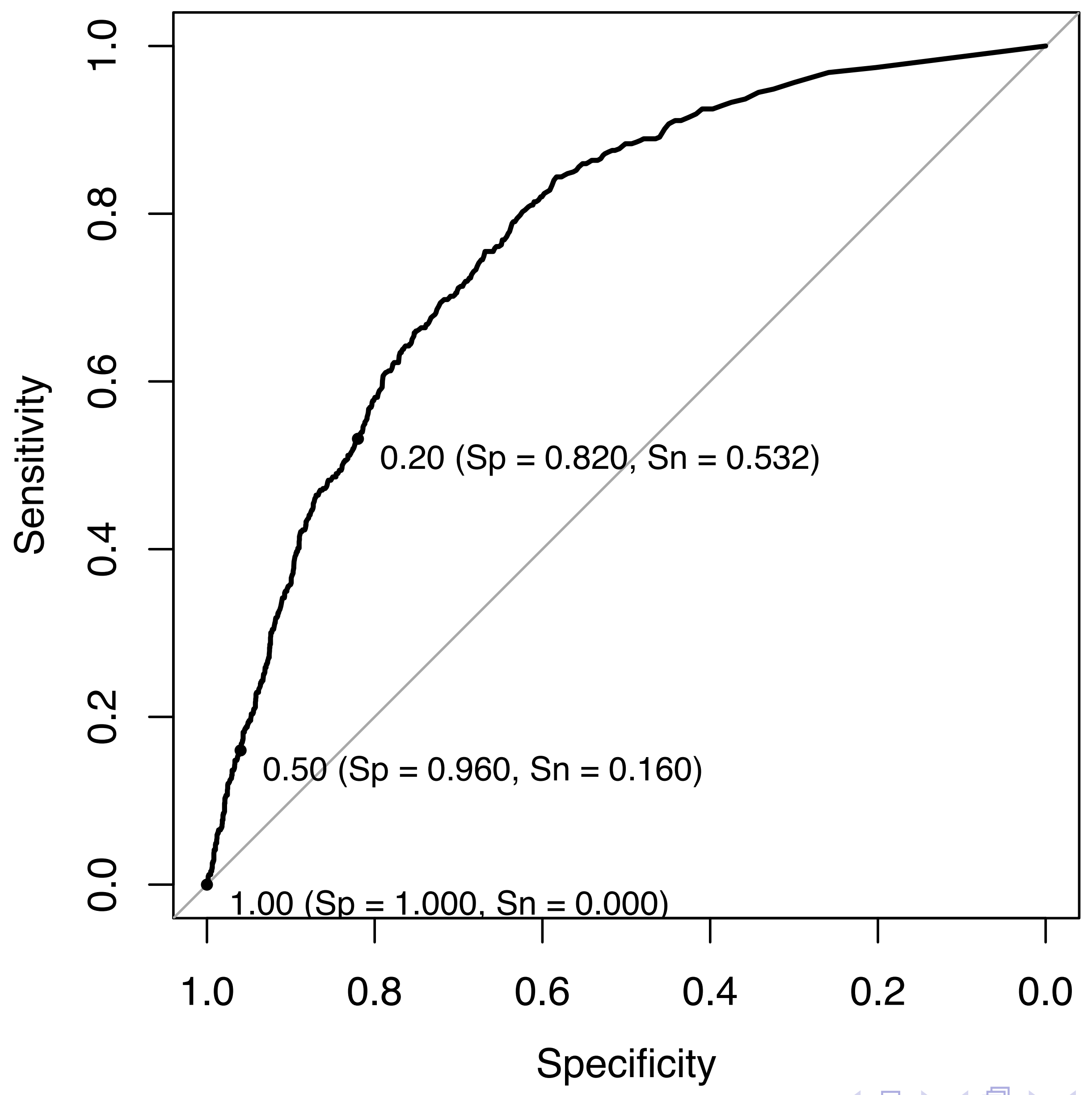
With two classes the Receiver Operating Characteristic (ROC) curve can be used to estimate performance using a combination of sensitivity and specificity.

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



0.20 (Sp = 0.820, Sn = 0.532)
0.50 (Sp = 0.960, Sn = 0.160)
1.00 (Sp = 1.000, Sn = 0.000)
Sensitivity
0.0
0.2
0.4
0.6
0.8
1.0
Specificity
1.0
0.8
0.6
0.4
0.2
0.0

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.

Random Forests

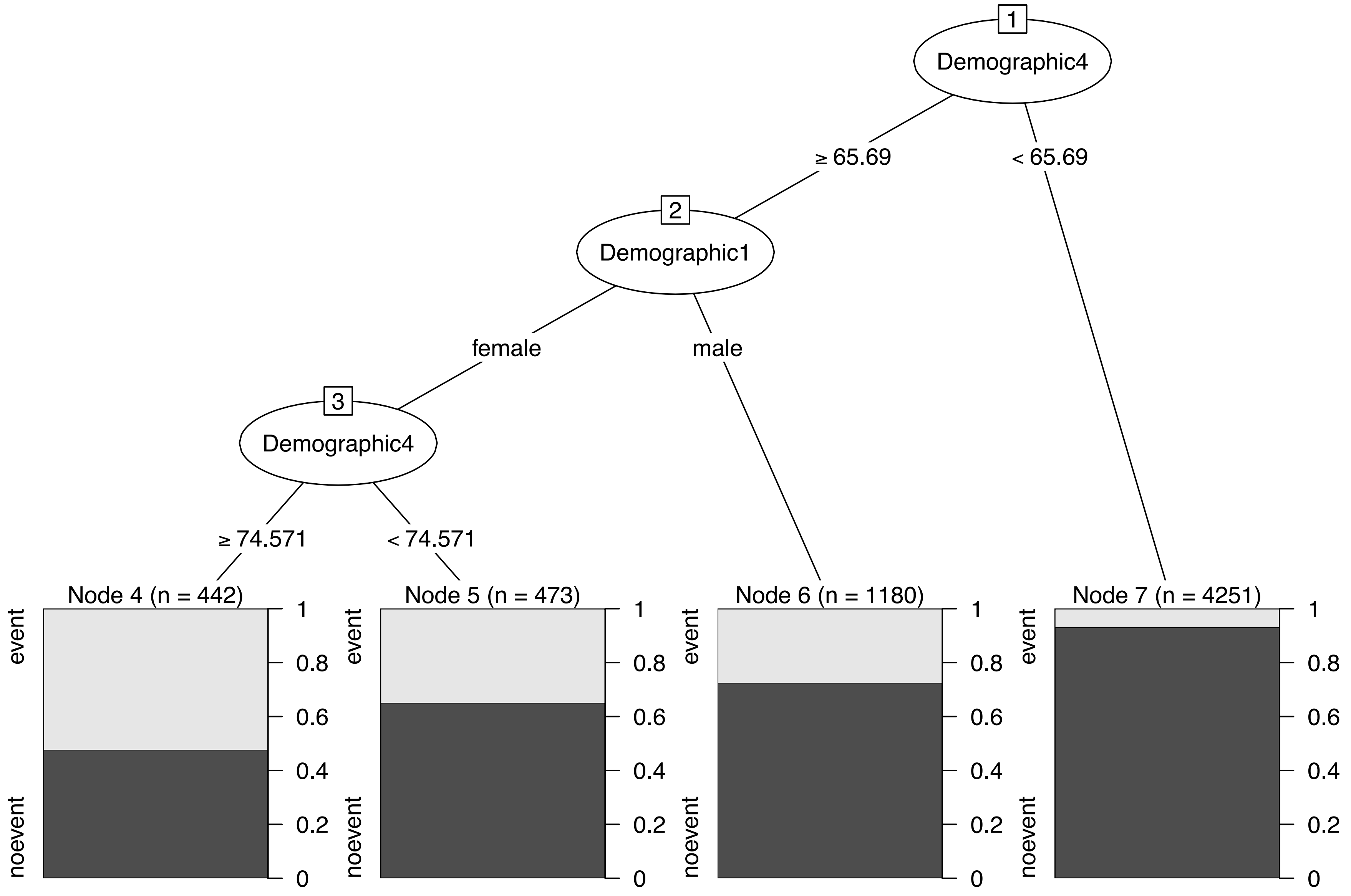
Random forest models are a collection (aka *ensemble*) of tree-based models created from variations of the training set.

Single trees are a set of `if`/`then` statements derived from the data. At the end of the conditional statements are *terminal nodes* with the training set samples that satisfy those conditions.

The tree structure is derived to produce terminal nodes that have the *purest* blend of class frequencies for those data.

Here is an example of a tree for the EMR data coerced to be fairly shallow.

A Single Shallow Tree



Pros and Cons of Single Trees

Trees can be computed very quickly and have simple interpretations.

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

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

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

This last point has been exploited to improve the performance of trees via ensemble methods where many trees are fit and predictions are aggregated across the trees.

Random Forests

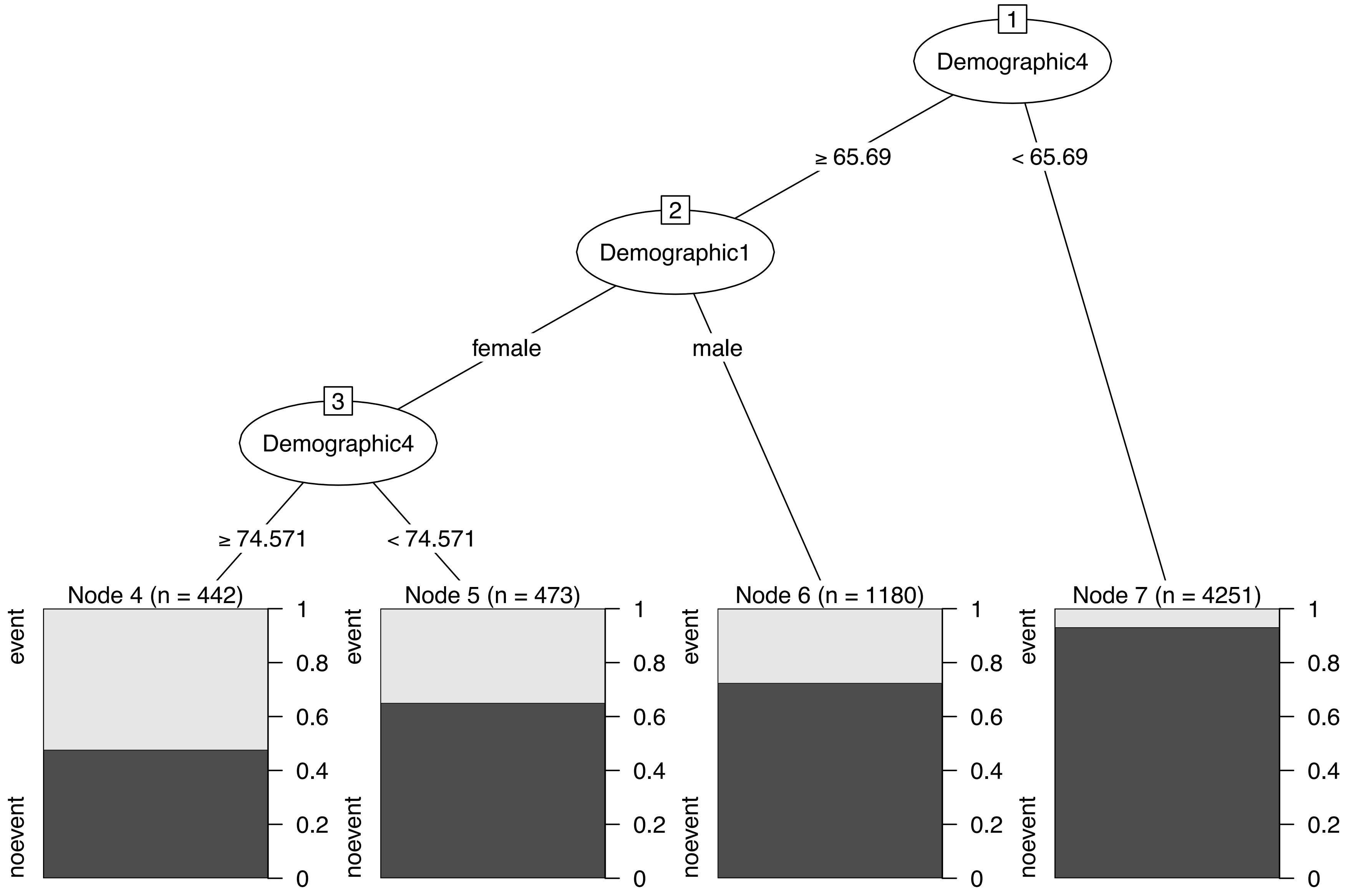
These models are created by producing a large number of trees on bootstrapped versions of the training set. The same tree–building process is used except that

- no pruning is used
- at *each split* that is made, only a random subset of predictors of size m_{try} are evaluated

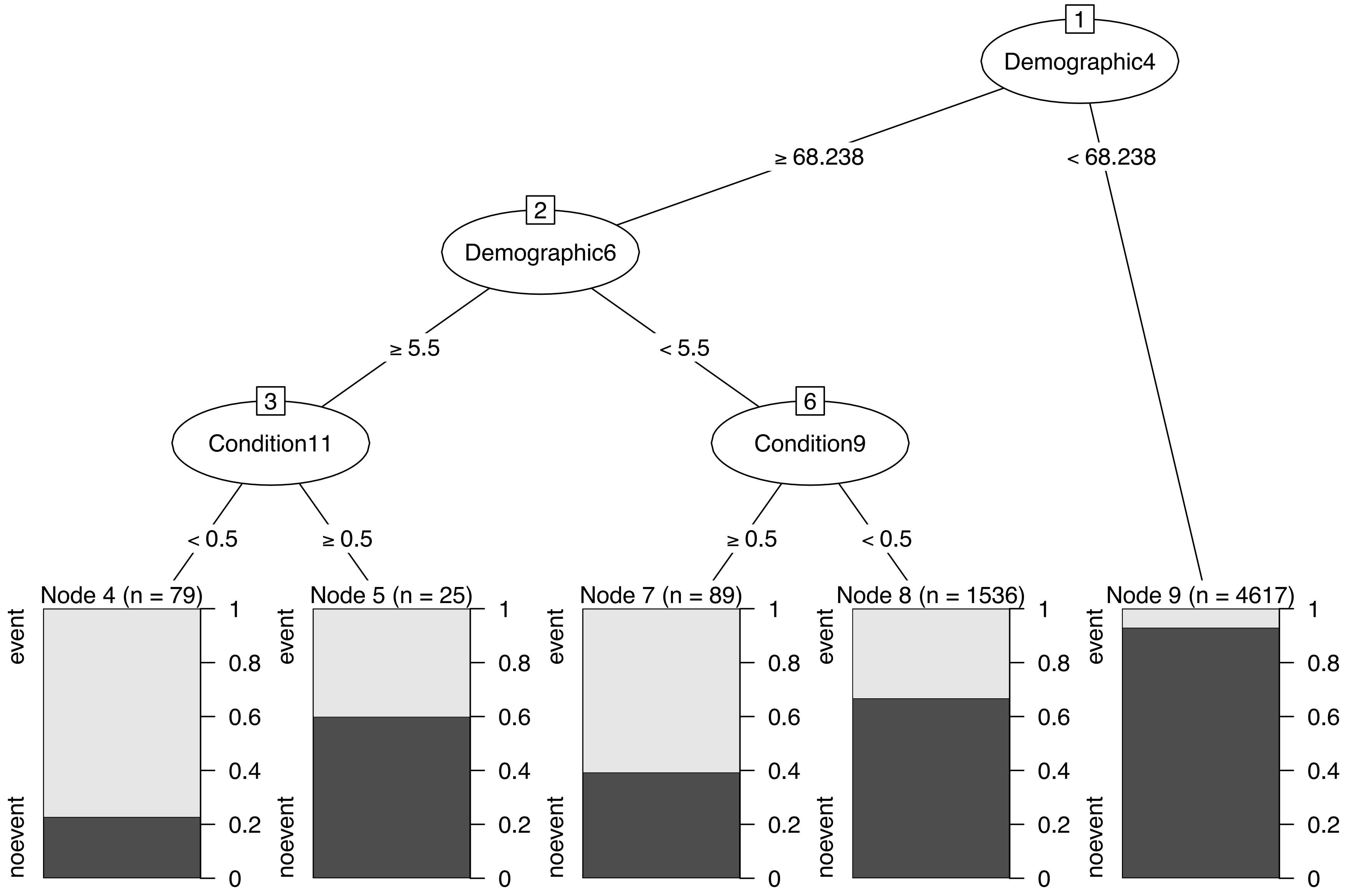
For a new sample, each tree in the forests votes each class using the prediction from that tree.

The sampling procedure is designed to decorrelate the trees so that more diverse votes are generated.

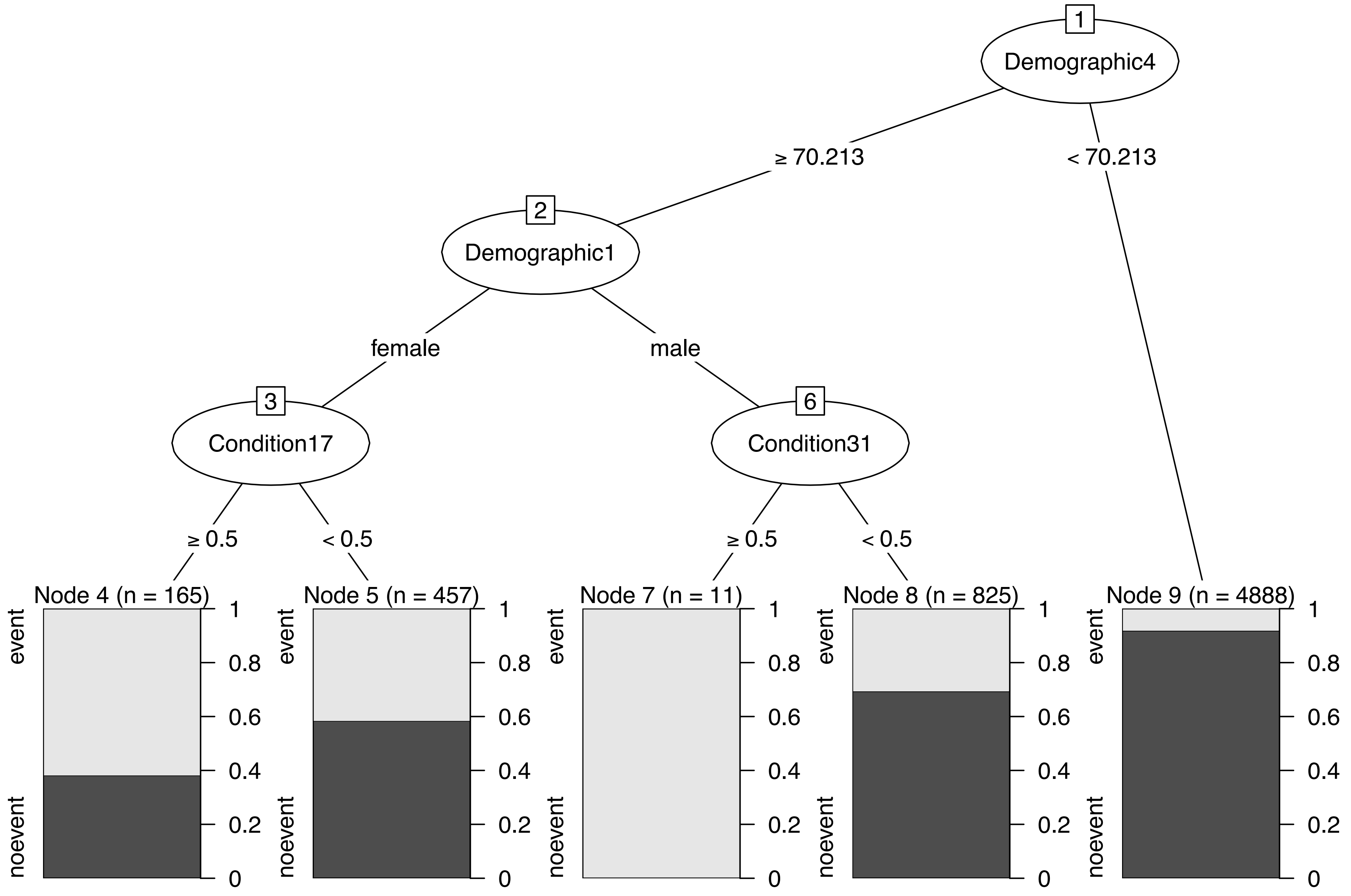
A Single Shallow Tree (Original)



A Single Shallow Tree (Bootstrapped)



A Single Shallow Tree (Bootstrapped)



Random Forests with the EMR Data

This model was fit using all the predictors and 1,000 constituent trees.

To optimize the value of m_{try}:

- the area under the ROC curve was optimized
- performance was evaluated using five repeats of 10–fold cross validation
- values of m_{try} between one and the total number of predictors (45) were evaluated.

Although not necessary, predictors that were categorical were converted to dummy variables prior to the random forest model fit to decrease training time.

Random Forests with the EMR Data

```
> ctrl <- trainControl(method = "repeatedcv",
+                      repeats = 5,
+                      classProbs = TRUE,
+                      savePredictions = TRUE,
+                      summaryFunction = twoClassSummary)
> emr_grid <- data.frame(mtry = c(1:15, (4:9)*5))
>
> set.seed(1537)
> rf_emr_mod <- train(Class ~ .,
+                     data = emr_train,
+                     method = "rf",
+                     metric = "ROC",
+                     tuneGrid = emr_grid,
+                     ntree = 1000,
+                     trControl = ctrl)
```

Two Quick Notes

- Using the formula method with `train` will feed the underlying model function dummy variables (if any factors are predictors). You can also use the non-formula method to preserve the grouped factors.
- The EMR data have no factor predictors.
- Setting the seed before calling `train` will ensure that the cross–validation folds are used across models.

Random Forests — EMR Example

The tuning results are shown on the next slide. The optimal value of m_{try} was estimated to be 8 predictors. This results in a model with 133,524 terminal nodes across the 1000 trees in the forest.

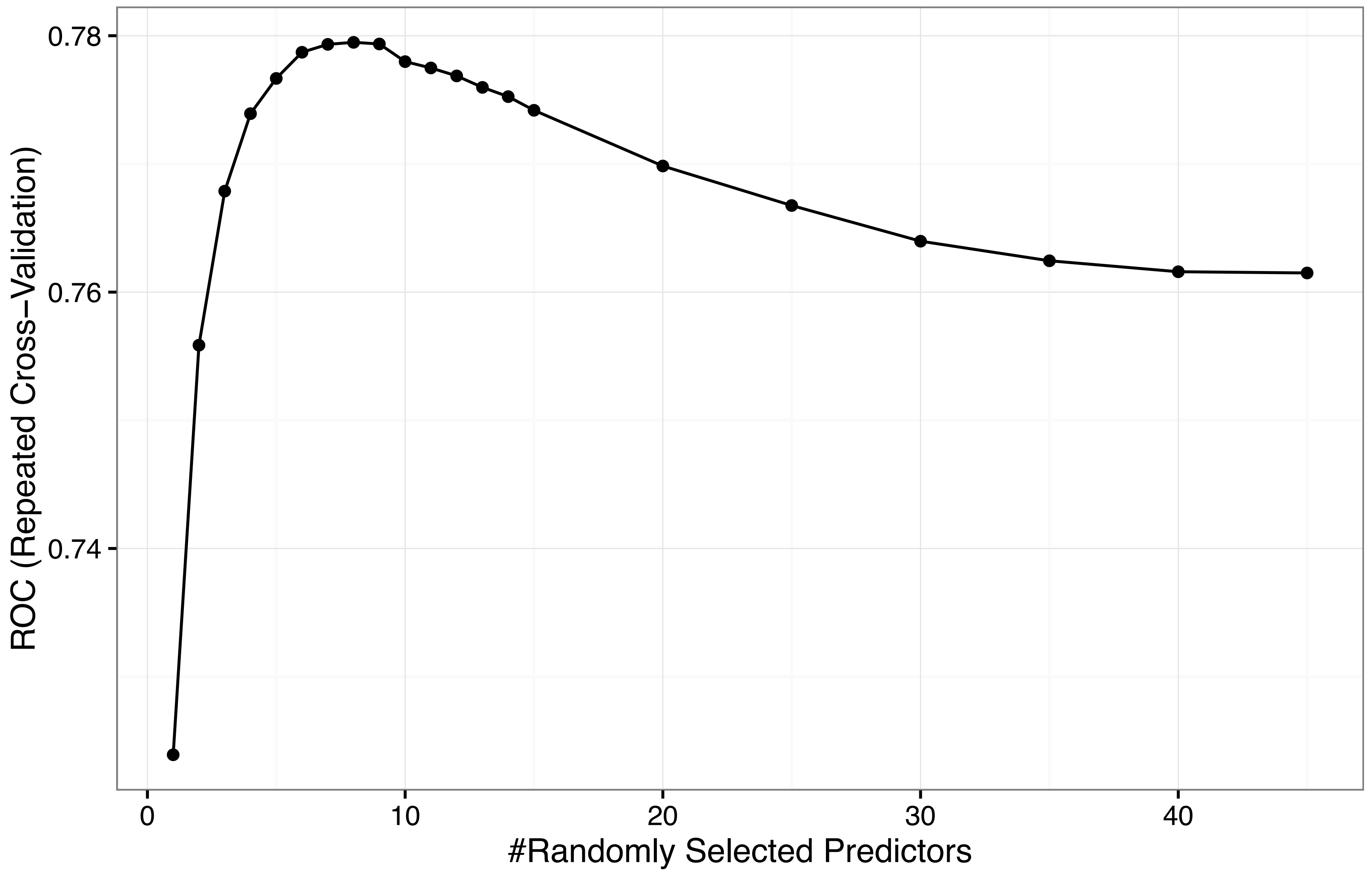
The associated area under the ROC curve was estimated to be 0.779.

Note that, with the default 50% class probability threshold, some metrics are lopsided

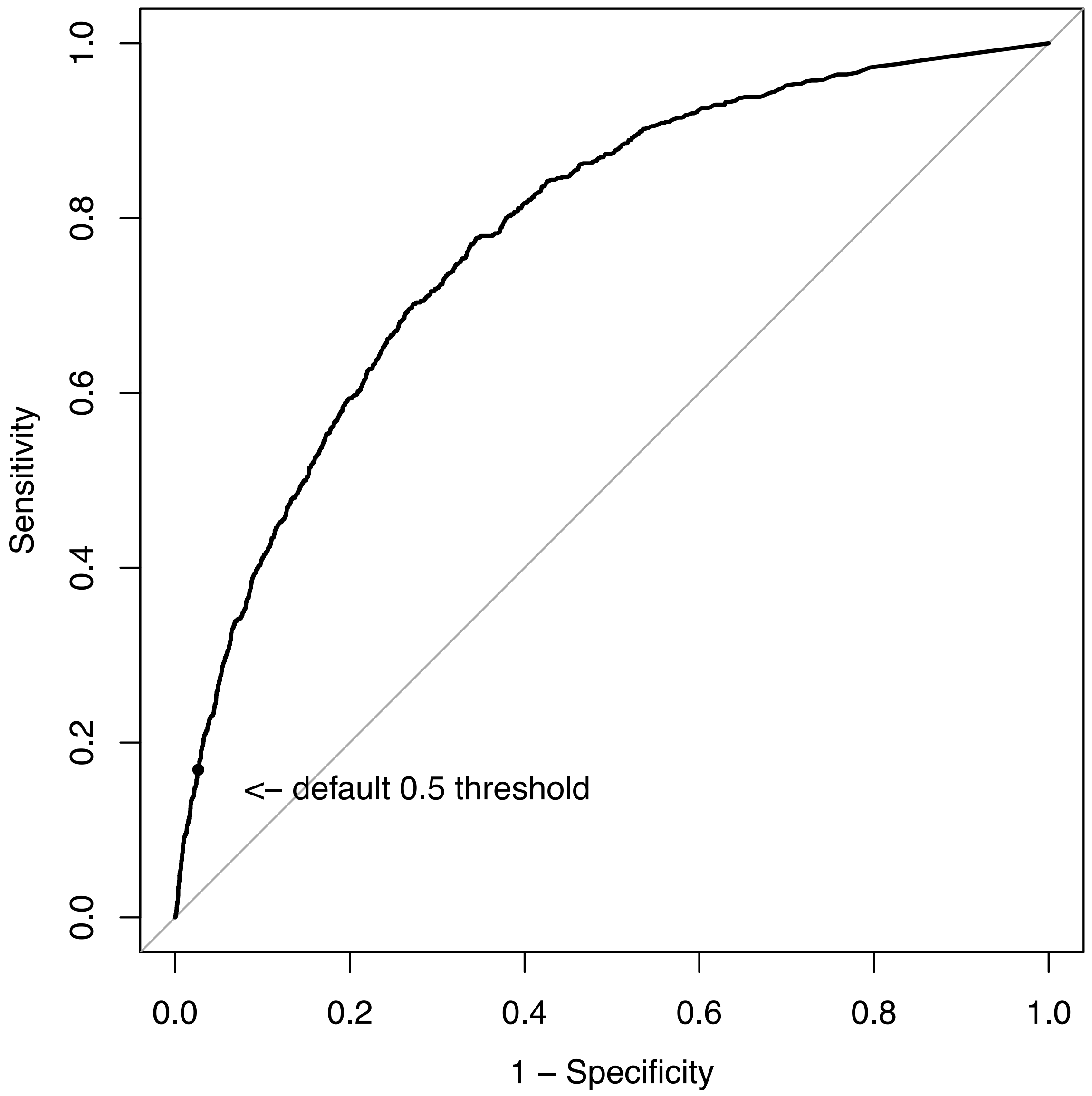
- sensitivity: 17.4%
- specificity: 97.2%

However, this does *not* indicate that better sensitivity and specificity combinations cannot be achieved using this model.

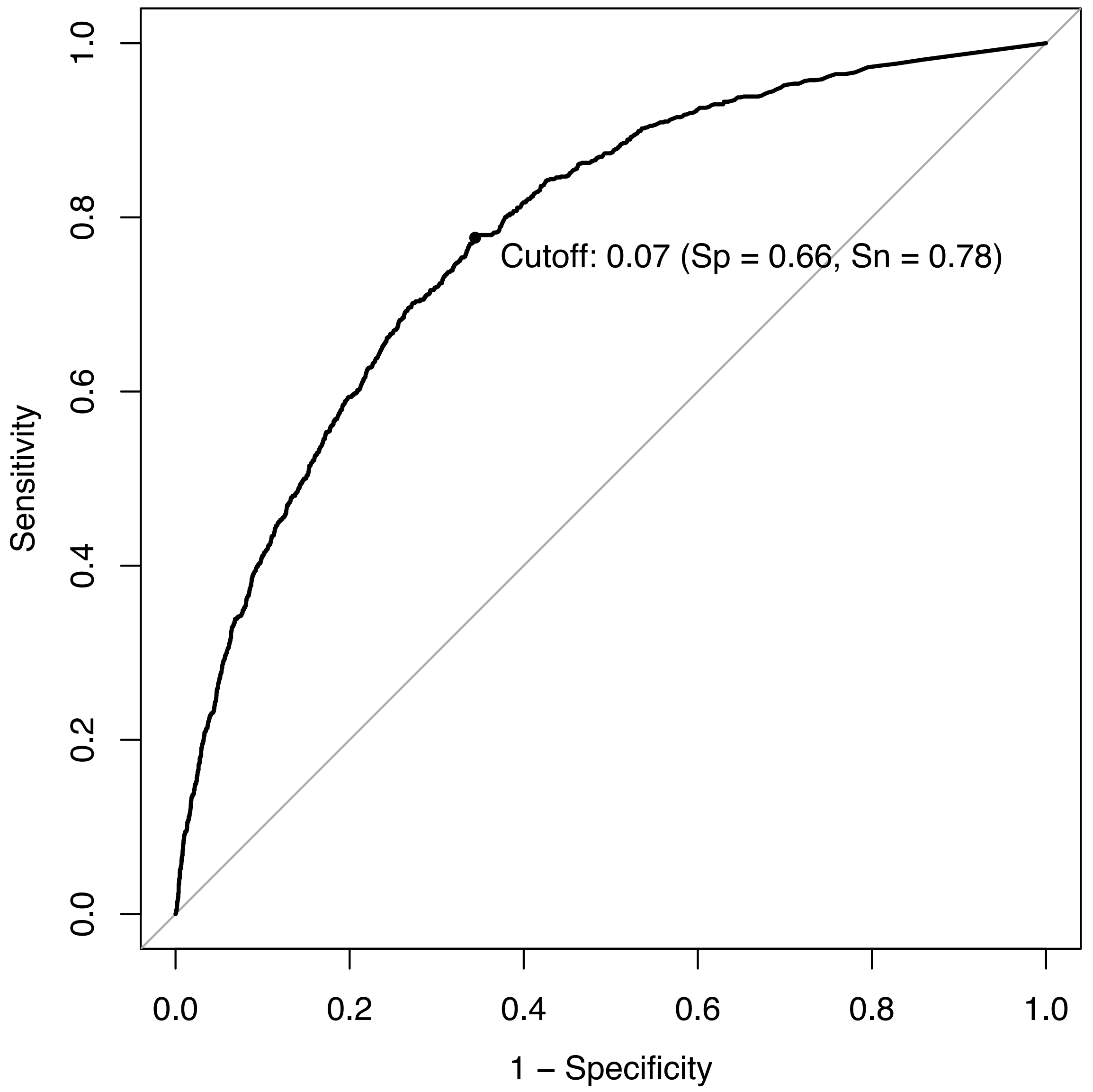
Random Forest Results — EMR Example



Approximate Random Forest Resampled ROC Curve



A Better Cutoff



Sampling for Class Imbalances

(APM Section 16.8)

Sampling Approaches

One view of the class imbalance problem focuses on the sheer difference in data points. The problem was illustrated with the previous CART model.

A simple approach to resolving this is the attempt to re–balance the data. Some approaches are:

- eliminate some of the majority class samples
- "enhance" the minor class samples
- a combination of the two

These occur *outside* of the model fit and are referred to as *external sampling* here.

We'll consider these in-turn and see how they impact the overall modeling process.

Sampling and Resampling

These sampling approaches should be embedded in any resampling used to tune the model or estimation performance:

- The sampling techniques can add variation to the model and the resampled performance estimates should reflect this noise.
- When a minority class sample is "replicated" during sampling, all of it's cloned children should stay in either the training *or* test set. Otherwise, the model ends up predicting a sample that is exactly the same (or highly similar to) one that was used to build the model.

An example of the second case can be found on the caret website.

Incorporating Additional Sampling

```
Define sets of model parameter values to evaluate;
for each parameter set do
    for each resampling iteration do
        Hold–out specific samples ;
        Sample the remainder;
        Fit the model on the sampled data;
        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;
```

Down–Sampling

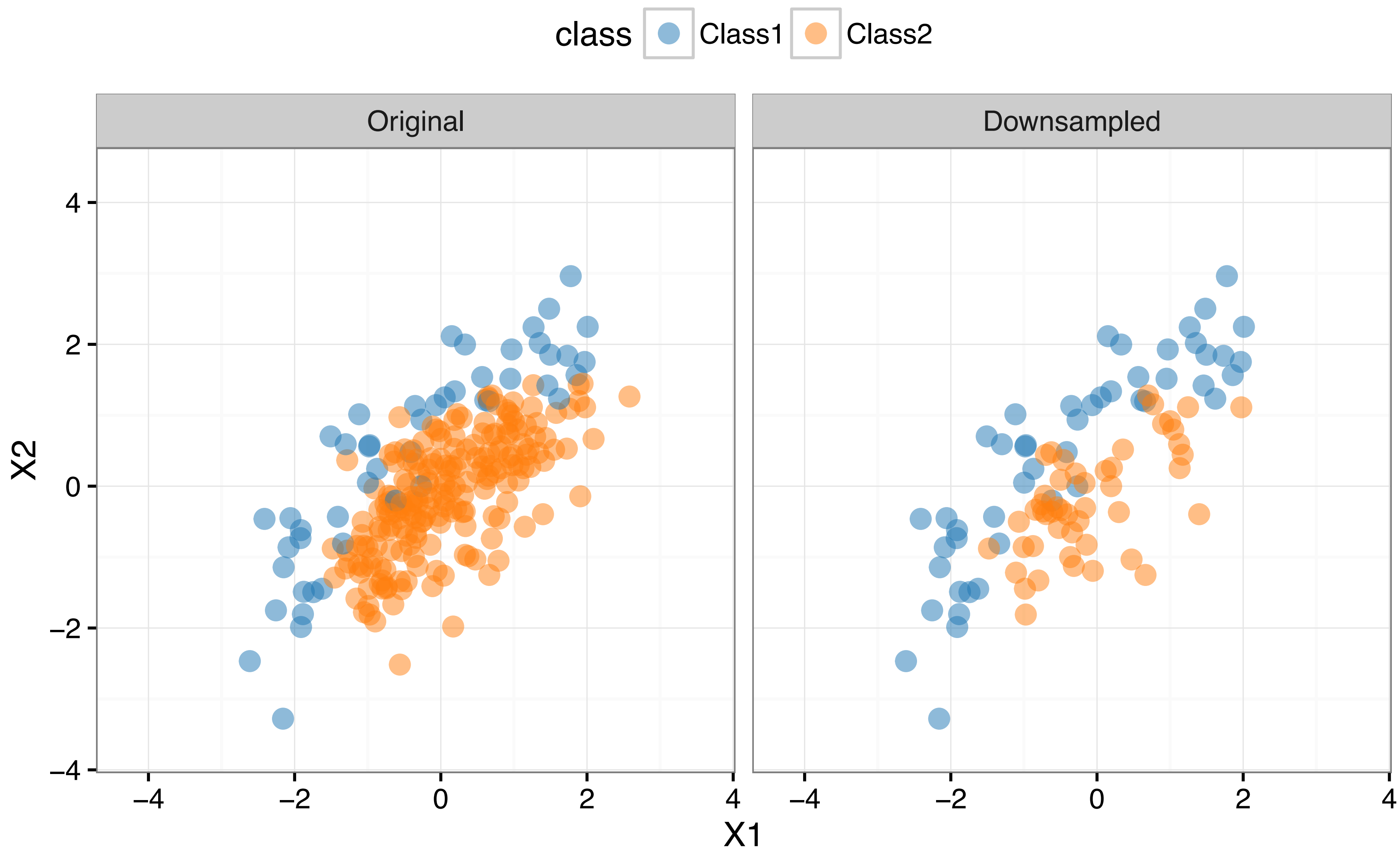
The majority class data are randomly sampled to be the same size as the smallest class.

Eliminating majority class data can have a profoundly positive effect on the model fit.

However, we are throwing away a lot of data.

Alternatively, we can create a ensemble bagging–like model that builds model with different samples of the majority class data. Model predictions could be averaged across the constituent model.

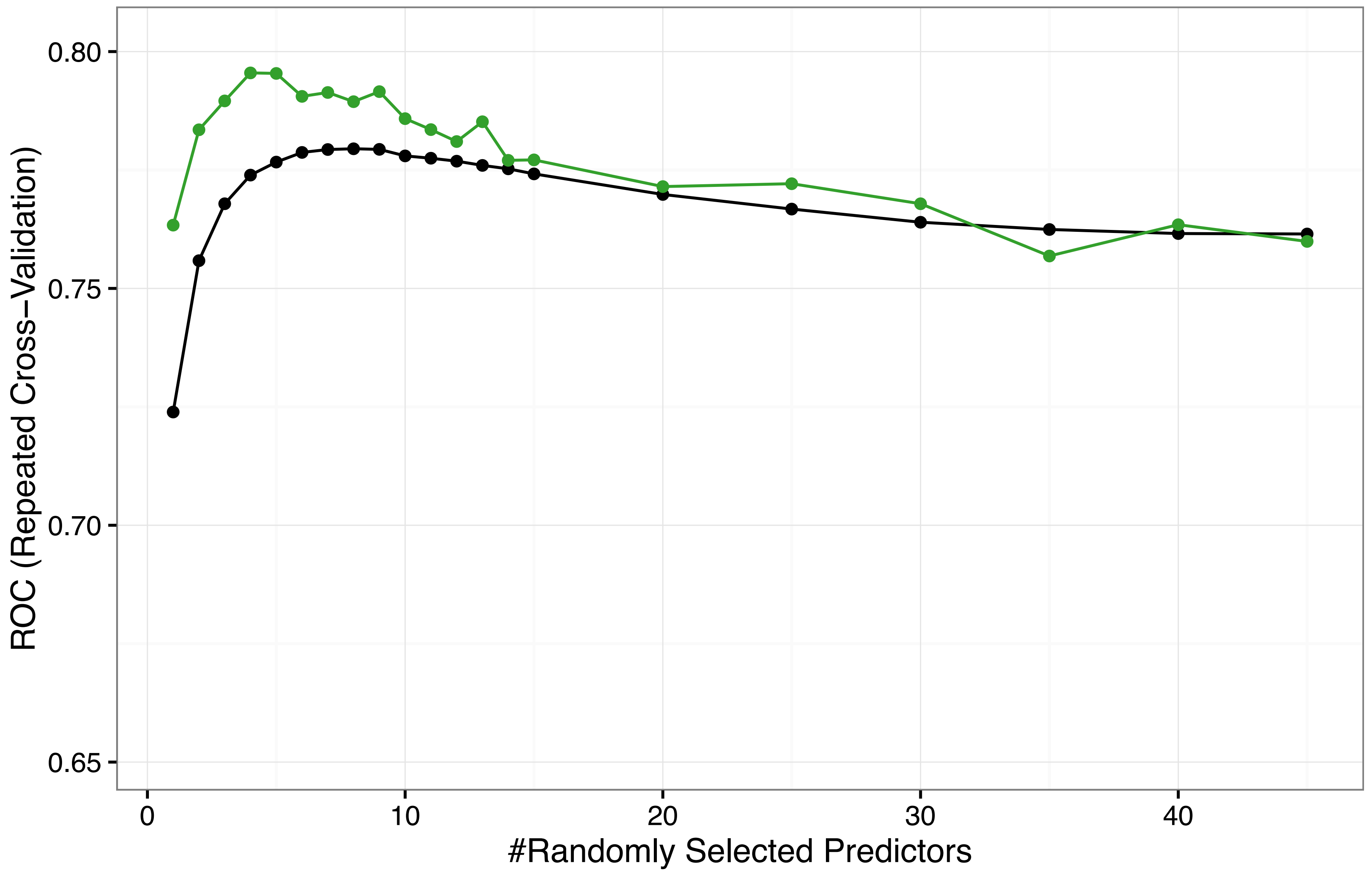
Down–Sampling — Simulated Data



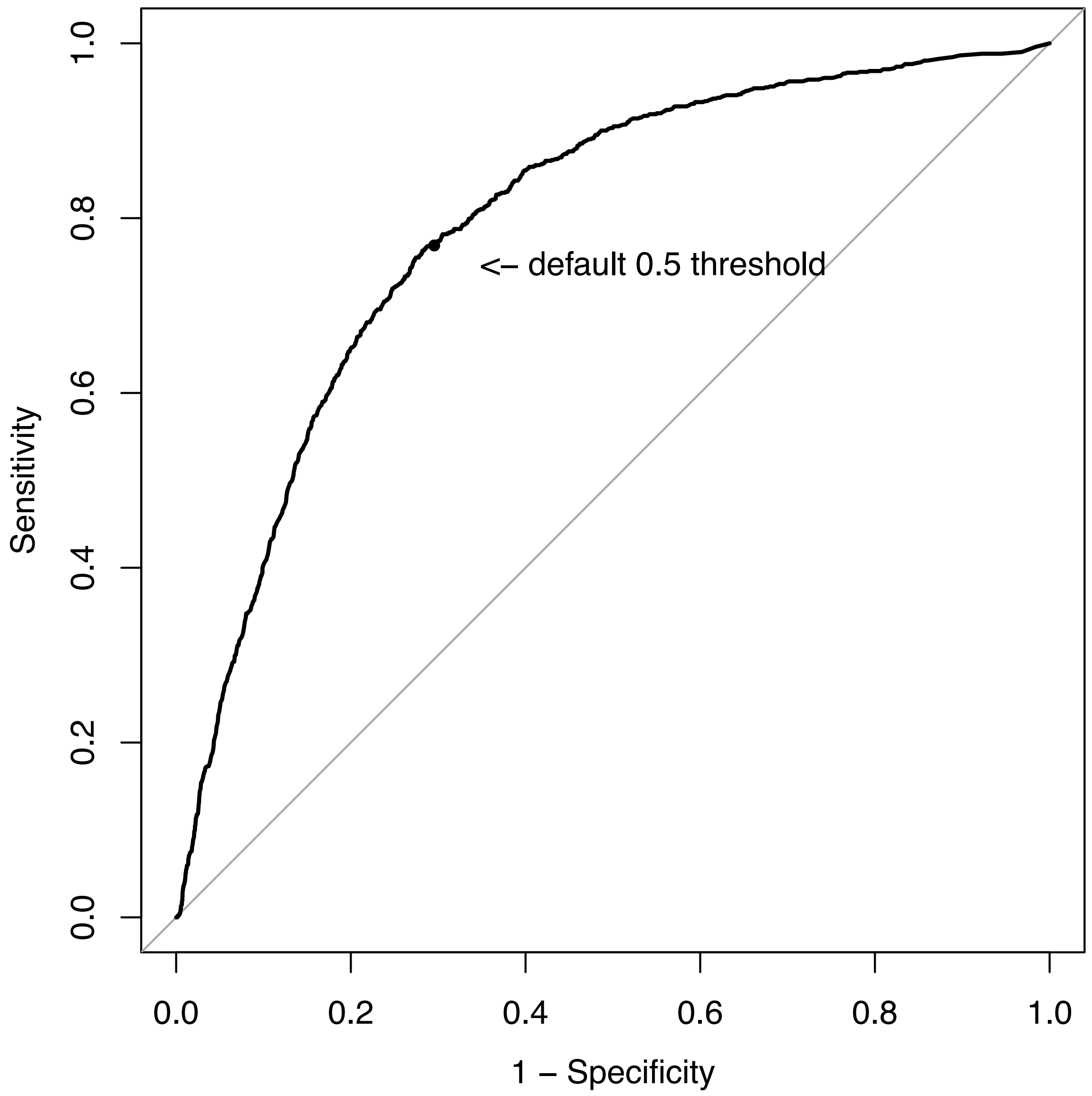
Down–Sampling — EMR Data

```
> down_ctrl <- ctrl
> down_ctrl$sampling <- "down"
> set.seed(1537)
> rf_emr_down <- train(Class ~ .,
+                      data = emr_train,
+                      method = "rf",
+                      metric = "ROC",
+                      tuneGrid = emr_grid,
+                      ntree = 1000,
+                      trControl = down_ctrl)
```

Down–Sampling — EMR Data



Approximate Resampled ROC Curve with Down–Sampling



Internal Sampling Using Random Forest

Recall that random forest fits a large number of independent trees.

Rather than using one external down-sample to rebalance the data, each individual tree could be down sampled to provide more exposure to the majority class samples.

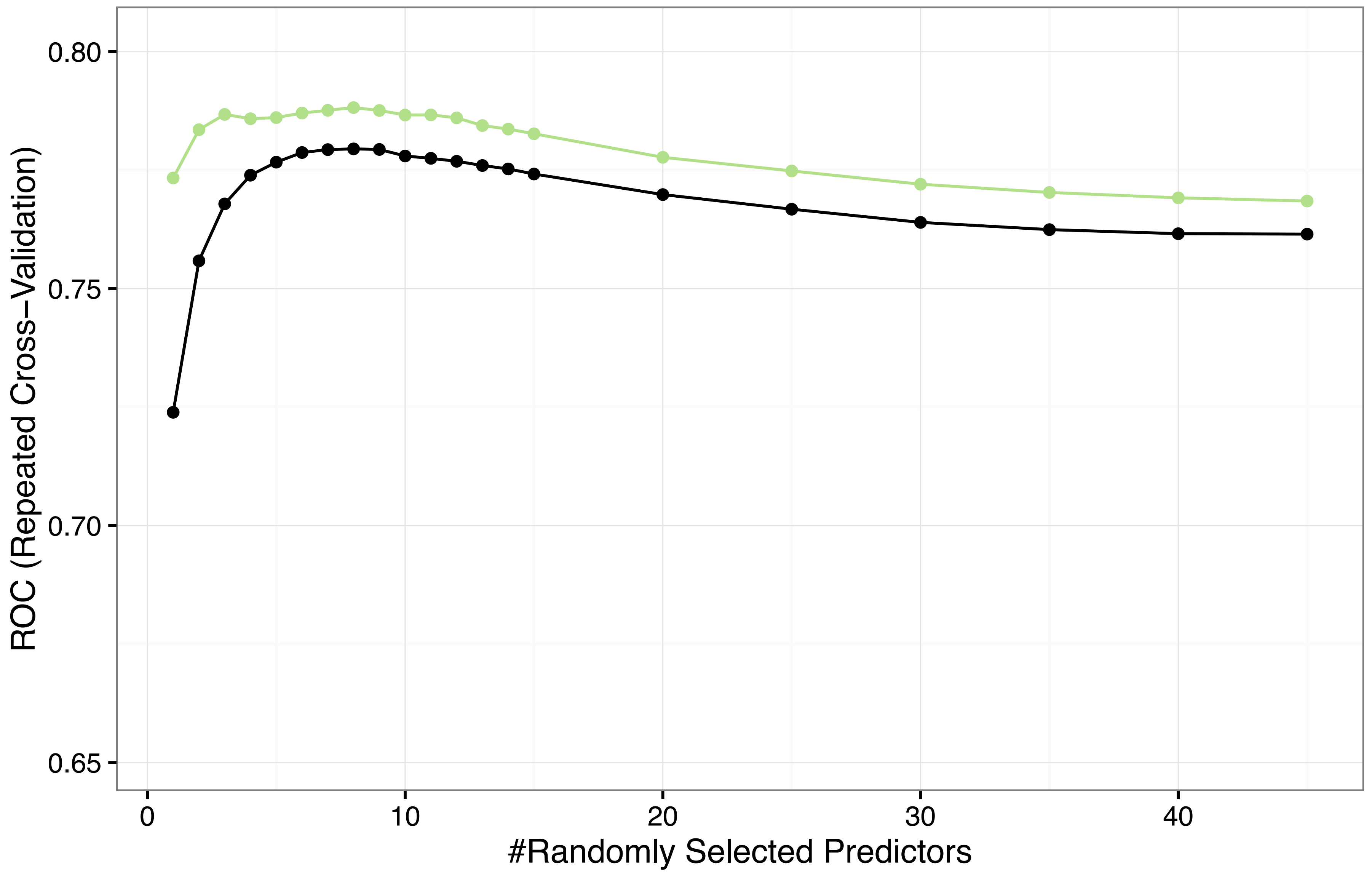
We will refer to this as *internal* down–sampling.

This does about the sample as external down–sampling but has somewhat reduced model variance.

Internal Down–Sampling — EMR Data

```
> set.seed(1537)
> rf_emr_down_int <- train(Class ~ .,
+                          data = emr_train,
+                          method = "rf",
+                          metric = "ROC",
+                          ntree = 1000,
+                          tuneGrid = emr_grid,
+                          trControl = ctrl,
+                          ## These are passed to `randomForest`
+                          strata = emr_train$Class,
+                          sampsize = rep(sum(emr_train$Class == "event"), 2))
```

Internal Down–Sampling — EMR Data



Up–Sampling

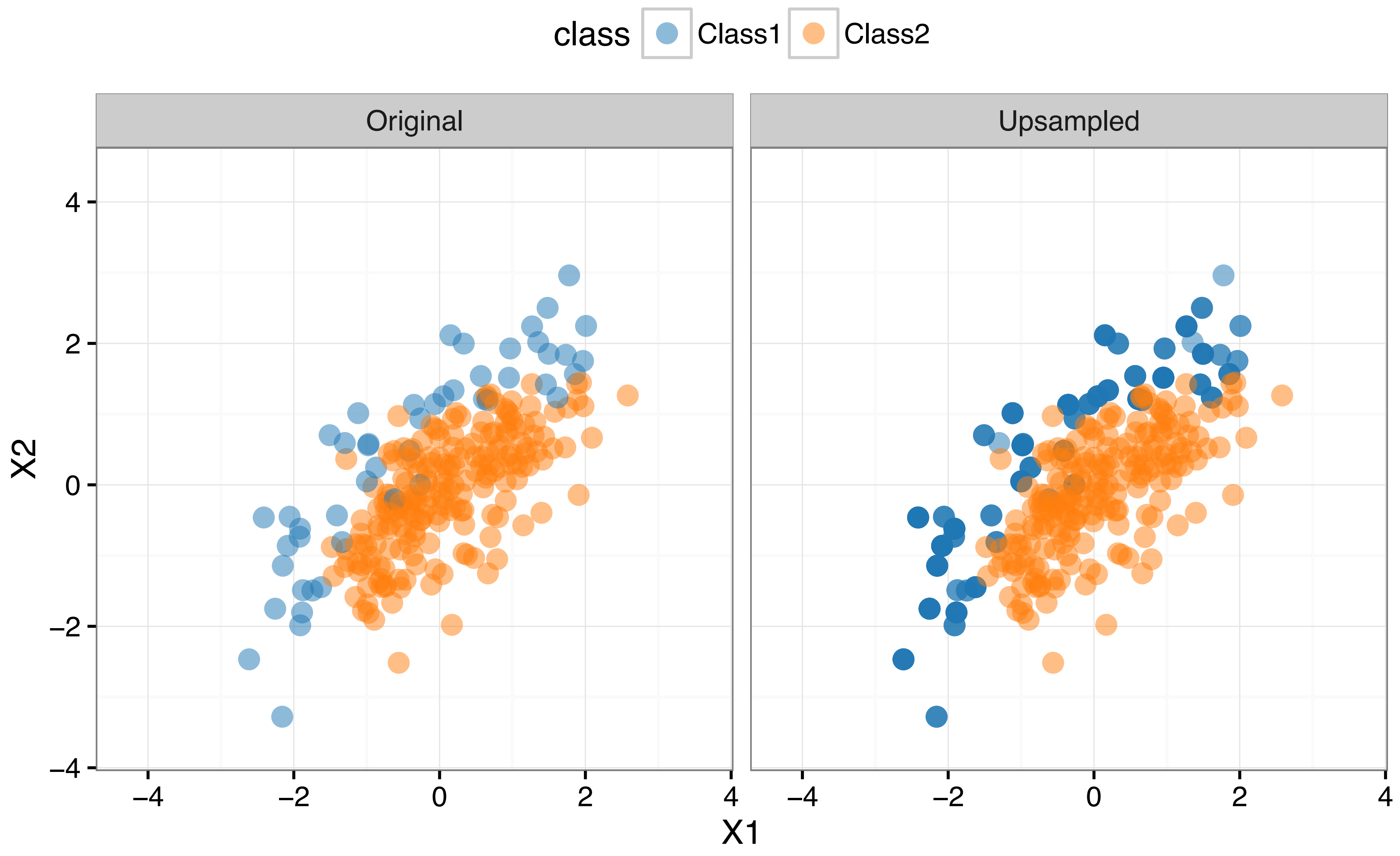
In this case, we randomly sample from the majority class to add additional replicates.

This is equivalent in some cases to increasing the case weights of the minority class.

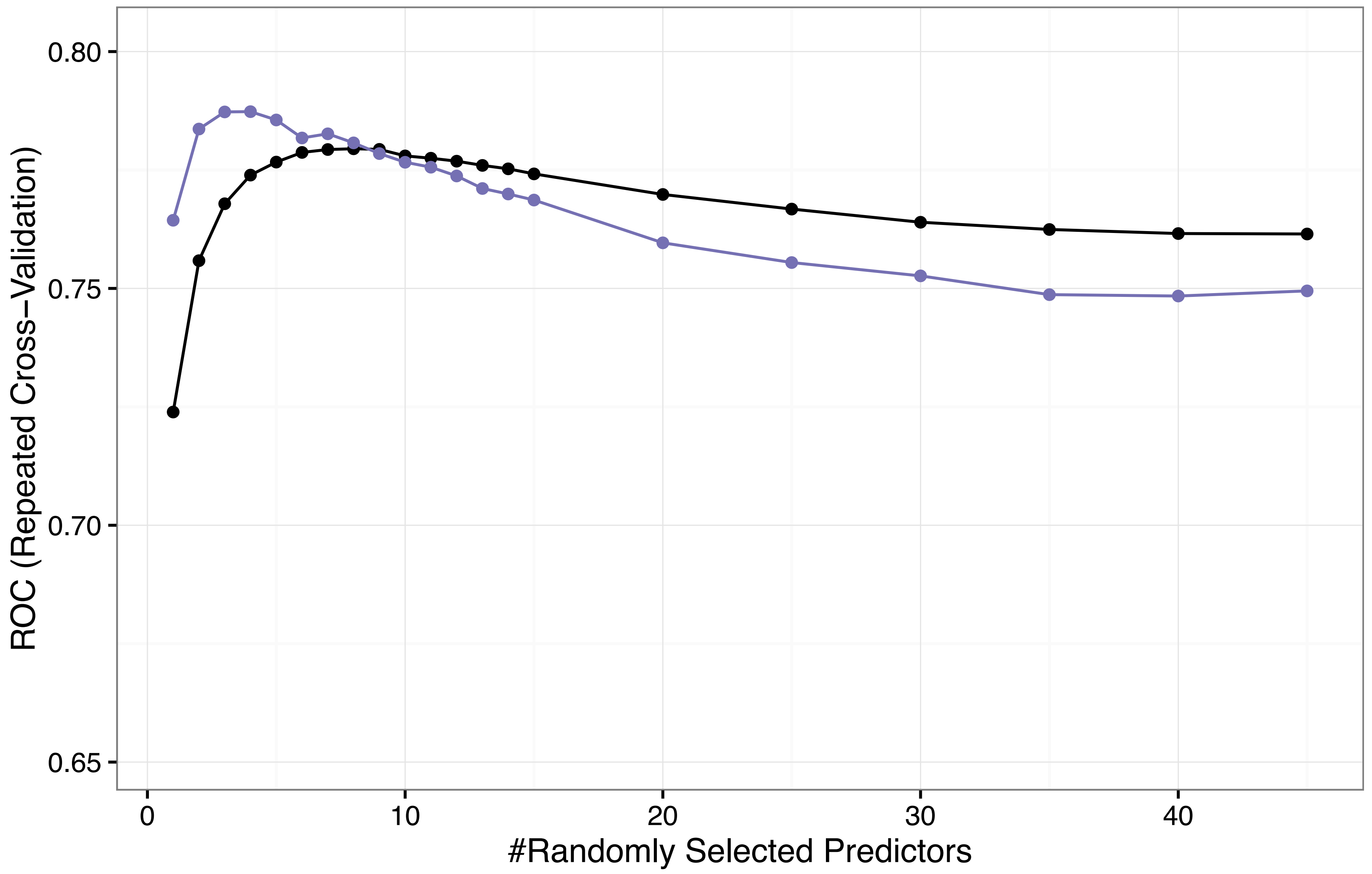
Up–sampling can wreak havoc with resampling when this is done prior to model fitting. The same sample can fall into the modeling and holdout sets, generating highly optimistic performance estimates.

Again, this last point is demonstrated on the caret package's website: http://bit.ly/1Vru6X1.

Up–Sampling — Simulated Data



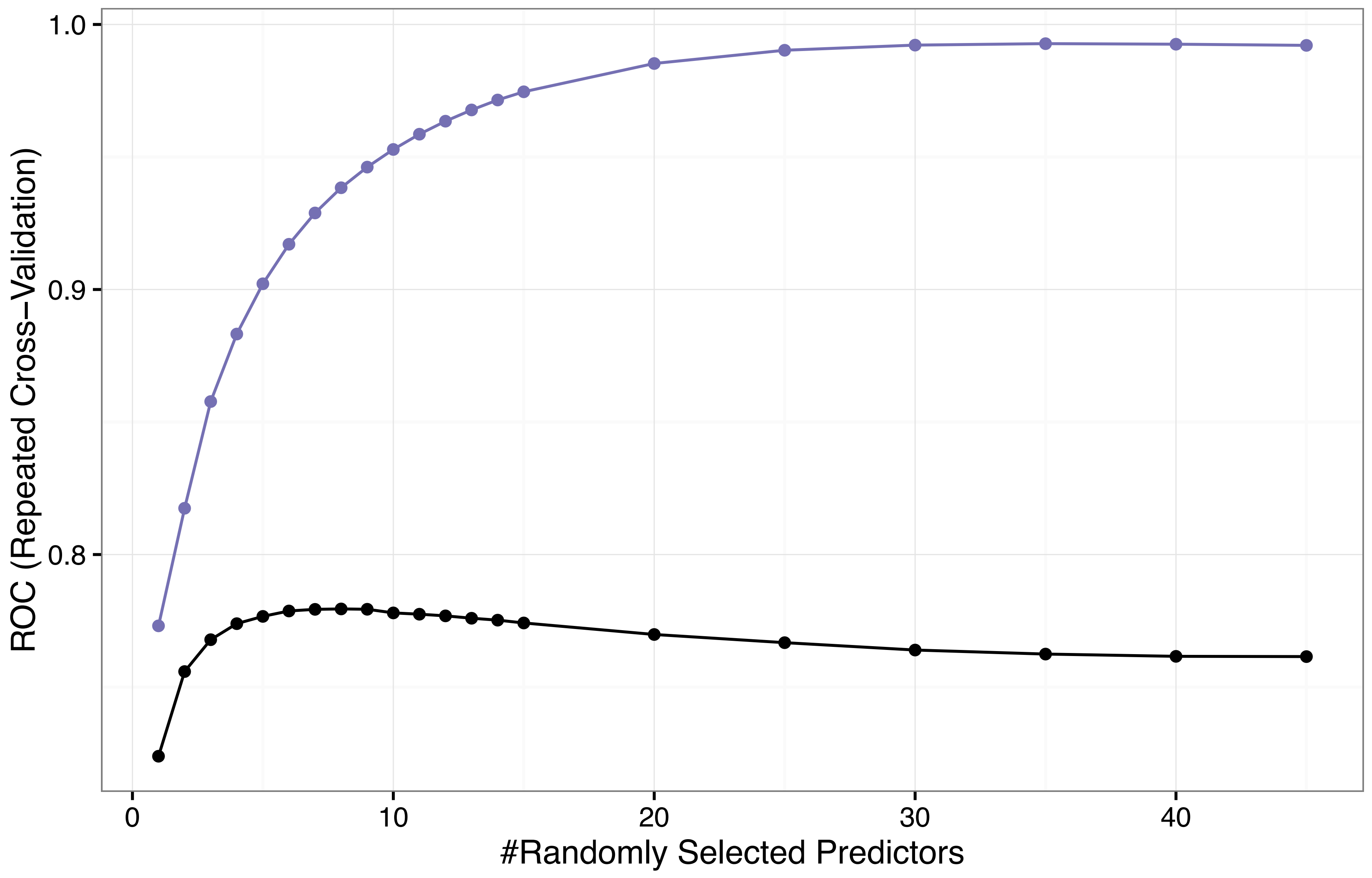
Up–Sampling — EMR Data



Up–Sampling — EMR Data

```
> up_ctrl <- ctrl
> up_ctrl$sampling <- "up"
> set.seed(1537)
> rf_emr_up <- train(Class ~ .,
+                    data = emr_train,
+                    method = "rf",
+                    tuneGrid = emr_grid,
+                    ntree = 1000,
+                    metric = "ROC",
+                    trControl = up_ctrl)
```

Up–Sampling Wrongly Done Prior to Resampling



Hybrid Approaches

These approaches combine downsampling with the artificial synthesis of new data points form the majority class.

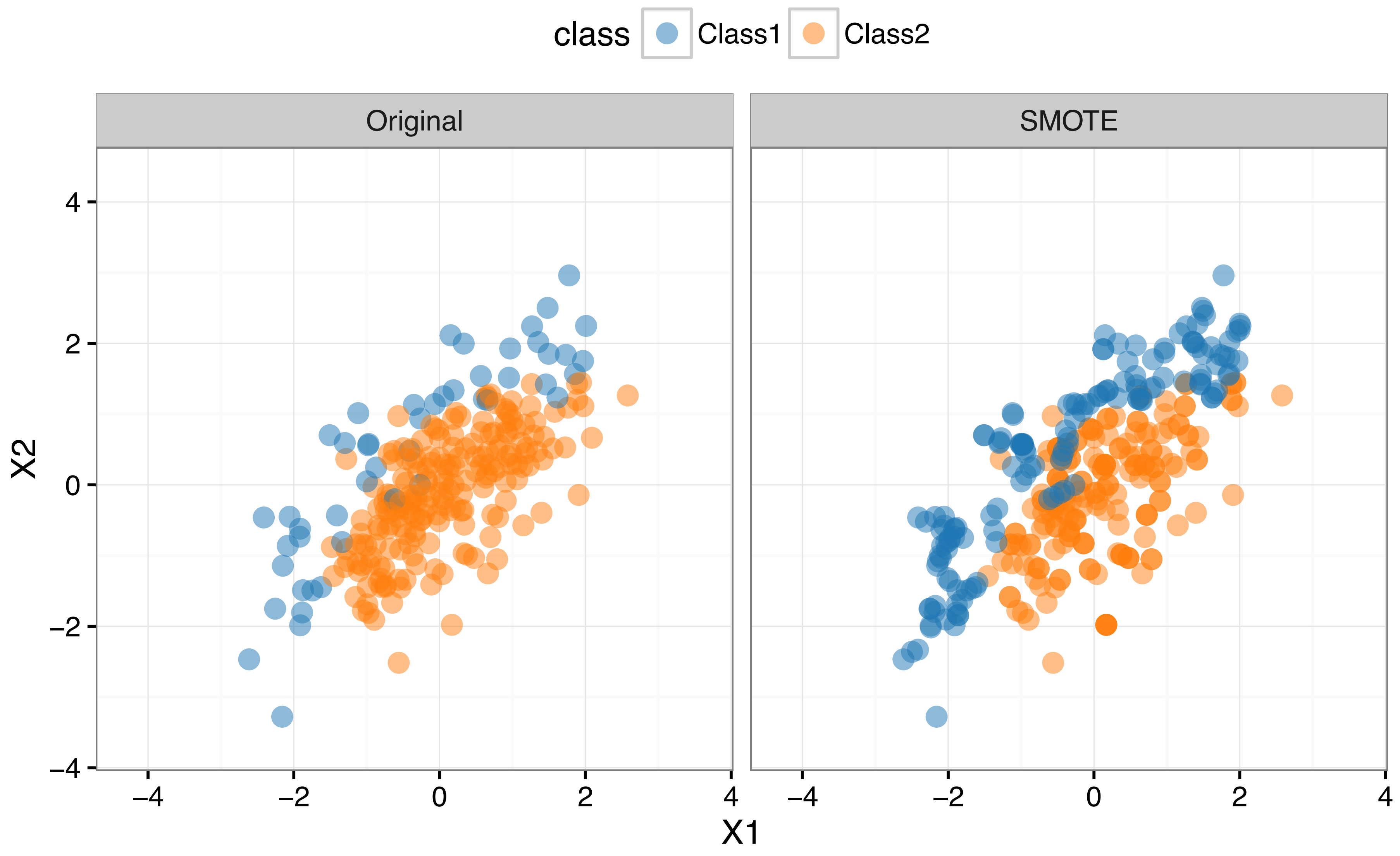
SMOTE (Synthetic Minority Over–Sampling TechniquE) and ROSE (Randomly Over Sampling Examples) use the data in the training set to create new artificial data points that are similar to the existing ones.

SMOTE uses nearest–neighbors for this purpose while ROSE uses kernel density estimation.

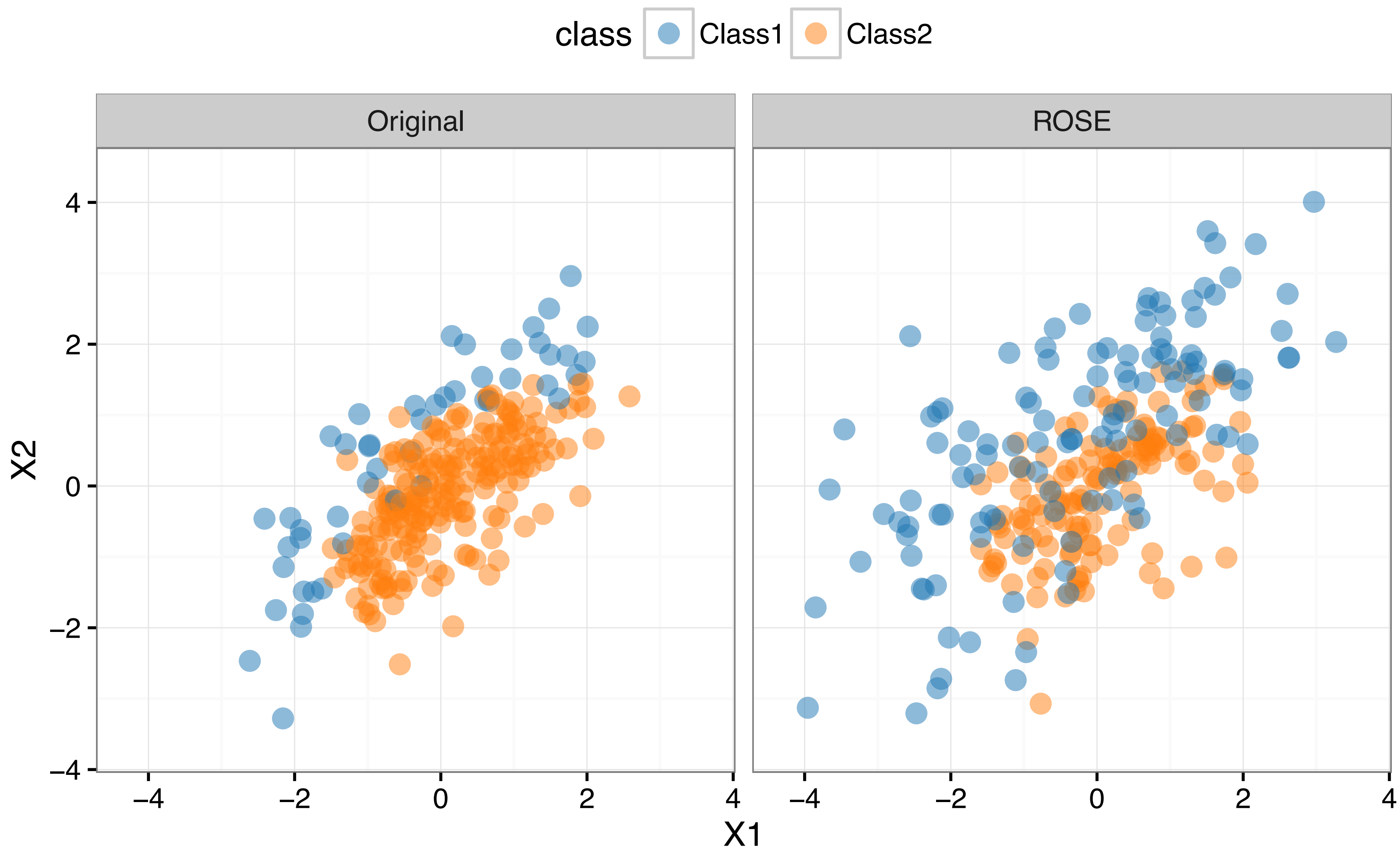
ROSE is less effective when the training set consists of mostly discrete predictors.

There are many other techniques for class sampling.

SMOTE — Simulated Data



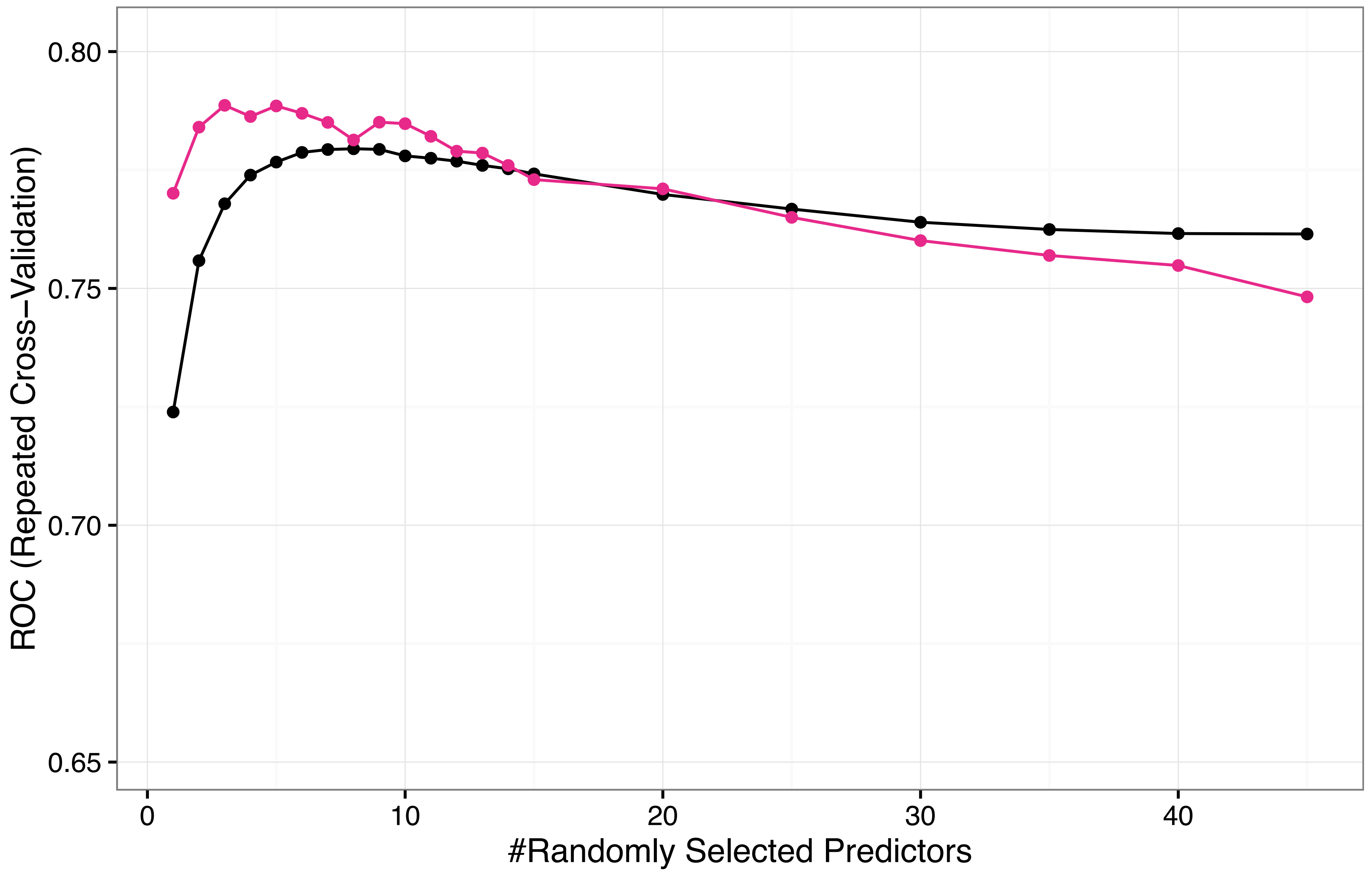
ROSE — Simulated Data



SMOTE — EMR Data

```
> smote_ctrl <- ctrl
> smote_ctrl$sampling <- "smote"
> set.seed(1537)
> rf_emr_smote <- train(Class ~ .,
+                       data = emr_train,
+                       method = "rf",
+                       tuneGrid = emr_grid,
+                       ntree = 1000,
+                       metric = "ROC",
+                       trControl = smote_ctrl)
```

SMOTE — EMR Data



EMR Test Set Results

```
> emr_test_pred <- data.frame(Class = emr_test$Class)
> emr_test_pred$normal <- predict(rf_emr_mod, emr_test, type = "prob")[, "event"]
> emr_test_pred$down <- predict(rf_emr_down, emr_test, type = "prob")[, "event"]
> emr_test_pred$down_int <- predict(rf_emr_down_int, emr_test, type = "prob")[, "ev
> emr_test_pred$up <- predict(rf_emr_up, emr_test, type = "prob")[, "event"]
> emr_test_pred$smote <- predict(rf_emr_smote, emr_test, type = "prob")[, "event"]
>
> get_auc <- function(pred, ref) auc(roc(ref, pred, levels = rev(levels(ref))))
>
> apply(emr_test_pred[, -1], 2, get_auc, ref = emr_test_pred$Class)
   normal      down  down_int        up     smote
0.7740987 0.7976243 0.7958548 0.7922414 0.7903370
```

Sampling Observations

It's been my experience that sub–sampling can have a positive effect in mitigating the imbalance issues.

However, that effect is often small and not consistent with the magnitudes seen in the literature.

At a minimum, these methods help generate *well–calibrated* probabilities that have a good balance between sensitivity and specificity.

Your mileage may vary

Cost–Sensitive Models

(APM Section 16.9)

Using Costs

One issue with large class imbalances is that the types of errors are treated equally.

In aggregate, this means that the errors for the majority class have the most impact on typical objective functions (e.g. accuracy).

Cost–sensitive methods allow for asymmetric costs for specific types of errors. For example, we might weight the cost of a false–negative to be 10–fold higher than false–positives to improve sensitivity.

The potential cost of a prediction takes into account several factors:

- the cost of the particular mistake s.t. $C(j|i)$ be the cost of mistakenly predicting a class i sample as class j
- the probability of making a specific mistake ($Pr[j|i]$)
- the prior probability of the classes (π_i)

Using Costs

Without costs, the first class would be predicted when

$$\frac{p_1}{p_2} > \frac{\pi_2}{\pi_1}$$

With costs:

$$\text{Expected Cost} = C(2|1)Pr[2|1]\pi_1 + C(1|2)Pr[1|2]\pi_2$$

The new rule would be:

$$\frac{p_1}{p_2} > \left(\frac{C(1|2)}{C(2|1)}\right)\left(\frac{\pi_2}{\pi_1}\right)$$

Costs and Trees

Tree–based models are easily adapted for using costs.

For CART, a modified Gini–index is used:

$$\begin{aligned} Gini^* &= C(1|2)p_1(1-p_1) + C(2|1)p_2(1-p_2) \\ &= \left[C(1|2) + C(2|1)\right] p_1 p_2 \end{aligned}$$

For C5.0, the cost is directly quantified and minimized during splitting and boosting.

Unfortunately, both of these methods invalidate the class probability predictions under the current implementations.

CART and Costs – OkC Data

```
> fourStats <- function (data, lev = levels(data$obs), model = NULL) {
+   accKapp <- postResample(data[, "pred"], data[, "obs"])
+   out <- c(accKapp,
+            sensitivity(data[, "pred"], data[, "obs"], lev[1]),
+            specificity(data[, "pred"], data[, "obs"], lev[2]))
+   names(out)[3:4] <- c("Sens", "Spec")
+   out
+ }
>
> ctrl_cost <- trainControl(method = "repeatedcv",
+                           repeats = 5,
+                           savePredictions = TRUE,
+                           summaryFunction = fourStats)
```

CART and Costs – OkC Data

```
> ## Get an initial grid of Cp values
> rpart_init <- rpart(Class ~ ., data = okc_train, cp = 0)$cptable
>
> cost_grid <- expand.grid(cp = rpart_init[, "CP"], Cost = 1:5)
>
> ## Use the non-formula method. Many of the predictors are factors and
> ## this will preserve the factor encoding instead of using dummy
> ## variables.
>
> set.seed(1537)
> rpart_costs <- train(x = okc_train[, names(okc_train) != "Class"],
+                      y = okc_train$Class,
+                      method = "rpartCost",
+                      tuneGrid = cost_grid,
+                      metric = "Kappa",
+                      trControl = ctrl_cost)
```

CART and Costs – OkC Data

We tuned a CART model over the complexity parameter C_p and cost values between 1 and 5

The same tuning process was used as before except that the Kappa statistic was used to pick the final model.

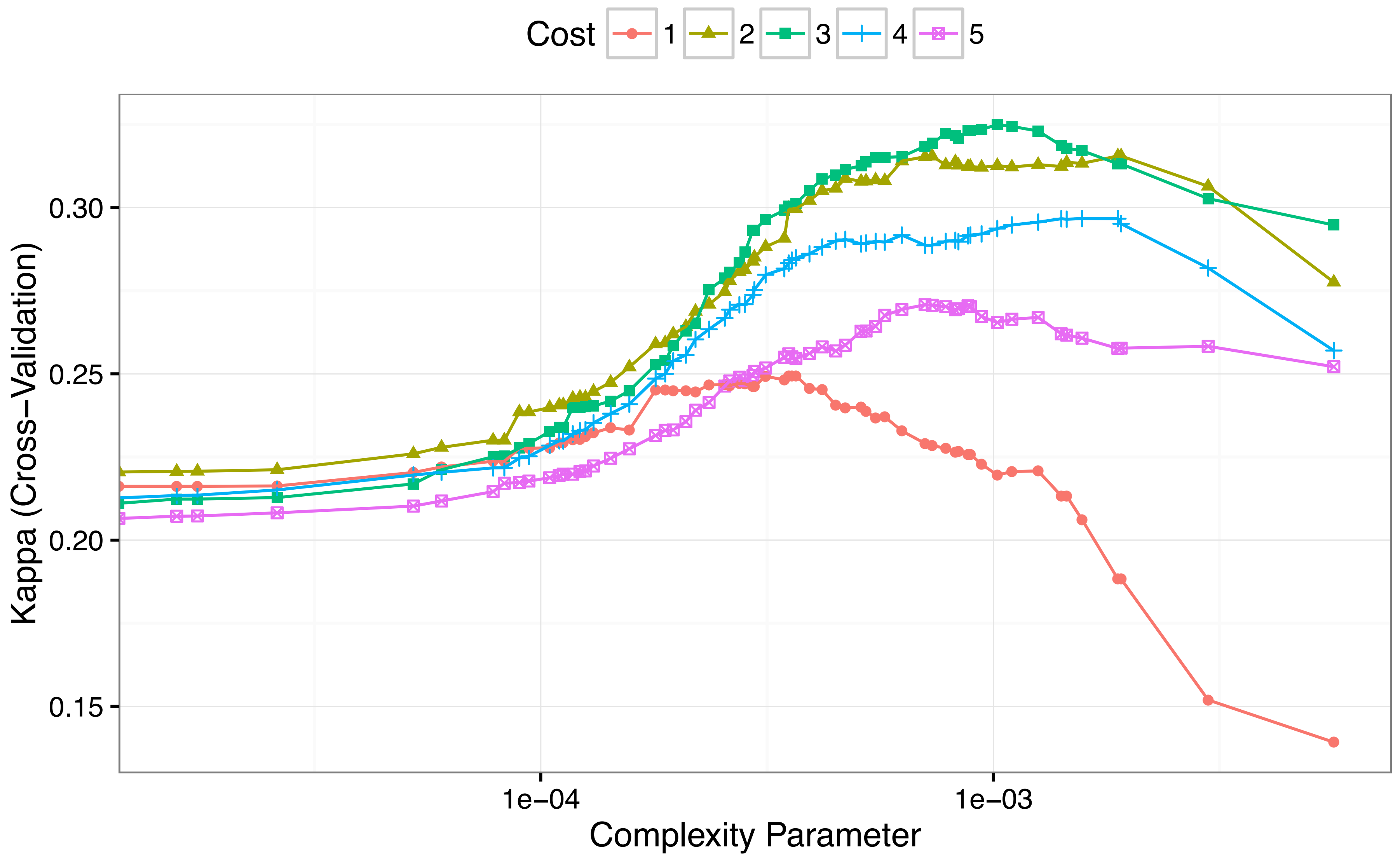
The optimal model used a complexity parameter value of 0.00102 (33 splits) and a cost value of 3.

In doing so, Kappa improved from 0.249 to 0.325.

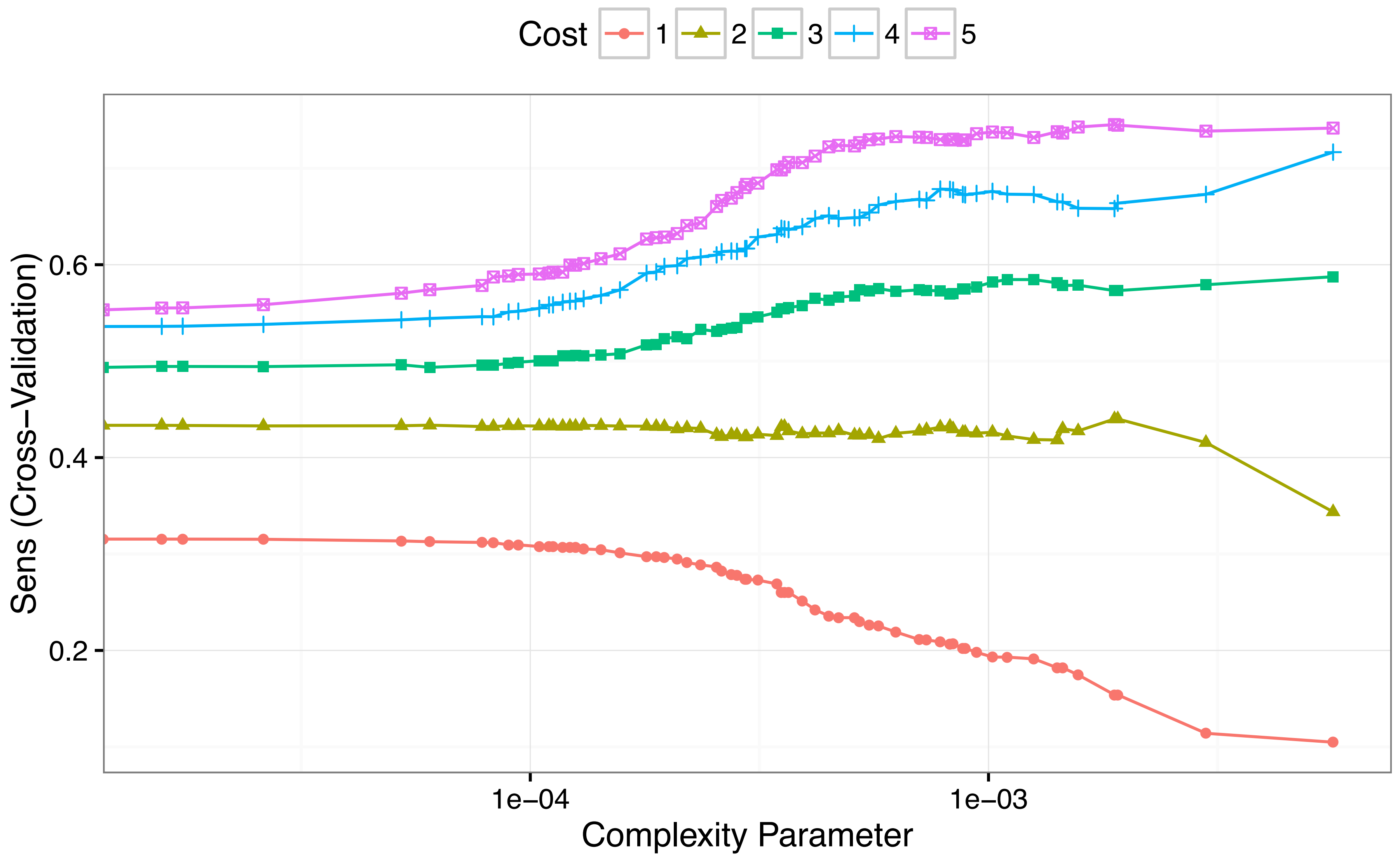
Interestingly, the equal cost model had increased performance with deeper trees; the cost–based models preferred simpler trees.

This is consistent with the remarks from Breiman *et al* (1984).

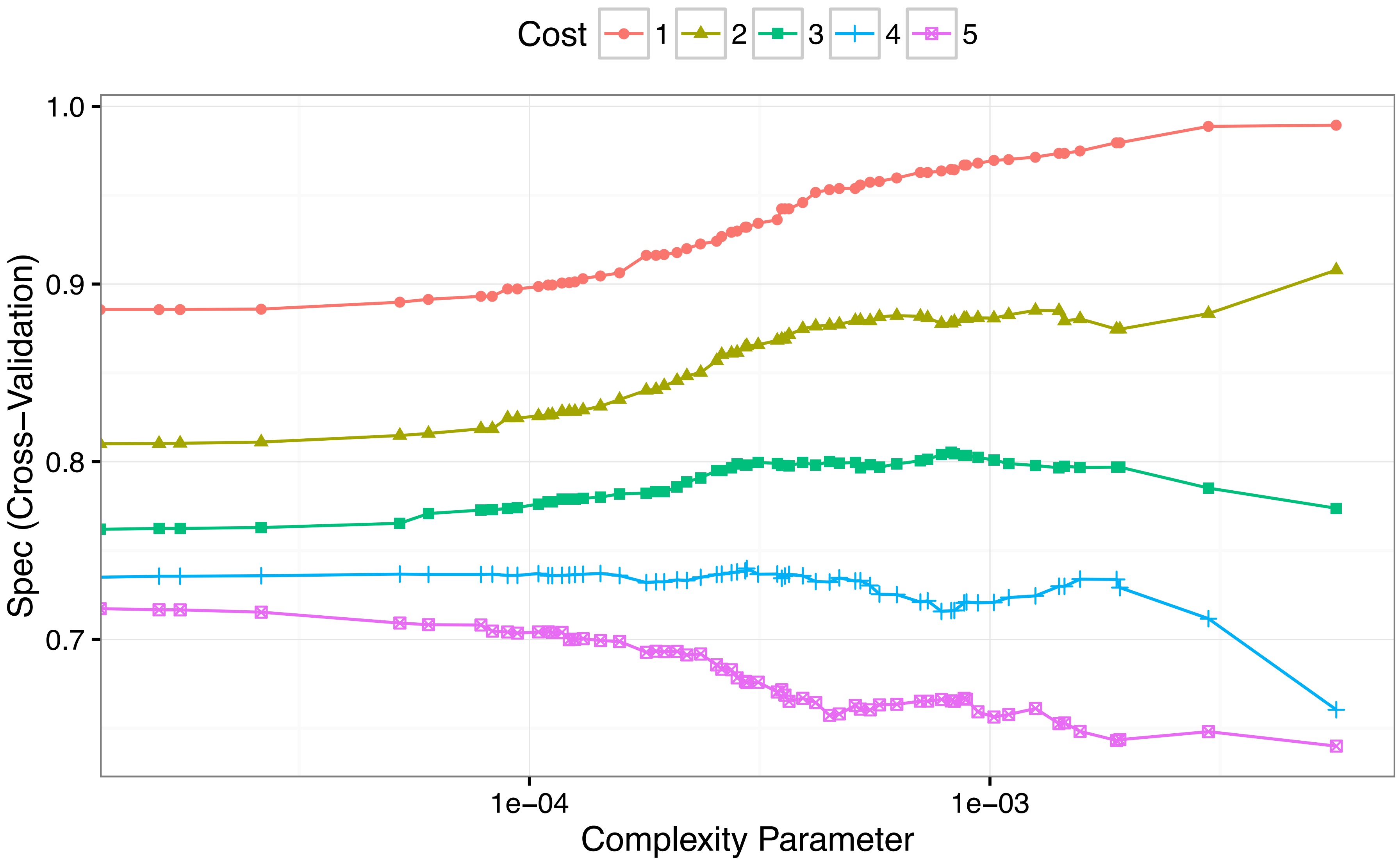
CART and Costs – OkC Data



CART and Costs – OkC Data



CART and Costs – OkC Data



C5.0 and Costs – OkC Data

```
> cost_grid <- expand.grid(trials = c(1:10, 20, 30),
+                          winnow = FALSE, model = "tree",
+                          cost = c(1, 5, 10, 15))
> set.seed(1537)
> c5_costs <- train(x = okc_train[, names(okc_train) != "Class"],
+                   y = okc_train$Class,
+                   method = "C5.0Cost",
+                   tuneGrid = cost_grid,
+                   metric = "Kappa",
+                   trControl = ctrl_cost)
```

C5.0 and Costs– OkC Data

We tuned a C5.0 tree–based model over the number of boosting iterations (1 to 30) and cost values between 1 and 15.

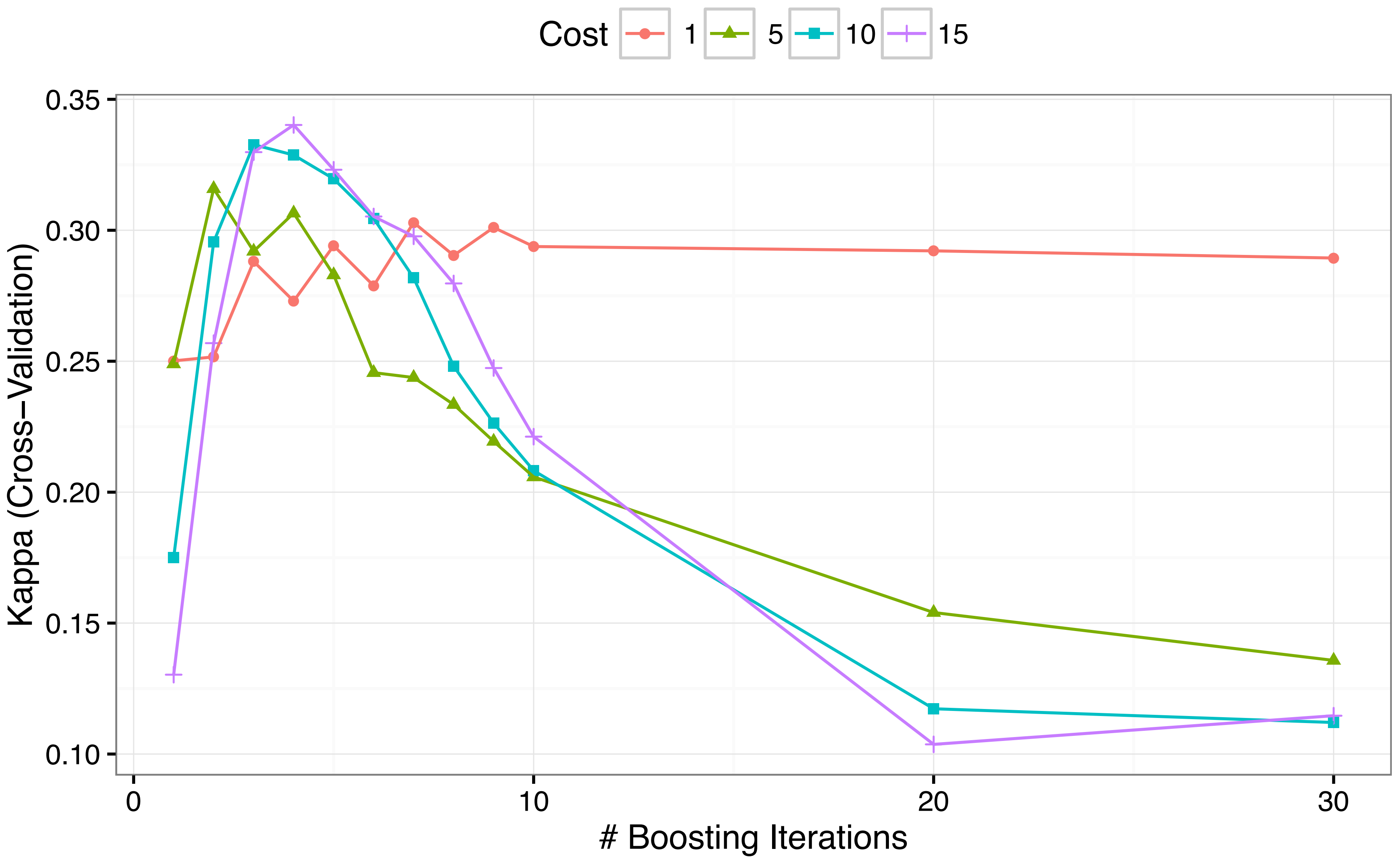
The same tuning process was used with CART.

The optimal model used 4 iterations of boosting and a cost value of 15.

Here, Kappa improved from 0.303 to 0.34.

As with CART, simpler models were optimal when weights were used. Increasing the number of boosting iterations resulted in the need for larger cost values.

C5.0 and Costs – OkC Data



OkC Test Results – CART

```
> rp_pred <- predict(rpart_costs, newdata = okc_test)
> confusionMatrix(rp_pred, okc_test$Class)
Confusion Matrix and Statistics

          Reference
Prediction  stem other
     stem   1882  3020
     other  1303 11043

               Accuracy : 0.7494
                 95% CI : (0.7428, 0.7558)
    No Information Rate : 0.8153
    P-Value [Acc > NIR] : 1

                  Kappa : 0.3113
 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.5909
            Specificity : 0.7853
         Pos Pred Value : 0.3839
         Neg Pred Value : 0.8945
             Prevalence : 0.1847
         Detection Rate : 0.1091
   Detection Prevalence : 0.2842
      Balanced Accuracy : 0.6881

       'Positive' Class : stem
```

OkC Test Results – C5.0

```
> c5_pred <- predict(c5_costs, newdata = okc_test)
> confusionMatrix(c5_pred, okc_test$Class)
Confusion Matrix and Statistics

          Reference
Prediction  stem other
     stem   1917  2810
     other  1268 11253

               Accuracy : 0.7636
                 95% CI : (0.7572, 0.7699)
    No Information Rate : 0.8153
    P-Value [Acc > NIR] : 1

                  Kappa : 0.3387
 Mcnemar's Test P-Value : <2e-16

            Sensitivity : 0.6019
            Specificity : 0.8002
         Pos Pred Value : 0.4055
         Neg Pred Value : 0.8987
             Prevalence : 0.1847
         Detection Rate : 0.1111
   Detection Prevalence : 0.2741
      Balanced Accuracy : 0.7010

       'Positive' Class : stem
```

Support Vector Machines and Costs

Support vector machines are characterized by their *margin*, which is a buffer that runs parallel to the class boundary produce by the model.

The support vectors are samples that lay inside of, or on the wrong side of, the margin.

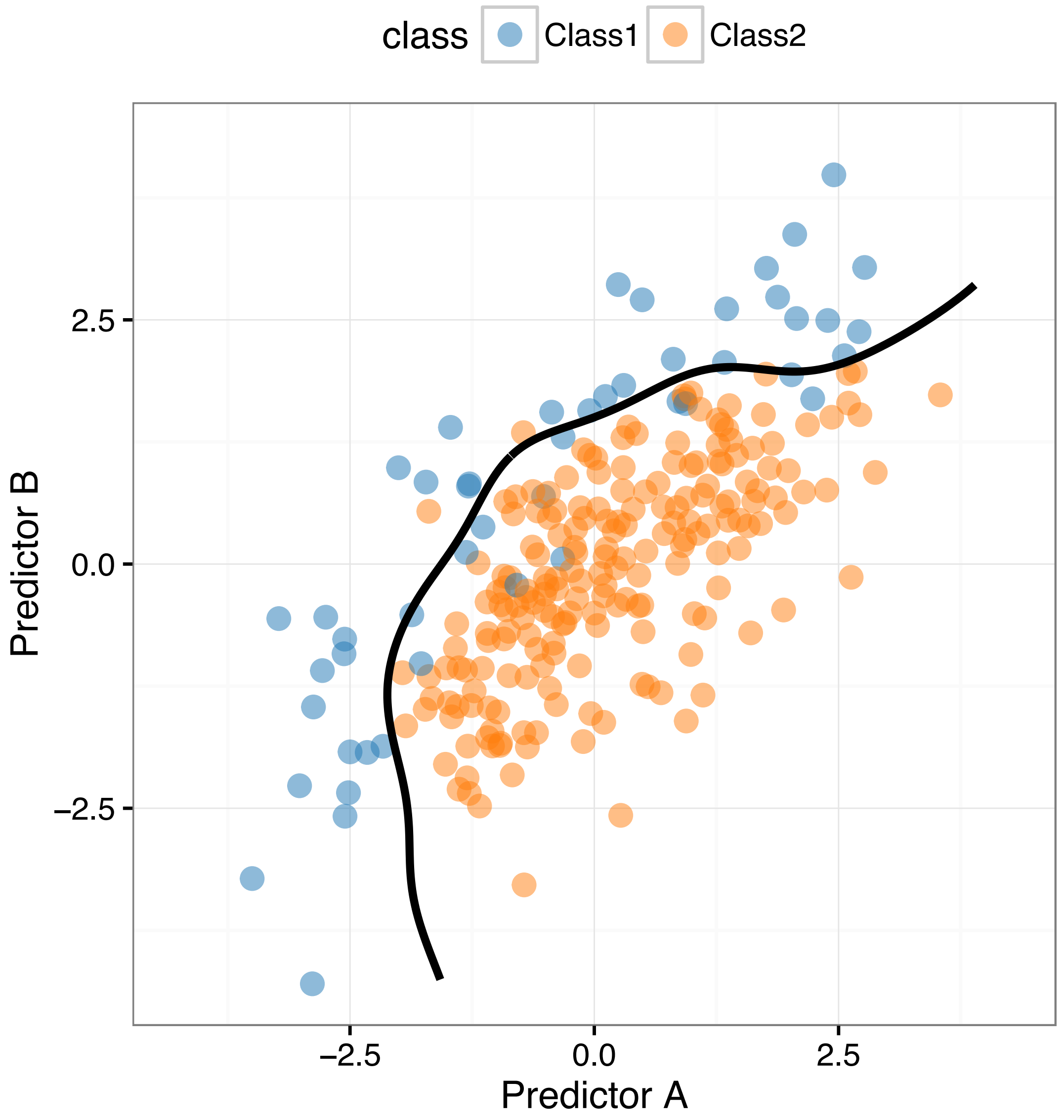
- the class boundary is *only a function of the support vectors*

To optimize these models, a cost is placed on samples that are on the

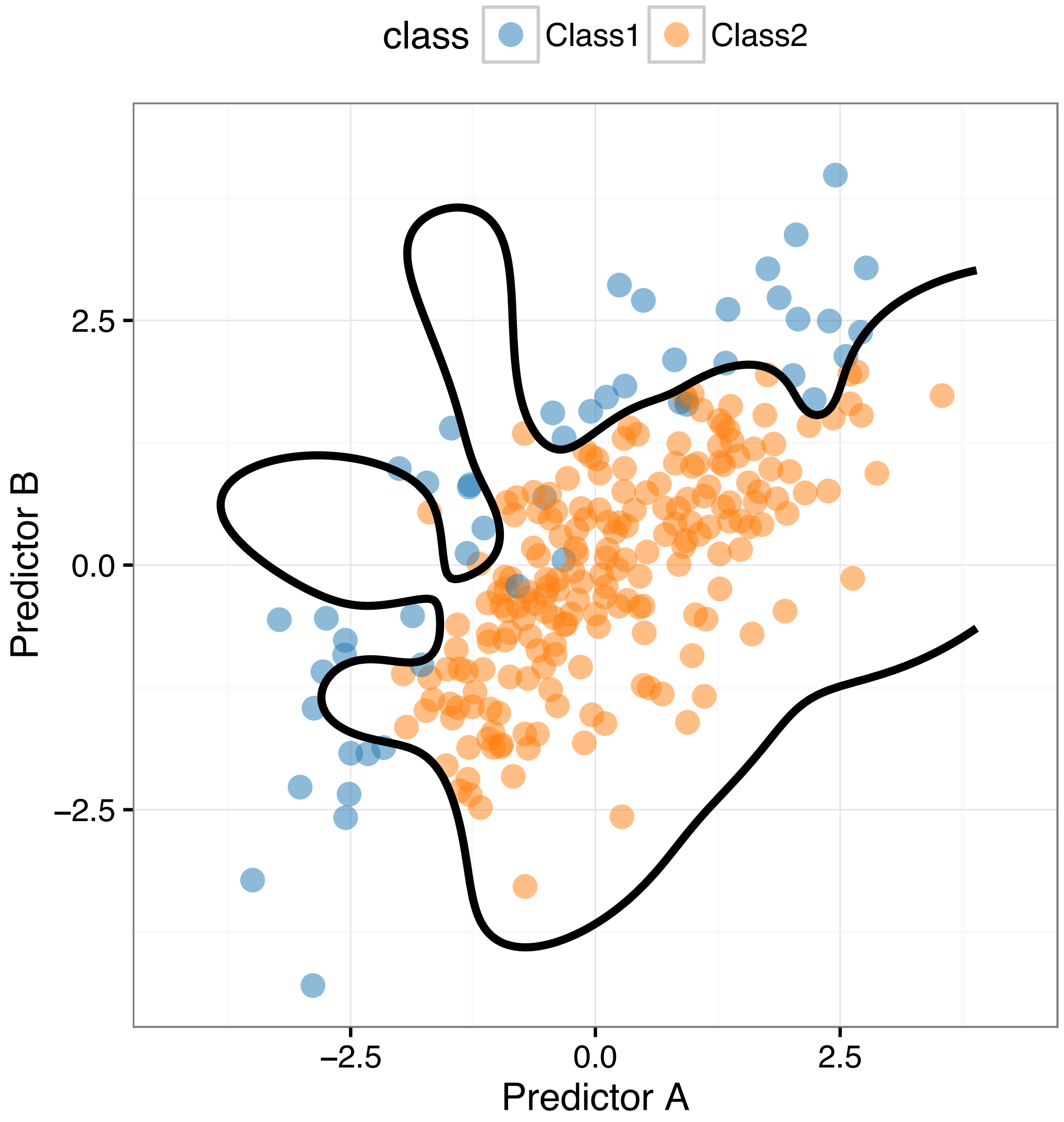
As the cost is very large, the SVM model will attempt to contort in ways to make sure that the *training set* data are accurately predicted.

Small costs values usually result in *underfit* models.

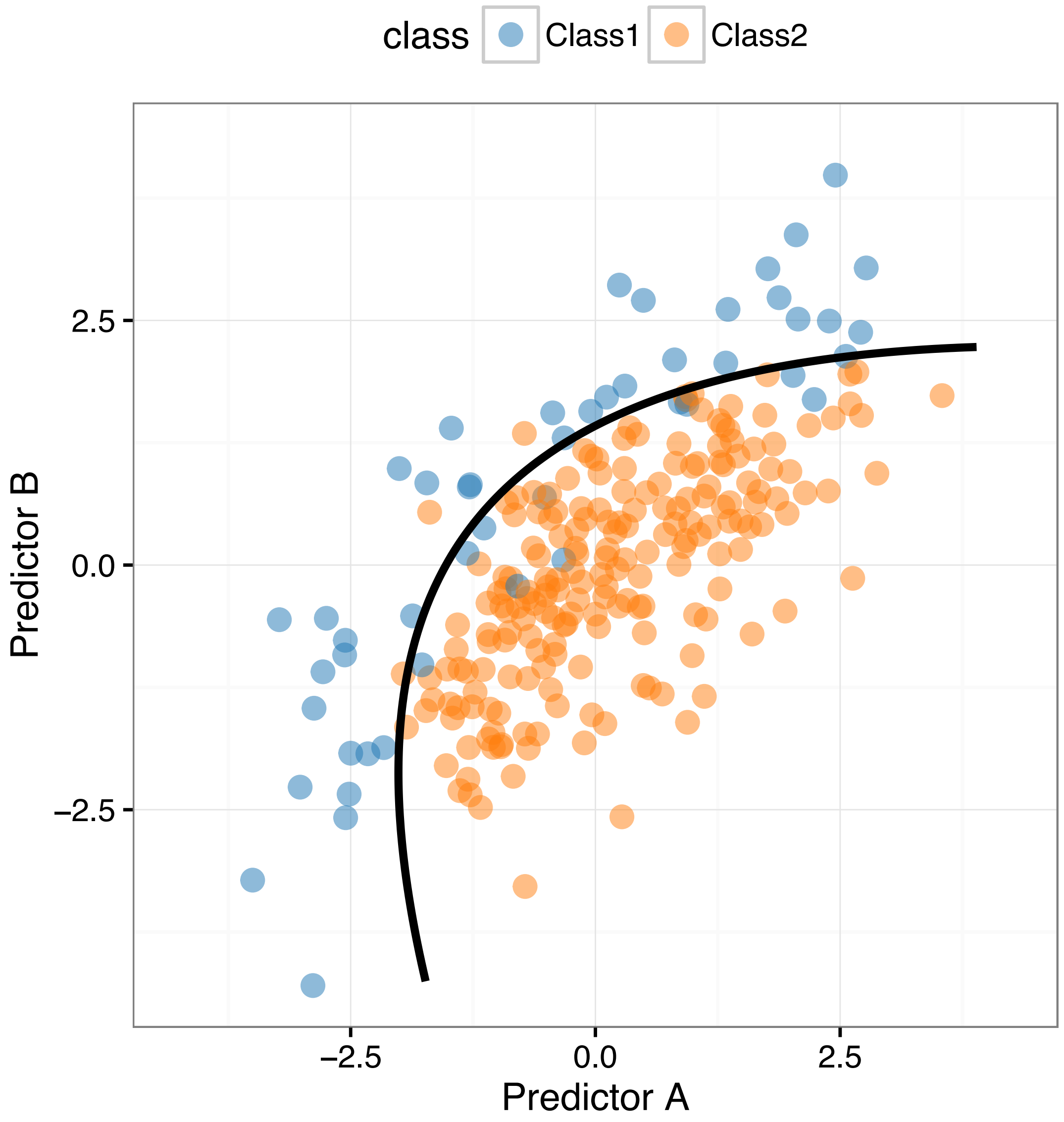
Reasonable Radial Basis Function Fit (Low Cost)



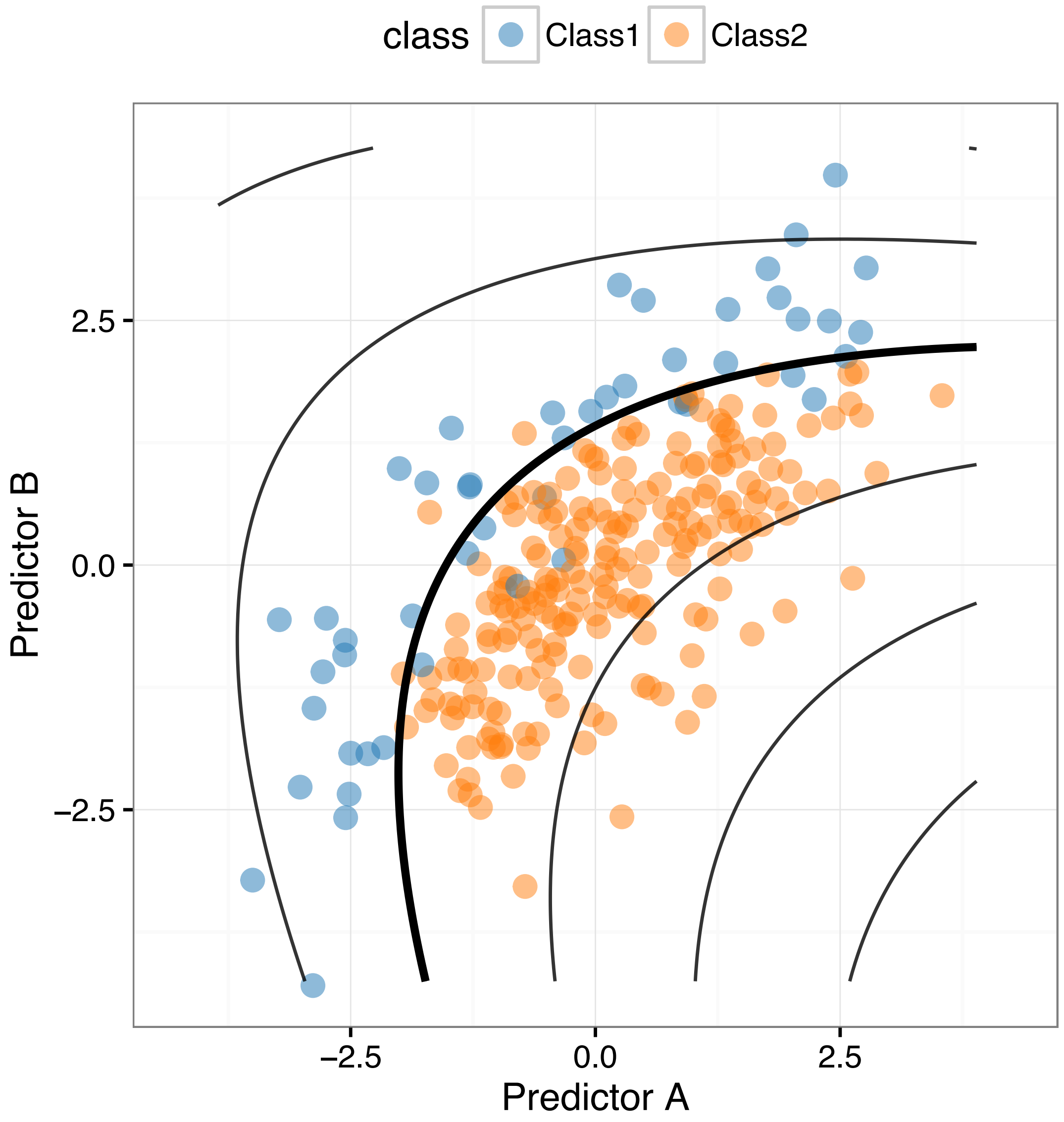
Radial Basis Function Trying Too Hard (High Cost)



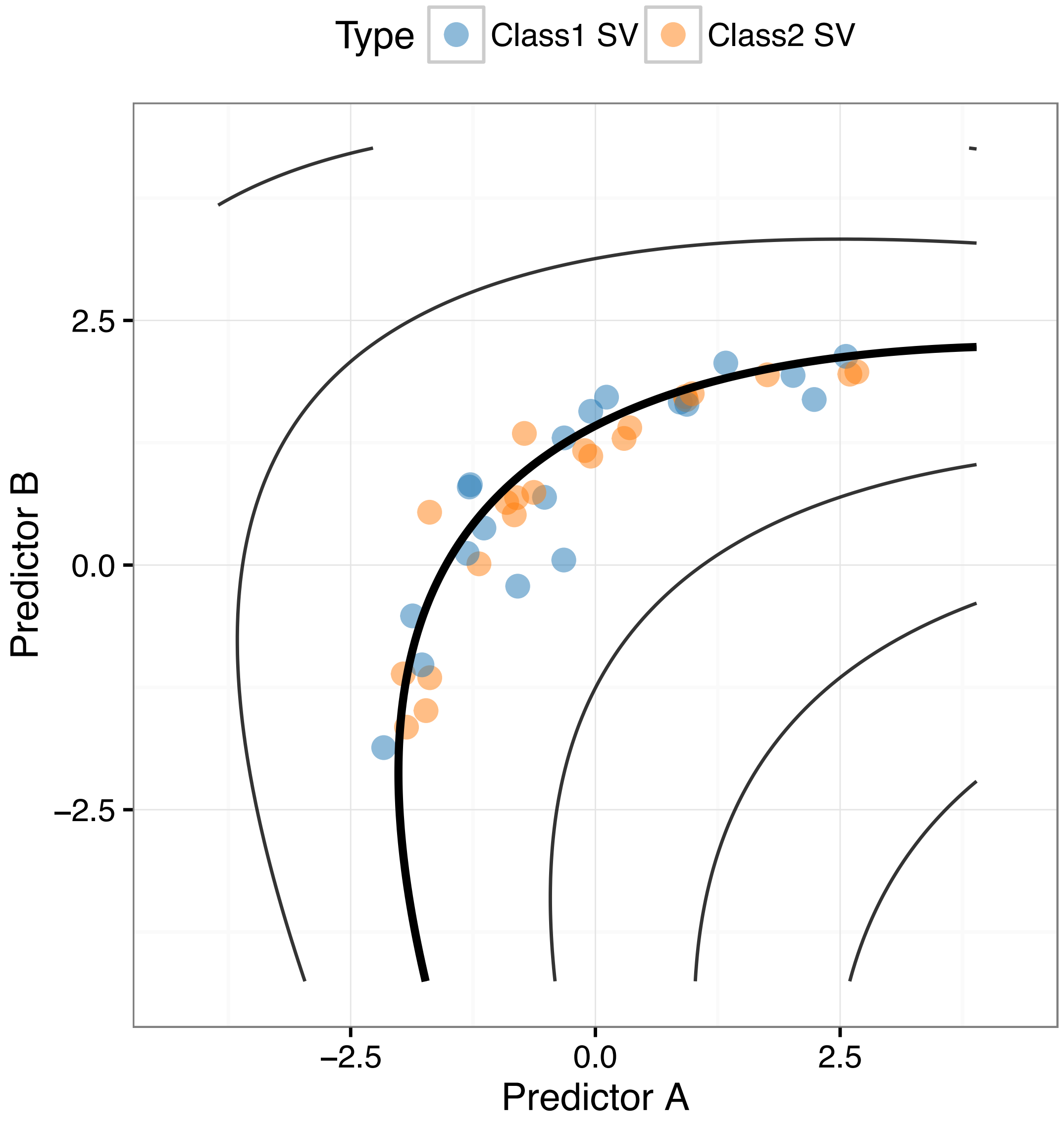
Polynomial SVM Fit



Decision Values ($\neq$ Probabilities)



Support Vectors Only

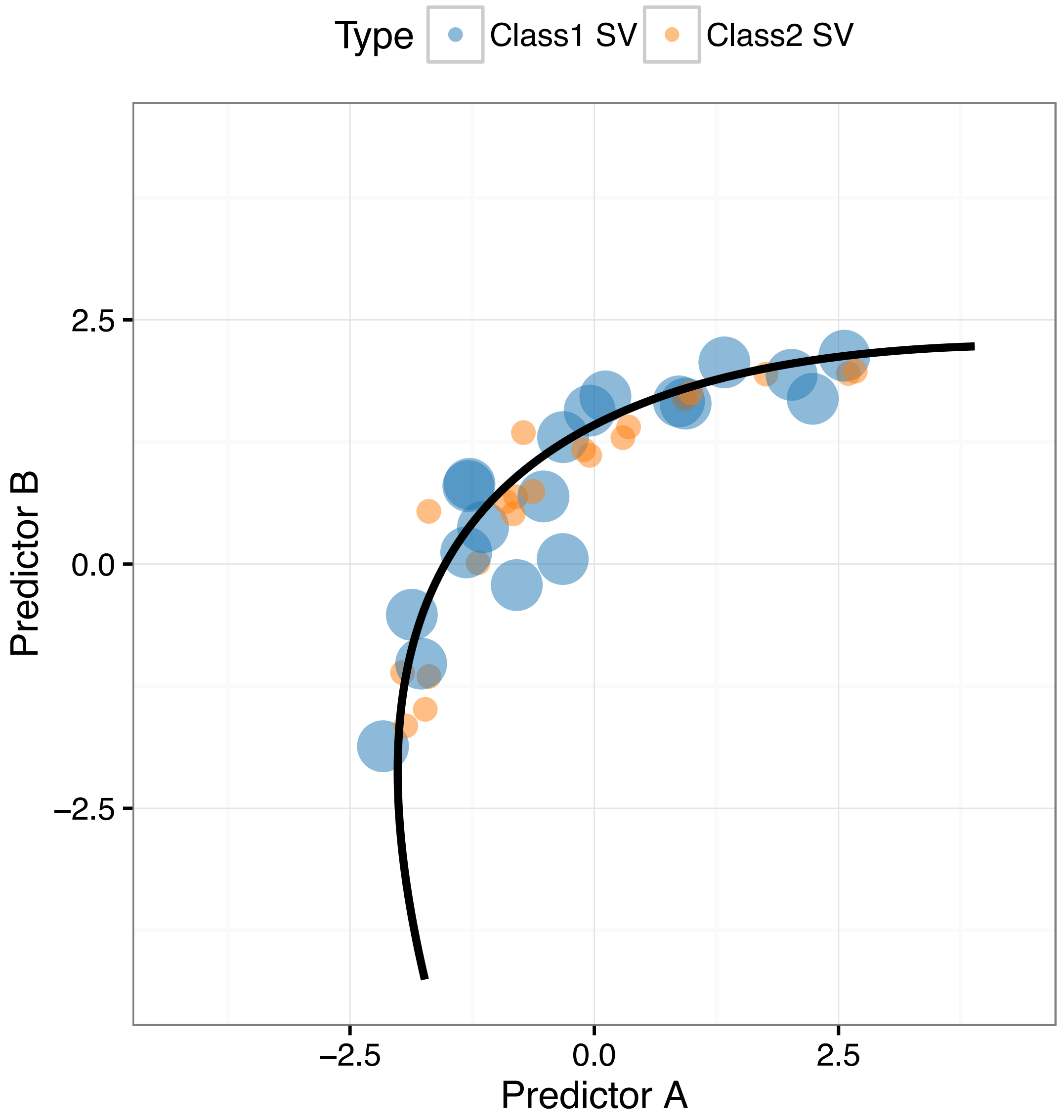


SVMs With Differential Class Weights

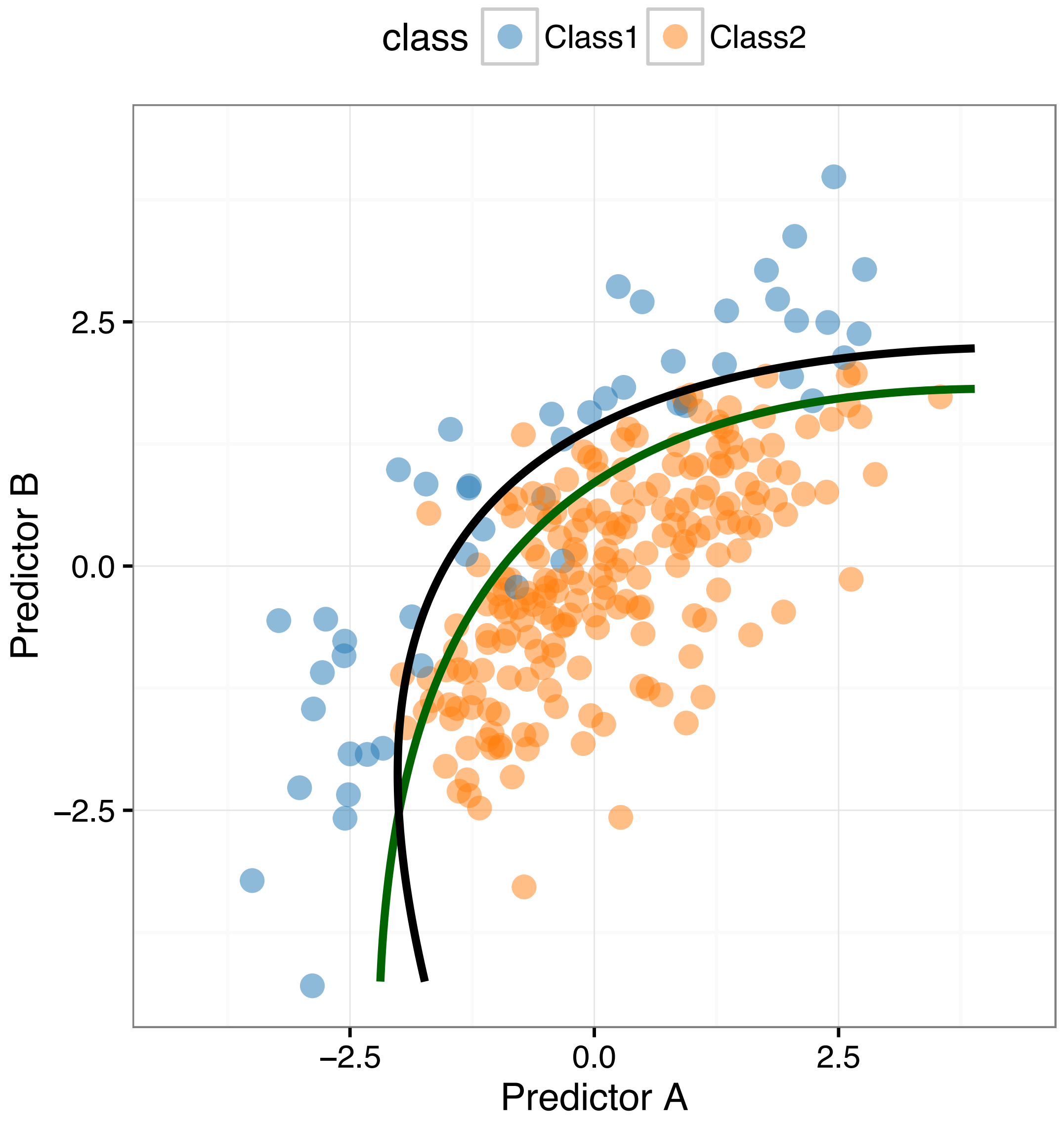
Since the SVM cost parameter affects each class differently, minimizing the cost can often mean overfitting to the majority class.

When an imbalance exists, we might be able to improve the fit by *weighting* the cost values for the minority class(es) to more heavily to bias the model in their favor.

SVMs With Differential Class Weights



10–Fold Cost for Class 1 in Green



SVM and Weights

The kernlab package's `ksvm` function has an argument called `class.weights` that should be a maned vector of weights.

caret's `train` function has a method called `"svmRadialWeights"` that can be used to optimize the cost, RBF kernel parameter, and class weights for an SVM.

Backup Slides

CART and Costs and Probabilities

```
> cost_mat <-matrix(c(0, 1, 5, 0), ncol = 2)
> rownames(cost_mat) <- colnames(cost_mat) <- levels(okc_train$Class)
> rp_mod <- rpart(Class ˜ ., data = okc_train, parms = list(loss = cost_mat))
> pred_1 <- predict(rp_mod, okc_test, type = "class")
> pred_2 <- ifelse(predict(rp_mod, okc_test)[, "stem"] >= .5, "stem", "other")
> pred_2 <- factor(pred_2, levels = levels(pred_1))
>
> table(pred_1, pred_2)

       pred_2
pred_1  stem other
  stem     0  8119
  other    0  9129
```