Trees and Forests

W. Evan Johnson, Ph.D.
Professor, Division of Infectious Disease
Director, Center for Data Science
Rutgers University – New Jersey Medical School

2025-05-30

Motivating example: olive oil

To motivate this section, we will use a new dataset that includes the breakdown of the composition of olive oil into 8 fatty acids:

```
library(tidyverse)
library(dslabs)
data("olive")
olive <- select(olive, -area) #remove the `area` column--don't use it
names(olive)
```

```
## [1] "region"      "palmitic"    "palmitoleic" "stearic"     "oleic"
## [6] "linoleic"    "linolenic"   "arachidic"   "eicosenoic"
```

Motivating example: olive oil

Here is a quick look at the data:

```
olive %>% head() %>%
  knitr::kable() %>% kable_styling(font_size =7)
```

region	palmitic	palmitoleic	stearic	oleic	linoleic	linolenic	arachidic	eicosenoic
Southern Italy	10.75	0.75	2.26	78.23	6.72	0.36	0.60	0.29
Southern Italy	10.88	0.73	2.24	77.09	7.81	0.31	0.61	0.29
Southern Italy	9.11	0.54	2.46	81.13	5.49	0.31	0.63	0.29
Southern Italy	9.66	0.57	2.40	79.52	6.19	0.50	0.78	0.35
Southern Italy	10.51	0.67	2.59	77.71	6.72	0.50	0.80	0.46
Southern Italy	9.11	0.49	2.68	79.24	6.78	0.51	0.70	0.44

Motivating example: olive oil

We will try to predict the region using the fatty acid composition values as predictors.

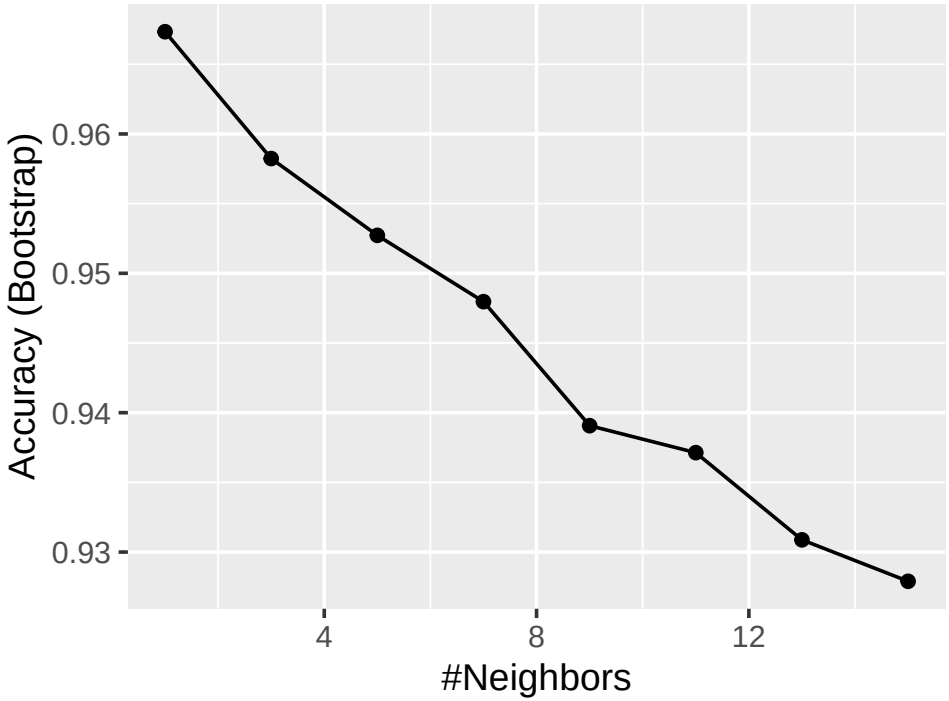
```
table(olive$region)
```

```
##
## Northern Italy       Sardinia Southern Italy
##            151             98            323
```

Motivating example: olive oil

Let's very quickly try to predict the region using kNN:

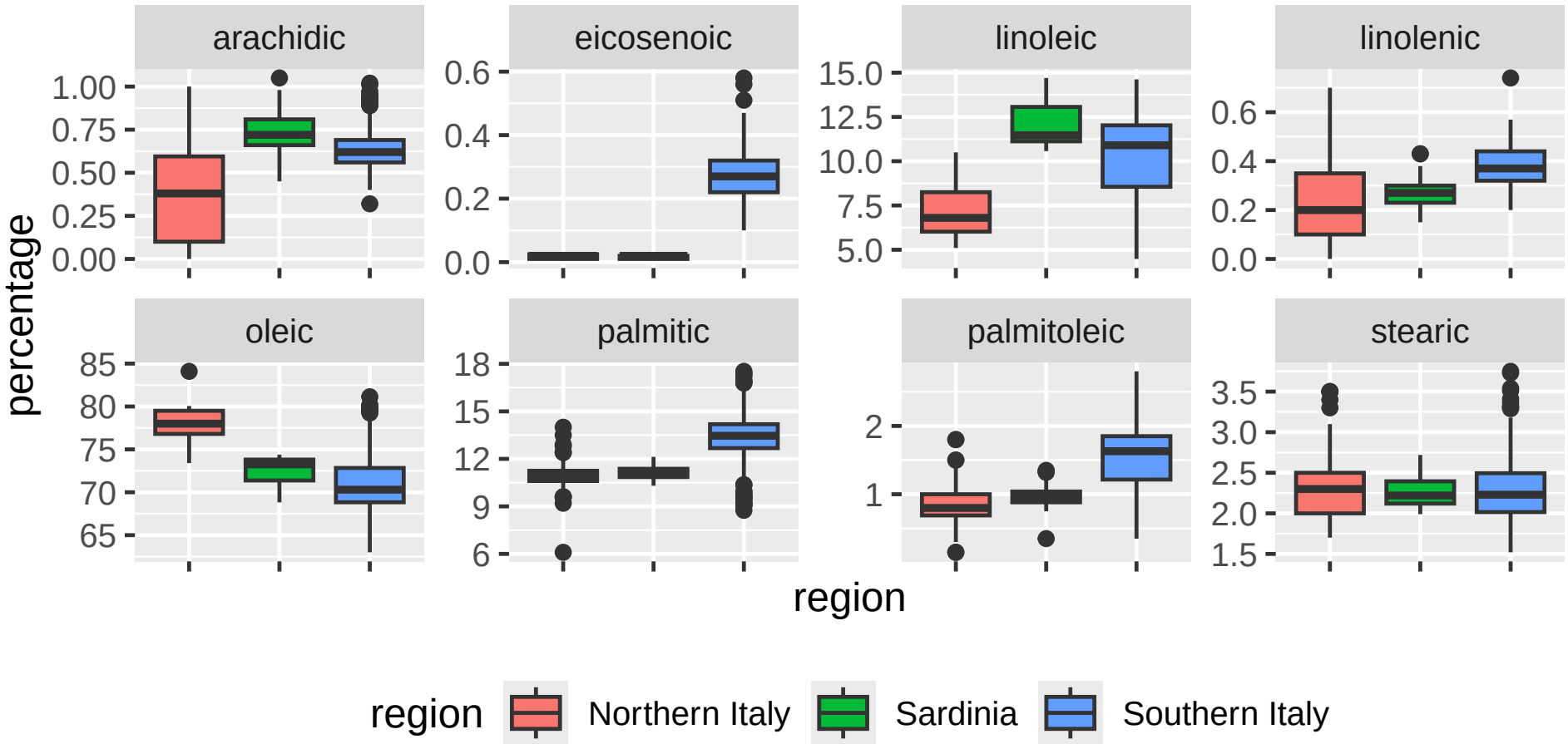
```
library(caret)
fit <- train(region ~ .,  method = "knn", data = olive,
             tuneGrid = data.frame(k = seq(1, 15, 2)))

ggplot(fit)
```



So using just one neighbor, we can predict relatively well.

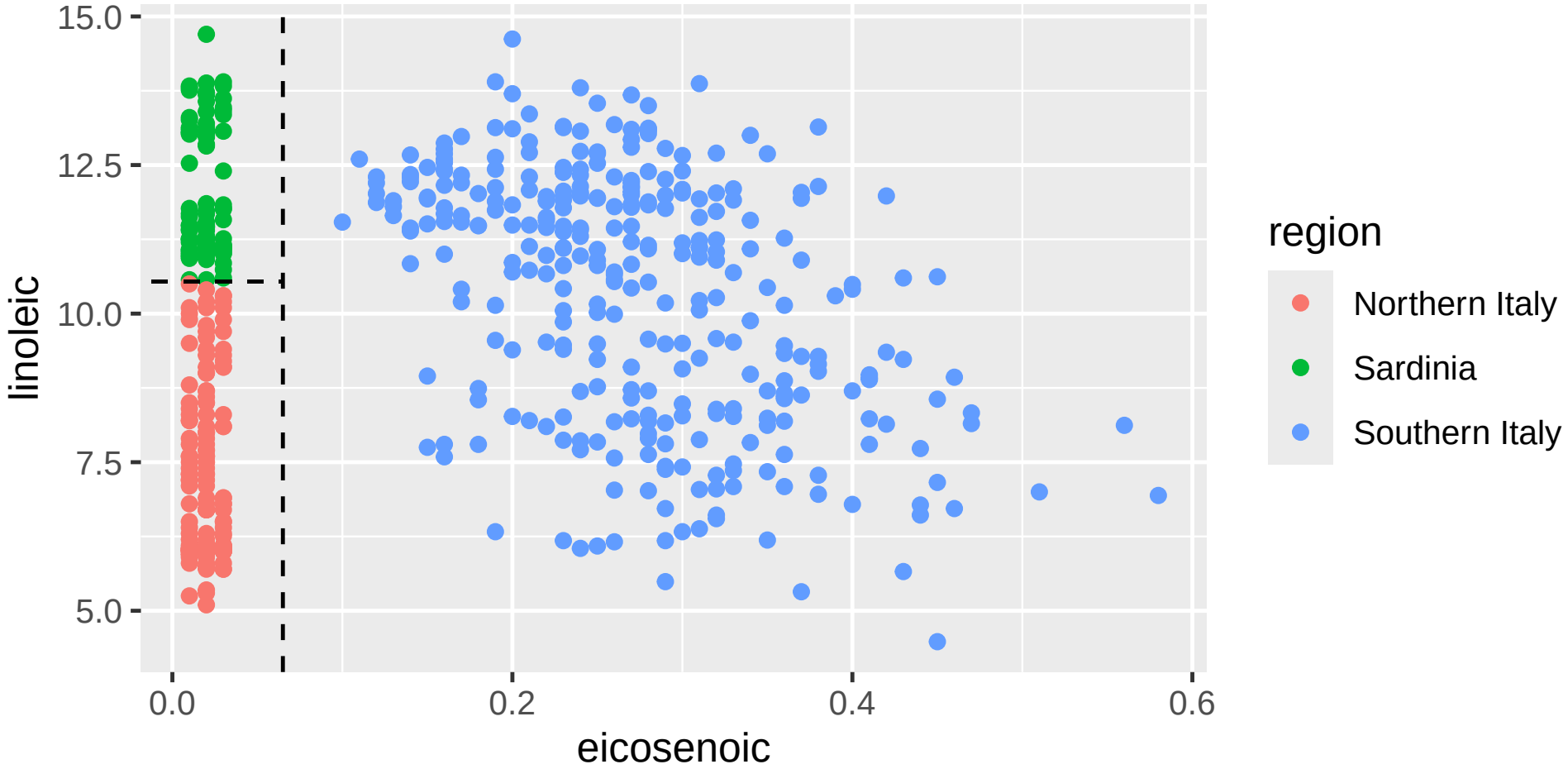
Motivating example: olive oil

However, a bit of data exploration reveals that we should be able to do even better: note that eicosenoic is only in Southern Italy and linoleic separates Northern Italy from Sardinia.



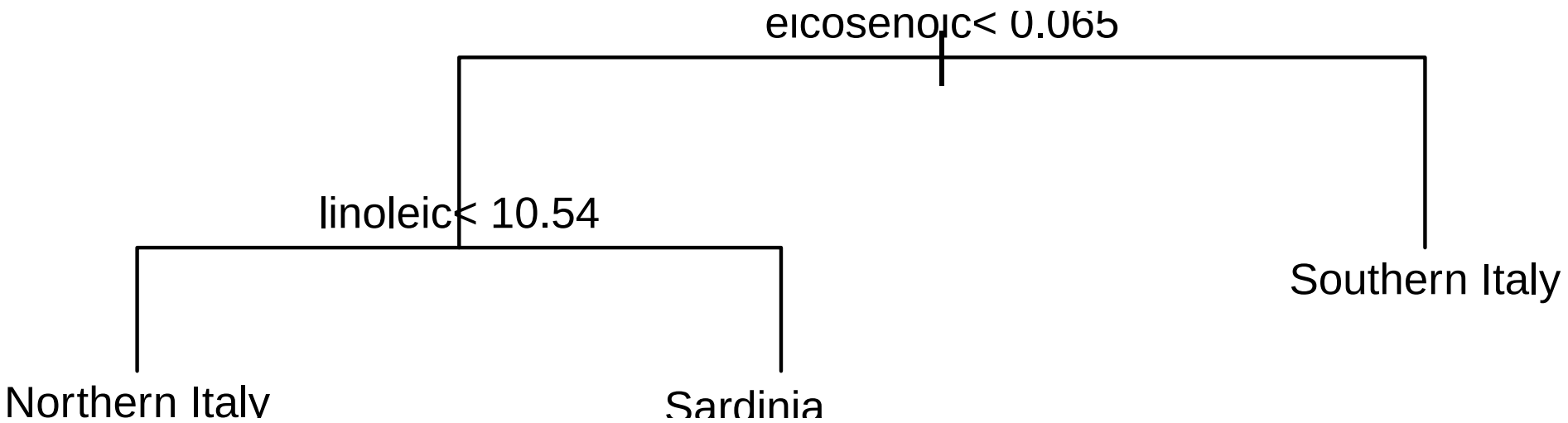
Motivating example: olive oil

This implies that we should be able to build an algorithm that predicts perfectly! Let's try plotting the values for eicosenoic and linoleic.



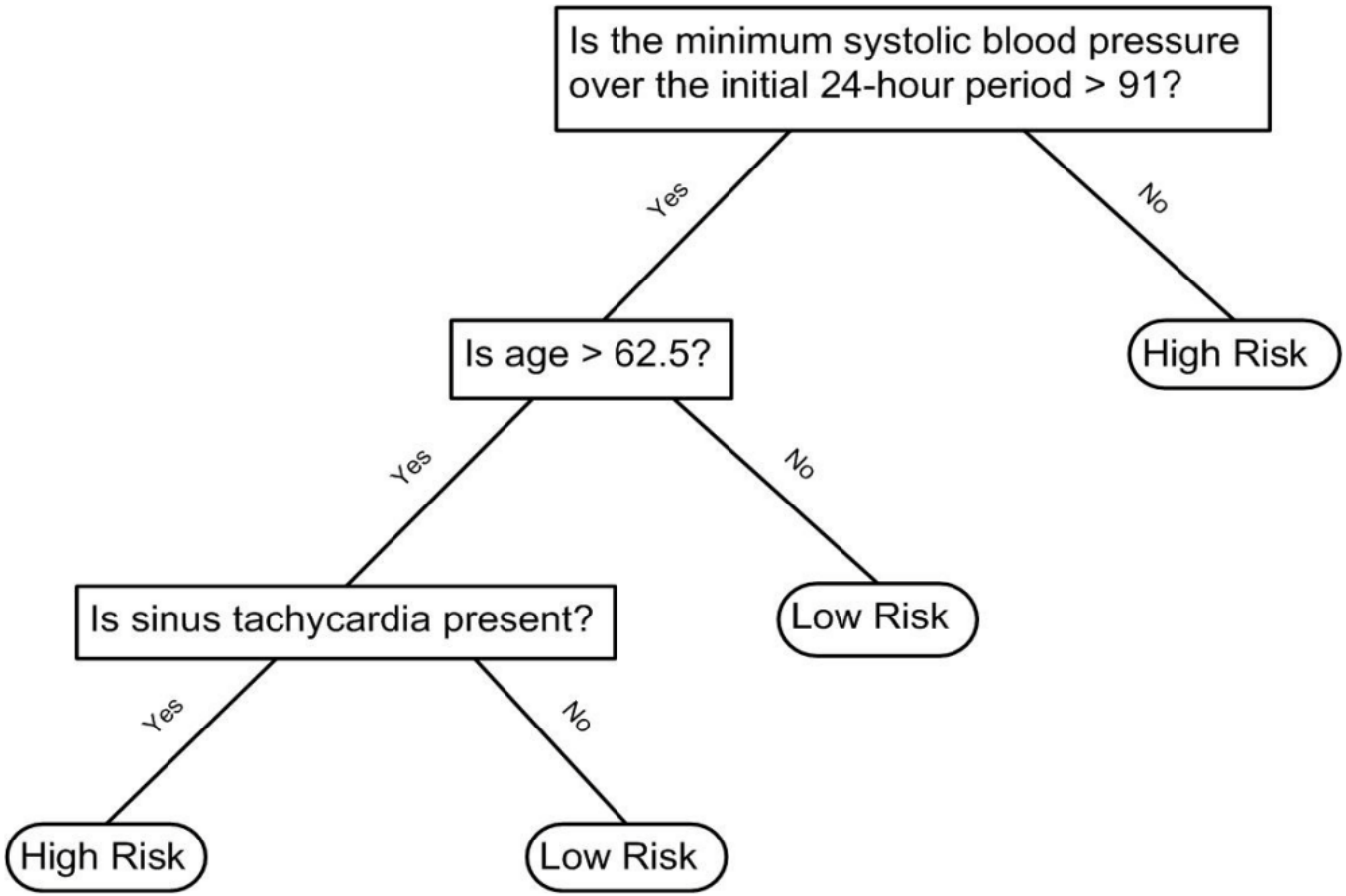
Motivating example: olive oil

Let's define a **decision rule**: If eicosenoic is larger than 0.065, predict Southern Italy. If not, then if linoleic is larger than 10.535, predict Sardinia, otherwise predict Northern Italy. We can draw this decision tree:



Decision Trees

Decision trees like this are often used in practice. For example, to evaluate a person's risk of poor outcome after a heart attack, doctors may use:



(Source: Walton 2010 Informal Logic, Vol. 30, No. 2, pp. 159-184.)

Decision Trees

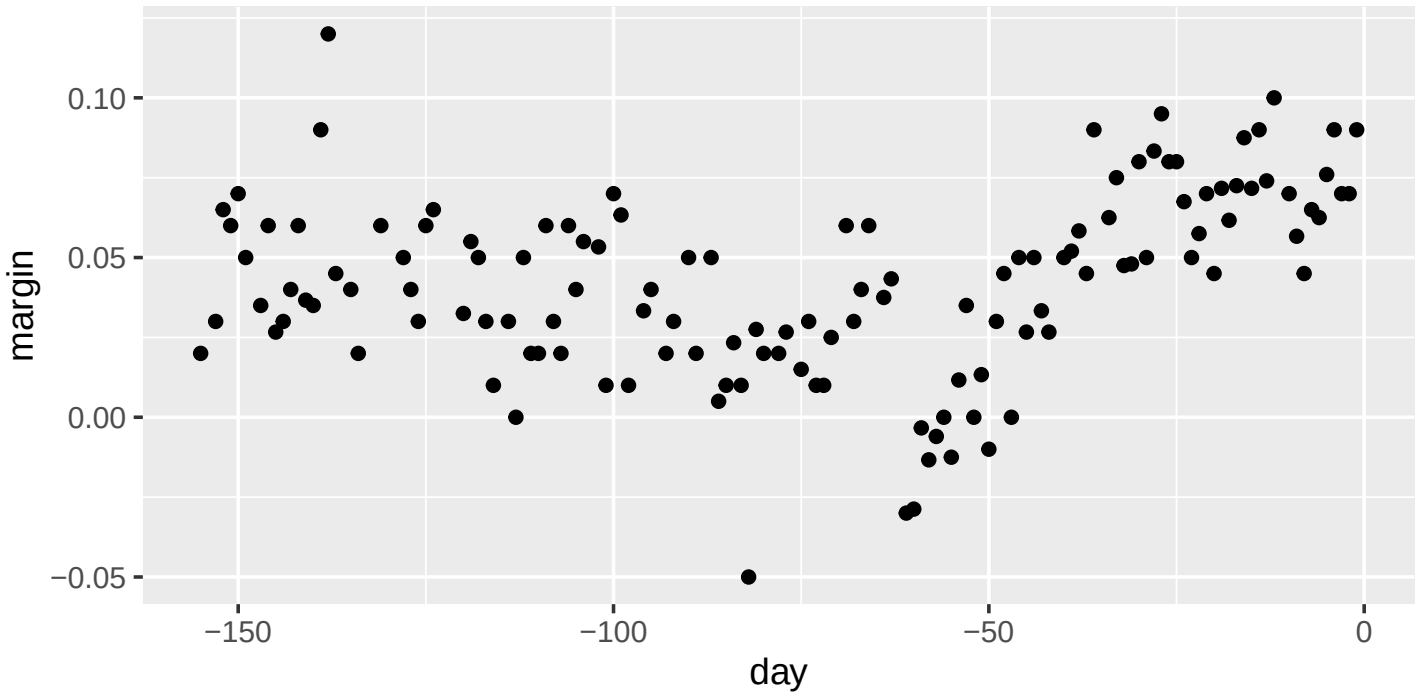
A tree is a flow chart of yes or no questions. We define an algorithm that creates trees with predictions at the ends, referred to as **nodes**.

Regression and decision trees, or sometimes called **Classification and Regression Trees (CART)**, predict an outcome Y by partitioning the predictors $\mathbf{X}$.

Regression trees

When the outcome is continuous, we call the method a **regression tree**. For example, we use polling data to estimate the conditional expectation $f(x) = \mathrm{E}(Y|X = x)$ with Y the poll margin and x the day.

```
data("polls_2008")
polls_2008 %>%
  ggplot() + geom_point(aes(day, margin))
```



Regression trees

The idea is to build a decision tree and, at the end of each **node**, obtain a predictor $\hat{y}$.

Mathematically, we are partitioning the predictor space into J non-overlapping regions, $R_1, R_2, \ldots, R_J$, and then for any predictor x that falls within region R_j, estimate $f(x)$ with the average of the training observations y_i for which the associated predictor x_i is also in R_j.

Regression trees

Regression trees create partitions recursively. We start the algorithm with one partition, the entire predictor space. In our simple first example, this space is the interval [-155, 1].

The first step will partition the space into two partitions. The second step will split one of these partitions into two and we will have three partitions, then four, then five, and so on.

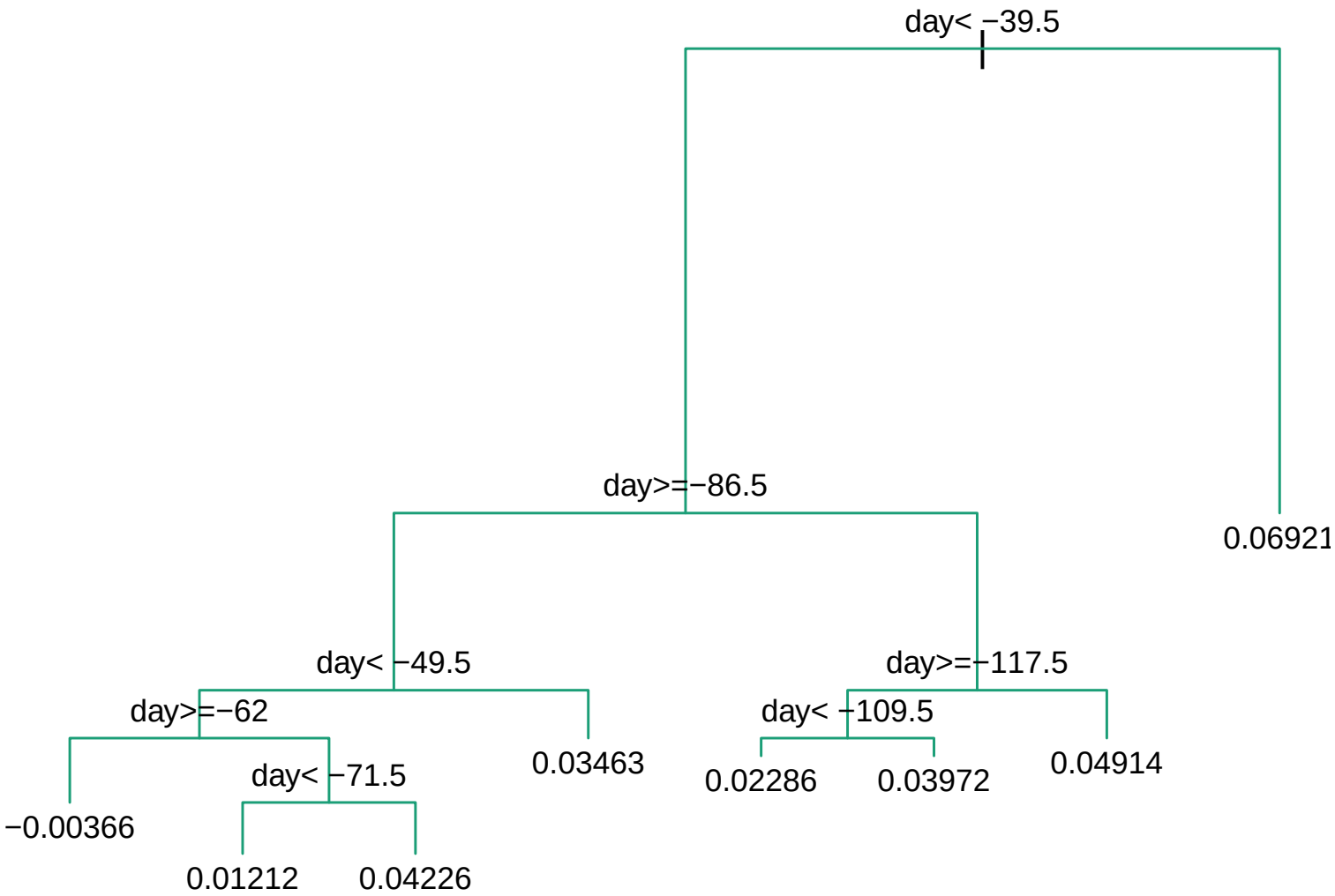
Regression trees

Let's take a look at what this algorithm does on the 2008 presidential election poll data. We will use the `rpart` function in the **rpart** package.

```
library(rpart)
fit <- rpart(margin ~ ., data = polls_2008)
```

Regression trees

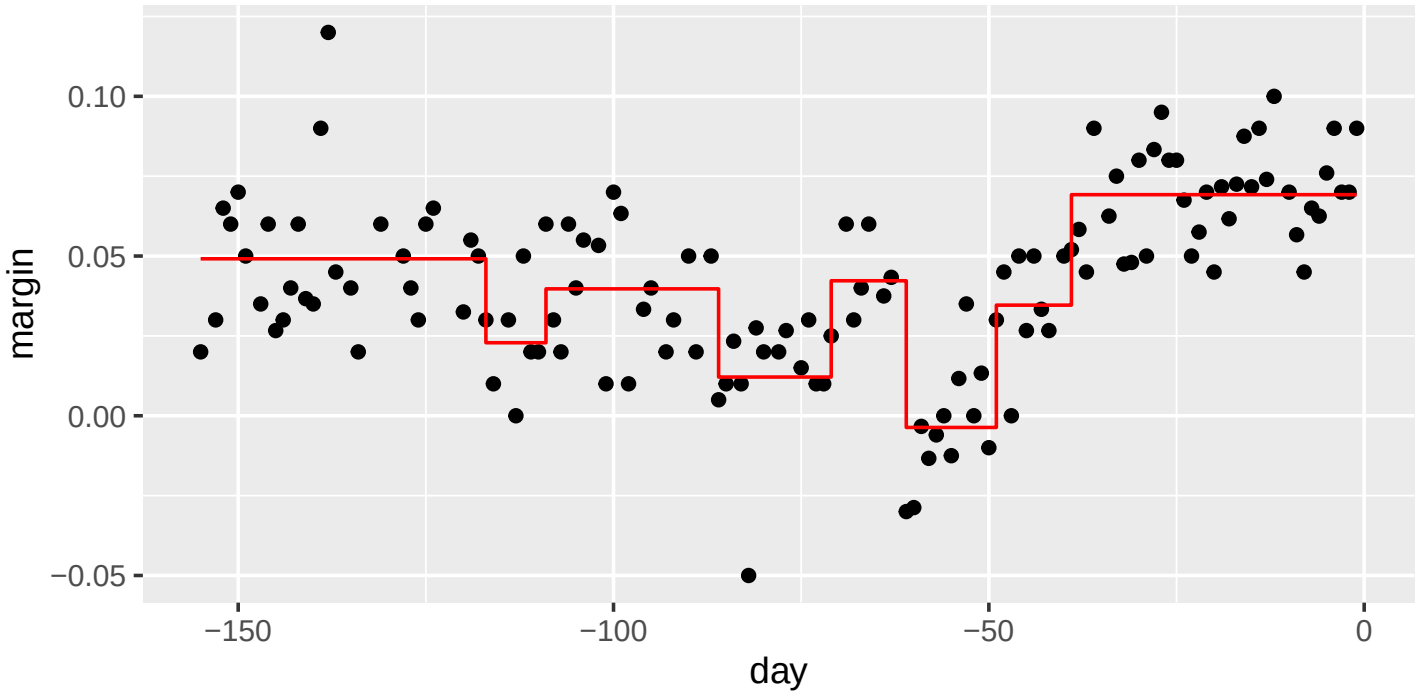
We can visually see where the splits were made:



Regression trees

The first split is made on day -39.5, then split at day -86.5. The two resulting new partitions are split on days -49.5 and -117.5, respectively, and so on. The final estimate $\hat{f}(x)$ looks like this:

```
polls_2008 %>%
  mutate(y_hat = predict(fit)) %>% ggplot() +
  geom_point(aes(day, margin)) + geom_step(aes(day, y_hat), col="red")
```



Regression trees

Note that the algorithm stopped partitioning at 8. Every time we split and define two new partitions, our training set RSS decreases because our model has more flexibility to adapt to the training data.

In fact, if you split until every point is its own partition, then RSS goes all the way down to 0.

To avoid this, the algorithm sets a minimum for how much the RSS must improve for another partition to be added. This parameter is referred to as the **complexity parameter (cp)**.

Regression trees

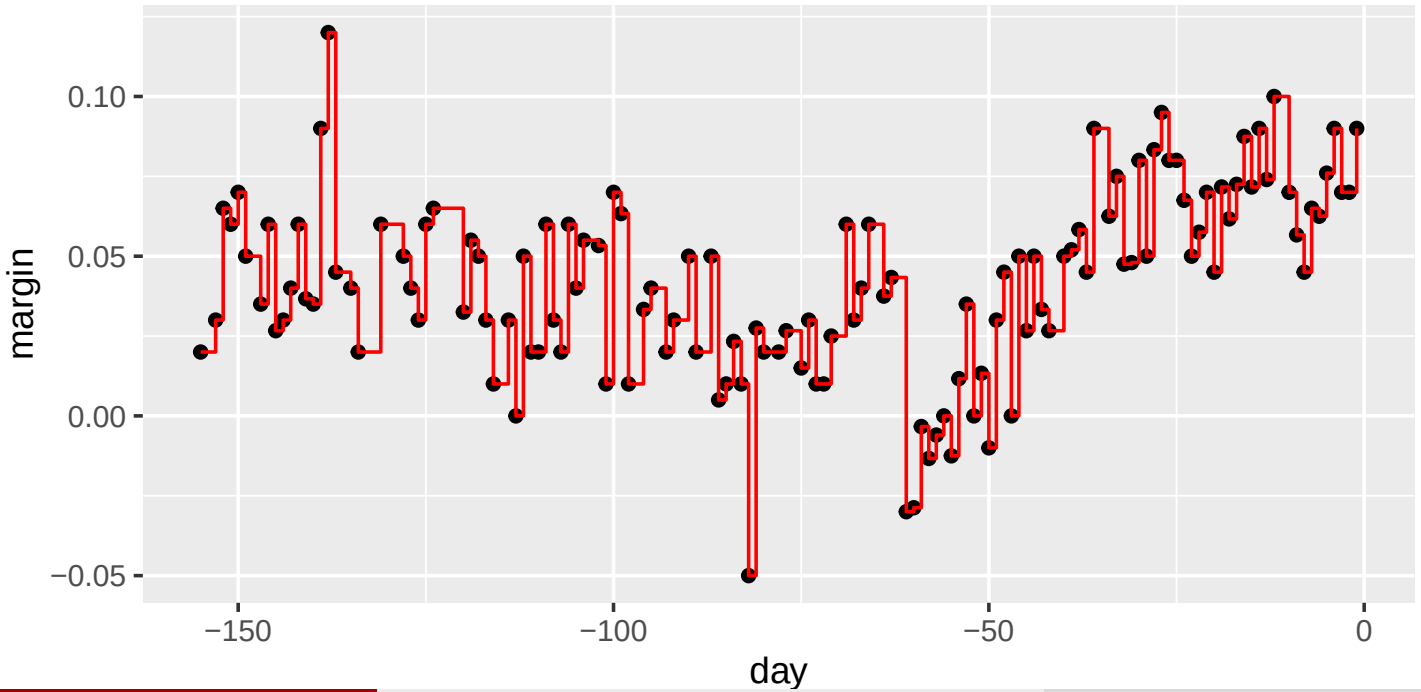
However, cp is not the only parameter — another common parameter is the minimum number of observations required in order to split a partition. The argument used in the `rpart` function is `minsplit` and the default is 20.

The `rpart` implementation of regression trees also permits users to determine a minimum number of observations in each node. The argument is `minbucket` and defaults to `round(minsplit/3)`.

Regression trees

If we set `cp = 0` and `minsplit = 2`, then our prediction is as flexible as possible and our predictor is our original data:

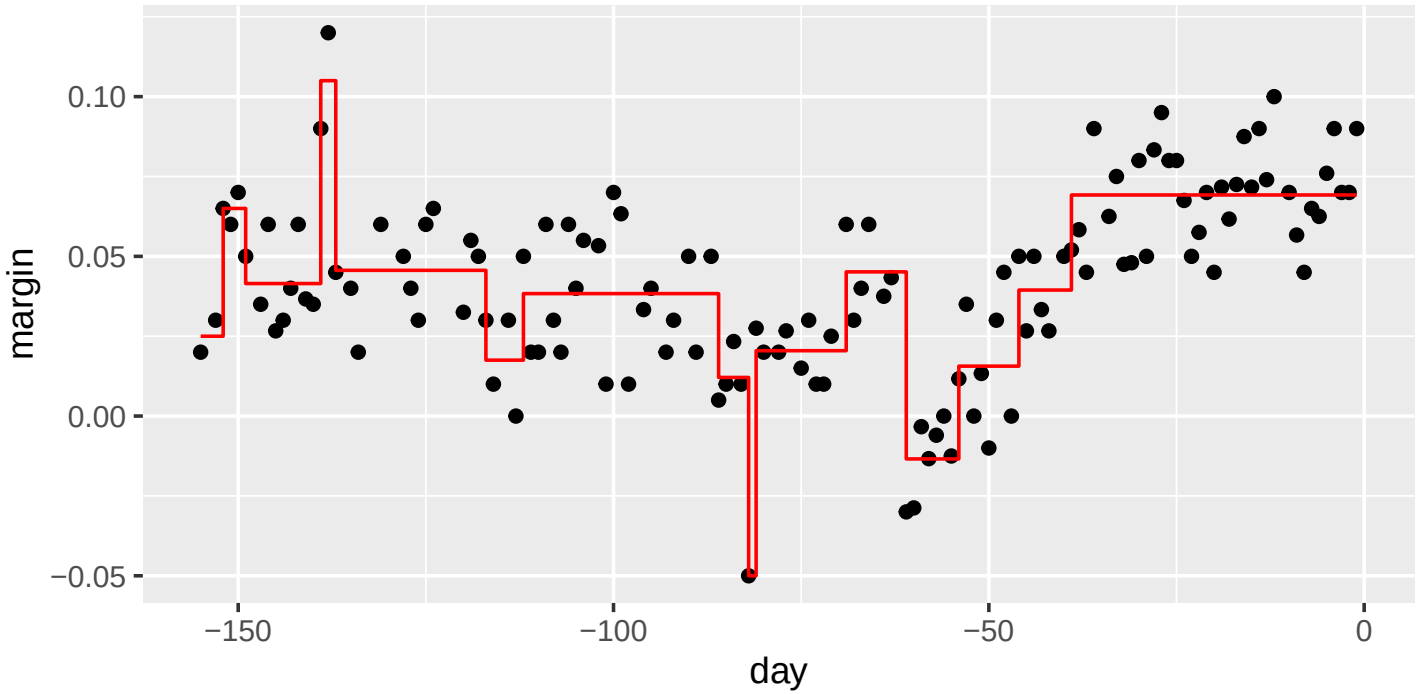
```
fit <- rpart(margin ~ ., data = polls_2008,
             control = rpart.control(cp = 0, minsplit = 2))
polls_2008 %>%
  mutate(y_hat = predict(fit)) %>% ggplot() +
  geom_point(aes(day, margin)) +  geom_step(aes(day, y_hat), col="red")
```



Regression trees

If we already have a tree and want to apply a higher cp value, we can **prune** the tree with the prune function:

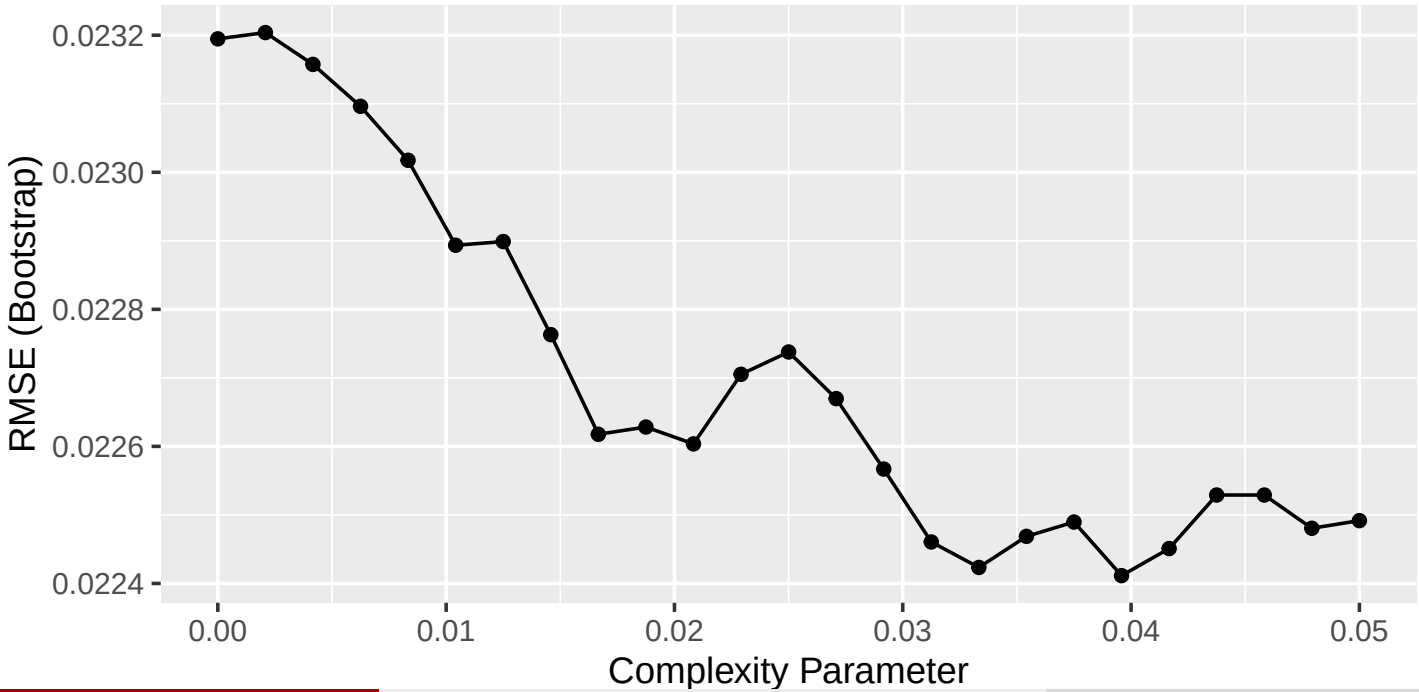
```
pruned_fit <- prune(fit, cp = 0.01)
polls_2008 %>% mutate(y_hat = predict(pruned_fit)) %>% ggplot() +
  geom_point(aes(day, margin)) + geom_step(aes(day, y_hat), col="red")
```



Regression trees

But how do we pick these parameters? We can use cross validation! Here is an example of using cross validation to choose cp:

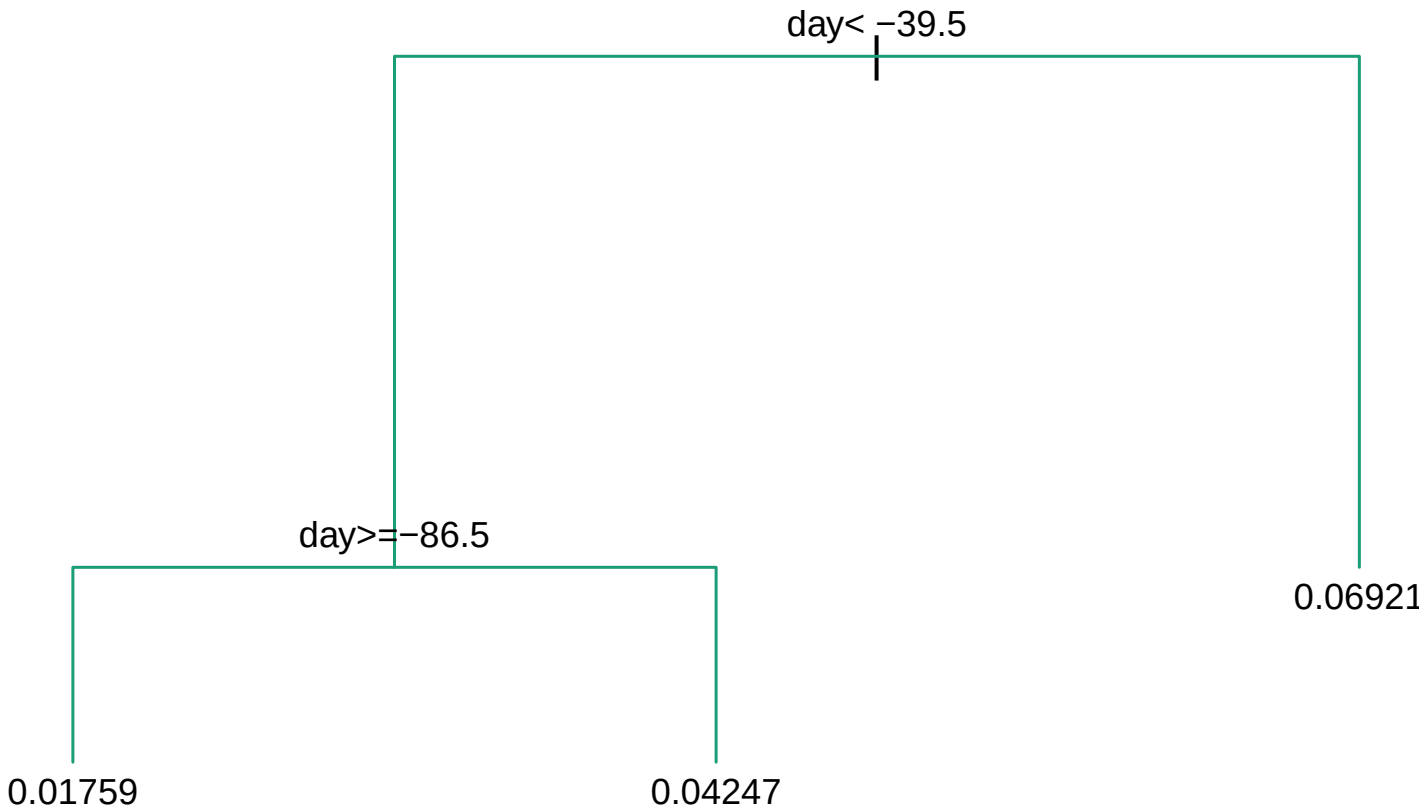
```
library(caret)
train_rpart <- train(margin ~ ., method = "rpart",
                     tuneGrid = data.frame(cp = seq(0, 0.05, len = 25)),
                     data = polls_2008)
ggplot(train_rpart)
```



Regression trees

To see the resulting tree, we access the `finalModel` and plot it:

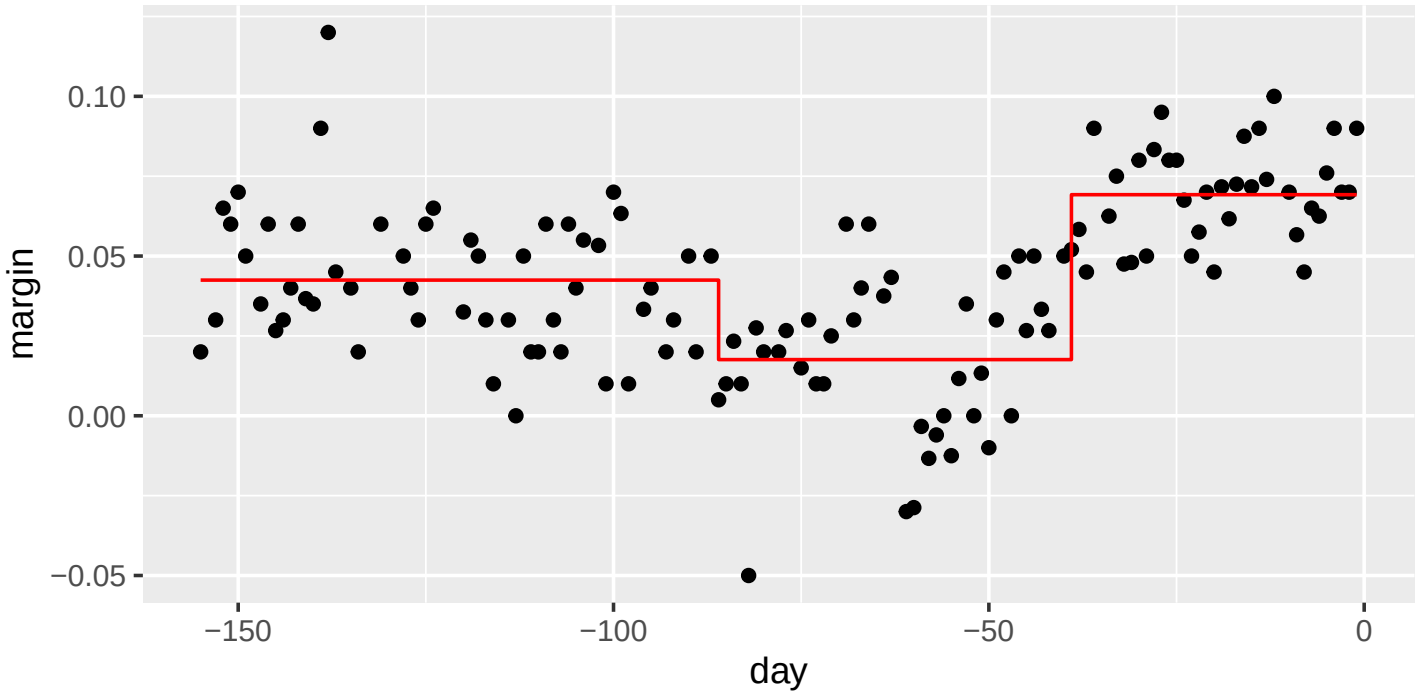
```
plot(train_rpart$finalModel, margin = 0.1)
text(train_rpart$finalModel, cex = 0.75)
```



Regression trees

And because we only have one predictor, we can actually plot $\hat{f}(x)$:

```
polls_2008 %>%
  mutate(y_hat = predict(train_rpart)) %>% ggplot() +
  geom_point(aes(day, margin)) +
  geom_step(aes(day, y_hat), col="red")
```



Classification (decision) trees

Classification trees, or **decision trees**, are used in prediction problems where the *outcome is categorical*.

First, we form predictions by calculating which class is the most common among the training set observations within the partition, rather than taking the average in each partition (as we can't take the average of categories).

Second, we metrics such as the the **Gini Index** and **Entropy** to optimize our results.

Classification (decision) trees

To define the **Gini Index**, we define $\hat{p}_{j,k}$ as the proportion of observations in partition j that are of class k. The Gini Index is defined as

$$\text{Gini}(j) = \sum_{k=1}^{K} \hat{p}_{j,k}(1 - \hat{p}_{j,k}).$$

In a perfect scenario, the outcomes in each of our partitions are all of the same category since this will permit perfect accuracy. The *Gini Index* would be 0, and becomes larger the more we deviate from this scenario.

Classification (decision) trees

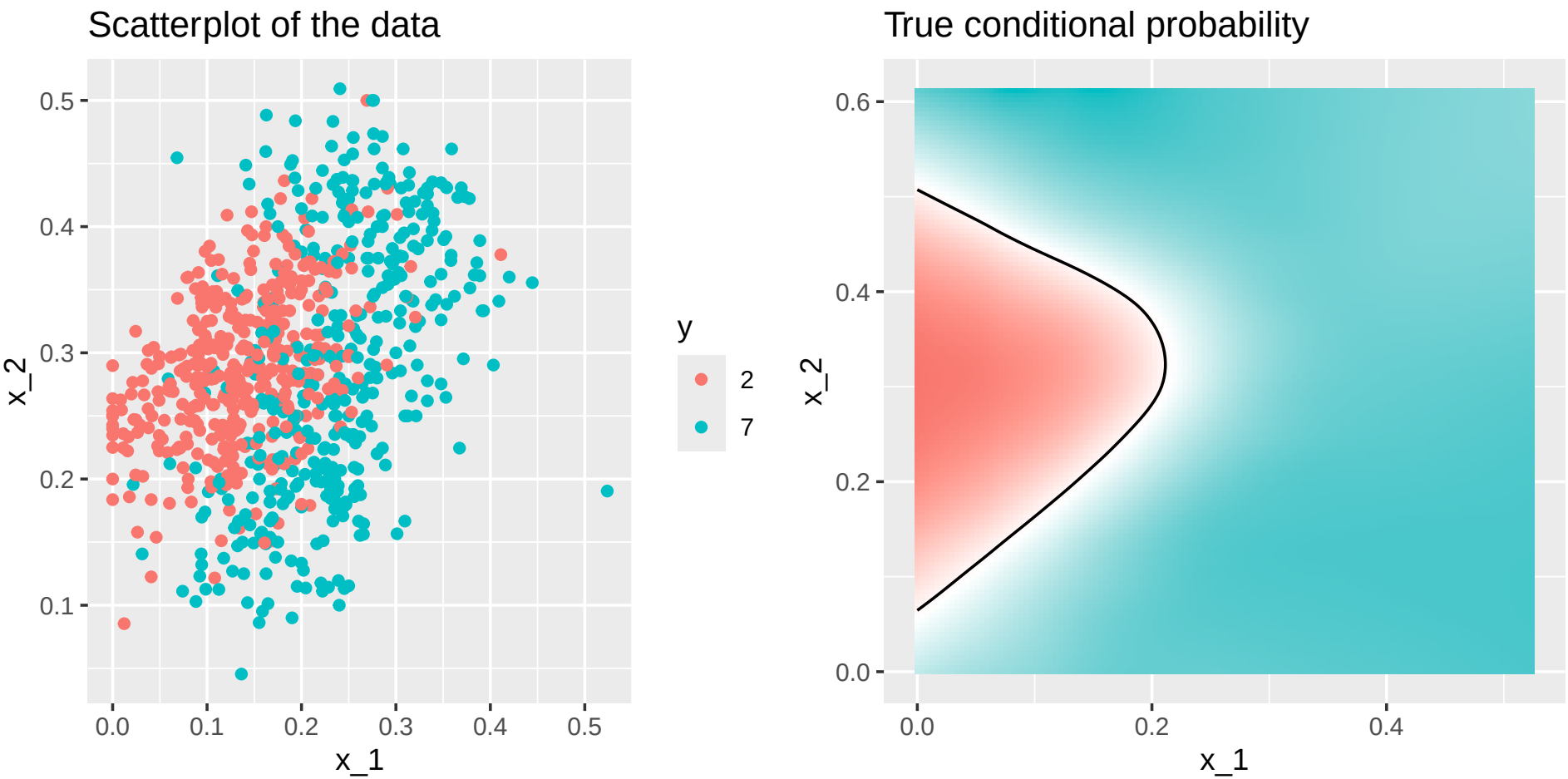
The **Entropy** is defined as

$$\text{entropy}(j) = -\sum_{k=1}^{K} \hat{p}_{j,k} \log(\hat{p}_{j,k}),$$

with $0 \times \log(0)$ defined as 0.

Classification (decision) trees

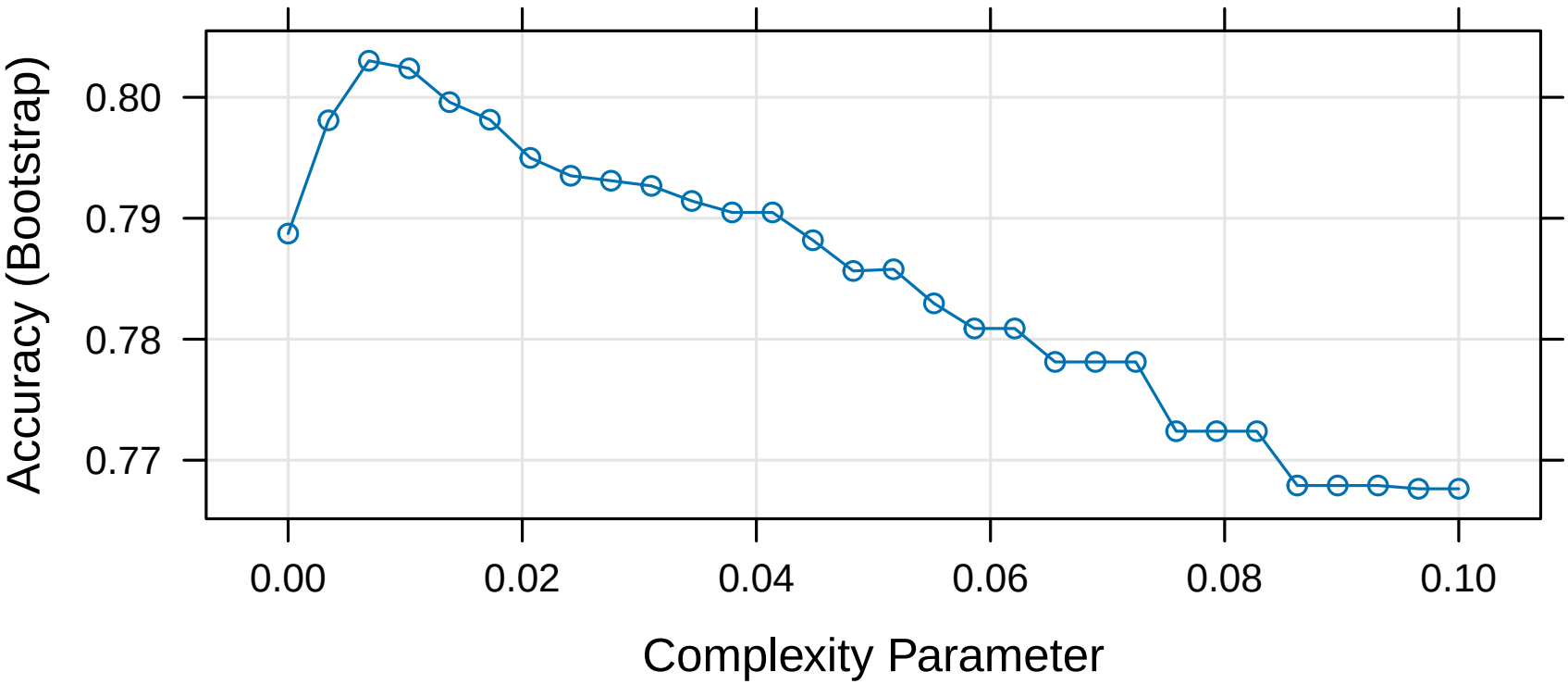
Lets look at how a classification tree performs on the following dataset:



Classification (decision) trees

Trying classification trees with different complexities:

```
train_rpart <- train(y ~ ., method = "rpart", data = mnist_27$train,
                     tuneGrid = data.frame(cp = seq(0.0, 0.1, len = 30)))
plot(train_rpart)
```



Classification (decision) trees

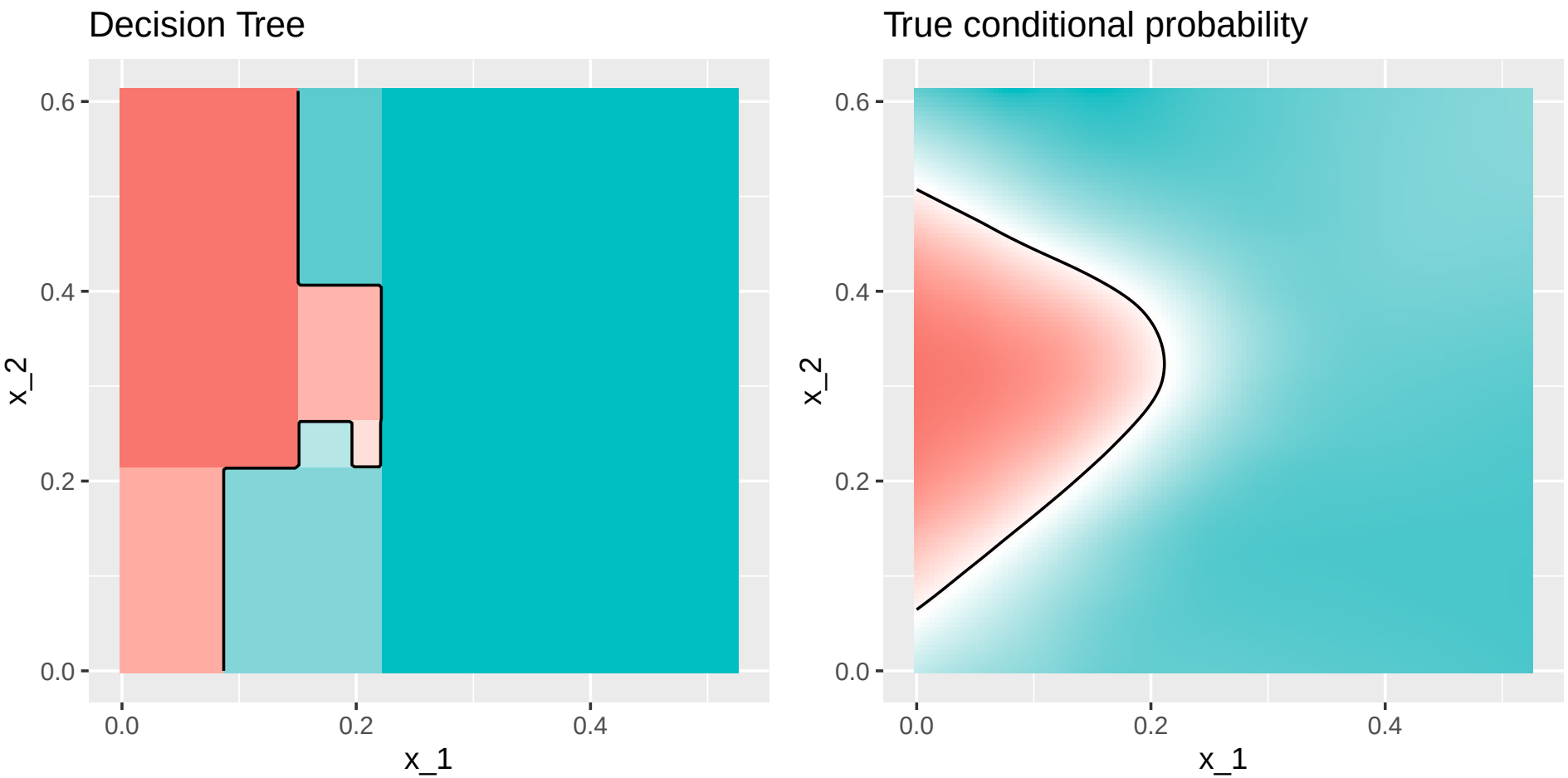
The accuracy achieved by this approach is better than what we got with regression, but is not as good as what we achieved with kernel methods:

```
y_hat <- predict(train_rpart, mnist_27$test)
confusionMatrix(y_hat, mnist_27$test$y)$overall["Accuracy"]
```

```
## Accuracy
##     0.81
```

Classification (decision) trees

The conditional probability shows us the limitations of classification trees:



Classification (decision) trees

Classification trees have certain advantages that make them very useful.

- They are highly interpretable, even more so than linear models.
- They are easy to visualize (if small enough).
- They can model human decision processes and don't require use of dummy predictors for categorical variables.

On the other hand:

- The approach via recursive partitioning can easily over-train and is therefore a bit harder to train than, for example, linear regression or kNN.
- In terms of accuracy, it is rarely the best performing method since it is not very flexible and is highly unstable to changes in training data.

Random forests, explained next, improve on several of these shortcomings.

Random forests

Random forests are a *very popular* and *very useful* machine learning approach that addresses the shortcomings of decision trees. The goal is to improve prediction performance and reduce instability by **averaging** multiple decision trees.

The first step is **bootstrap aggregation** or **bagging**. We will generate many regression or classification trees, and then form a final prediction based on the average of all these trees. These two features combined explain the name: the bootstrap makes the individual trees **randomly** different, and the combination of trees is the **forest**.

Random forests

The specific steps are as follows.

1. Build B decision trees, $T_1, T_2, \ldots, T_B$. using the training set. We later explain how we ensure they are different.

2. For every observation in the test set, form a prediction $\hat{y}_j$ using tree T_j.

3. For continuous outcomes, form a final prediction with the average $\hat{y} = \frac{1}{B}\sum_{j=1}^{B} \hat{y}_j$. For categorical data classification, predict $\hat{y}$ with majority vote (most frequent class among $\hat{y}_1, \ldots, \hat{y}_T$).

Random forests

So how do we get different decision trees from a single training set? For this, we use randomness in two ways. Let N be the number of observations in the training set. To create T_j, $j = 1, \ldots, B$ we do the following:

1. Create a bootstrap training set by **sampling N observations** from the training set *with replacement*. This is the first way to induce randomness.

2. The second way random forests induce randomness is by **randomly selecting predictor variables**, **x**, to be included (or excluded) in the building of each tree. A different random subset is selected for each tree.

Random forests

We will demonstrate how this works by fitting a random forest to the 2008 polls data. We will use the **randomForest** package:

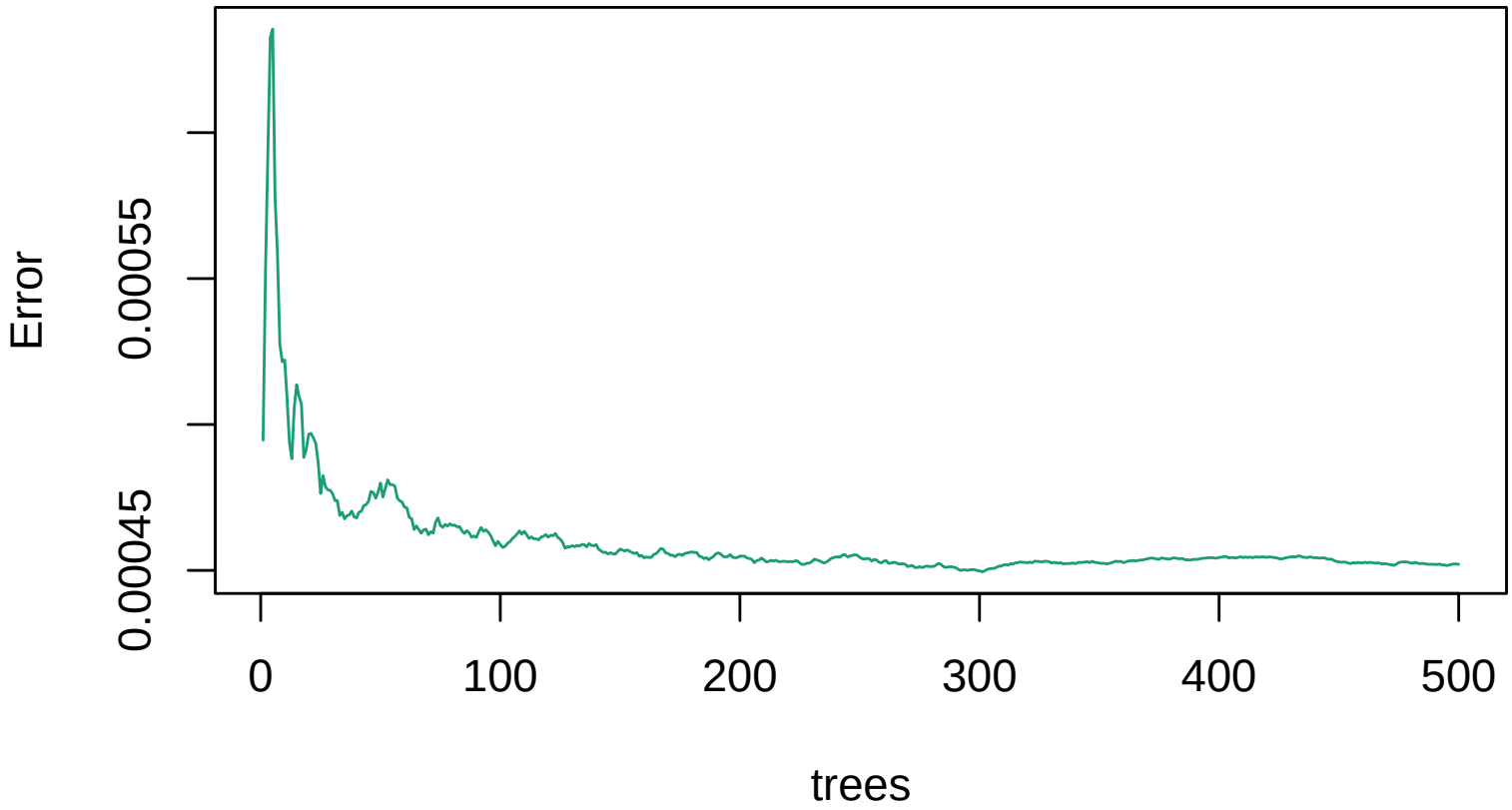
```
library(randomForest)
fit <- randomForest(margin~., data = polls_2008)
fit
```

```
##
## Call:
##  randomForest(formula = margin ~ ., data = polls_2008)
##                Type of random forest: regression
##                      Number of trees: 500
## No. of variables tried at each split: 1
##
##           Mean of squared residuals: 0.0004520772
##                     % Var explained: 45.84
```

Random forests

Note that if we apply the function `plot` to the resulting object, we see how the error rate of our algorithm changes as we add trees.

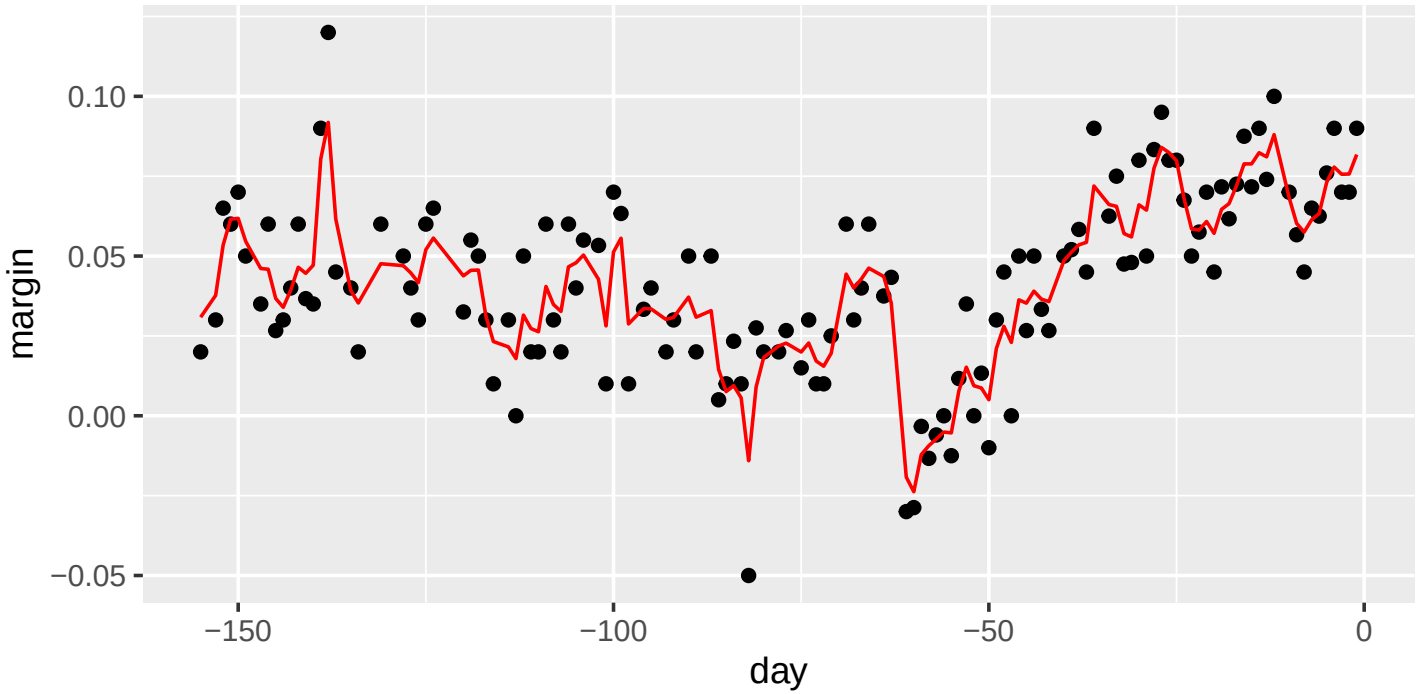
```
plot(fit,main='')
```



Random forests

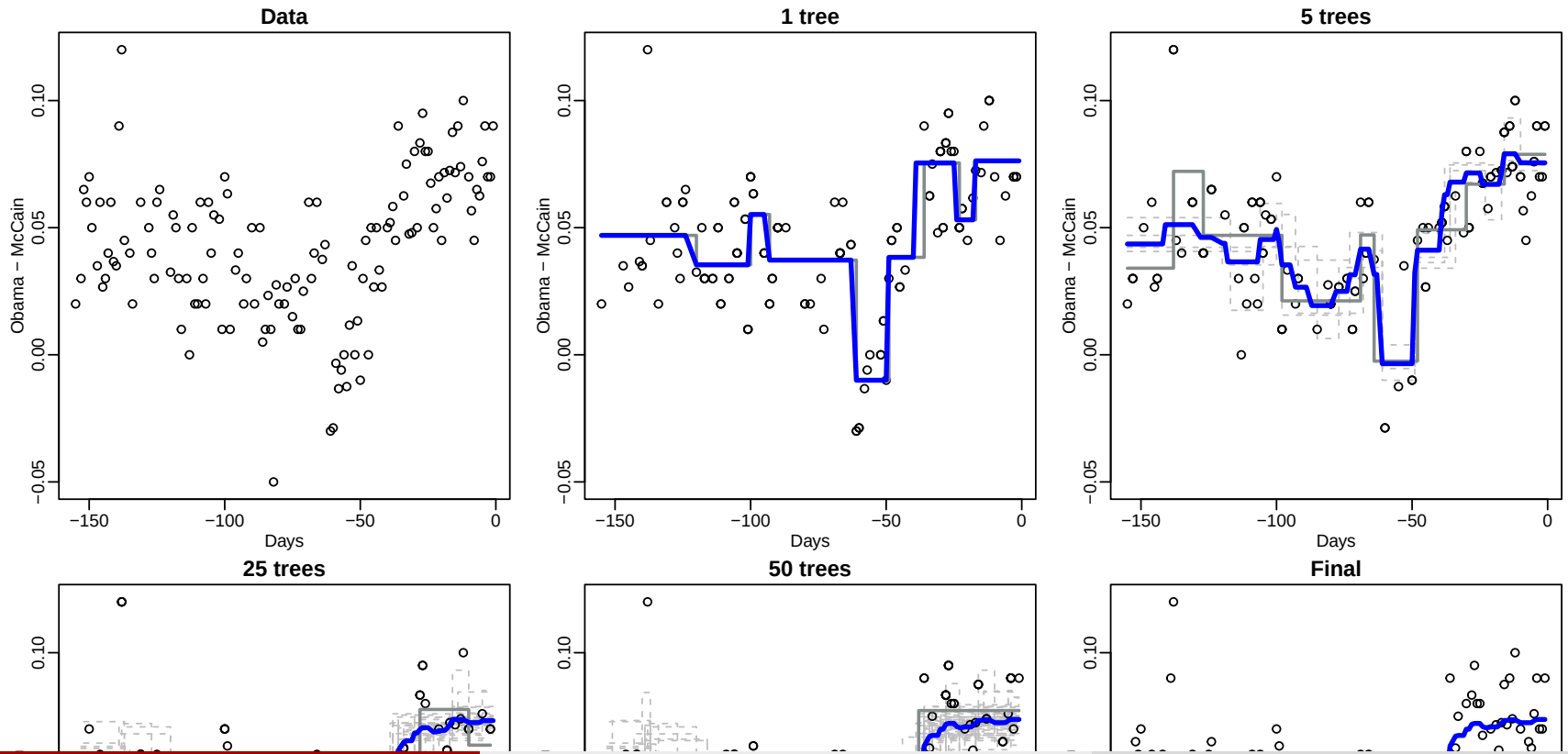
In this case, the accuracy improves as we add more trees until about 30 trees where accuracy stabilizes.The results for this random forest looks like this:

```
polls_2008 %>%
  mutate(y_hat = predict(fit, newdata = polls_2008)) %>% ggplot() +
  geom_point(aes(day, margin)) + geom_line(aes(day, y_hat), col="red")
```



Here we see each of the bootstrap samples for several values of b:

```
##
## Attaching package: 'rafalib'

## The following object is masked from 'package:lattice':
##
##     stripplot
```



Random forests

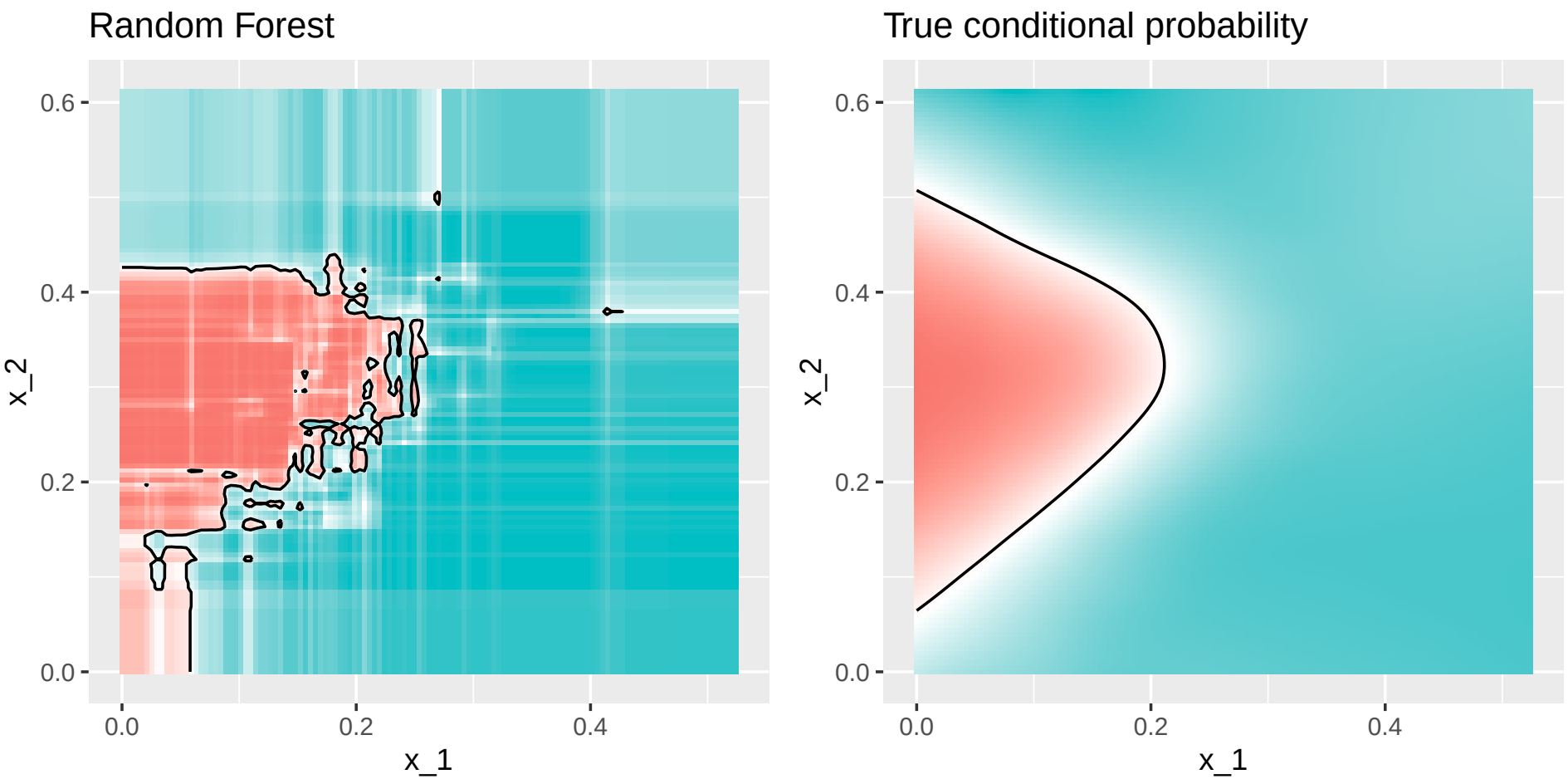
Here is the random forest fit for our digits example based on two predictors:

```
library(randomForest)
train_rf <- randomForest(y ~ ., data=mnist_27$train)

confusionMatrix(predict(train_rf, mnist_27$test),
                mnist_27$test$y)$overall["Accuracy"]
```

```
## Accuracy
##     0.82
```

Random forests

Here is what the conditional probabilities look like:



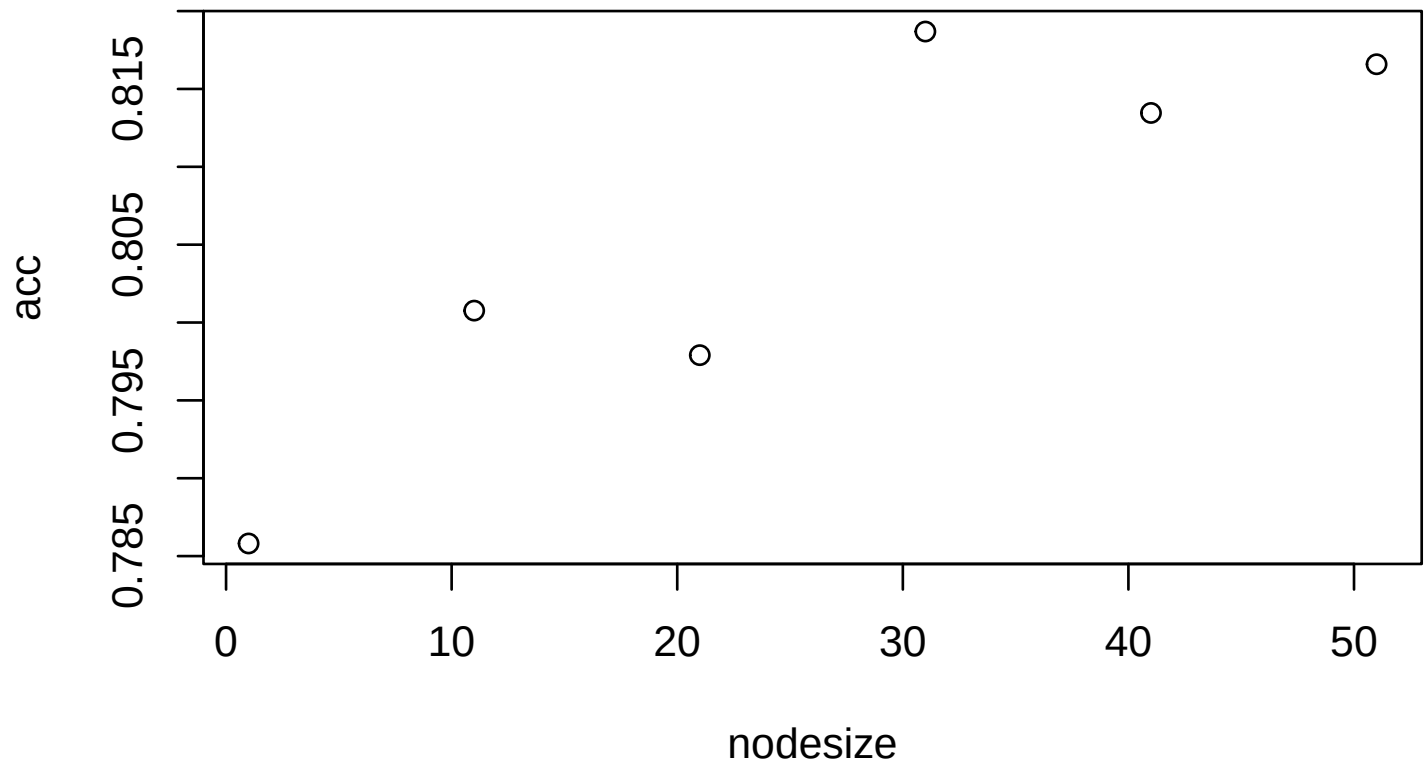
Random forests

Visualizing the estimate shows that, although we obtain high accuracy, it appears that there is room for improvement by making the estimate smoother. This could be achieved by changing the parameter that controls the minimum number of data points in the nodes of the tree. The larger this minimum, the smoother the final estimate will be. We can train the parameters of the random forest.

Random forests

We can use the **caret** package to optimize over the minimum node size:

```
nodesize <- seq(1, 51, 10)
acc <- sapply(nodesize, function(ns){
  train(y ~ ., method = "rf", data = mnist_27$train,
        tuneGrid = data.frame(mtry = 2),
        nodesize = ns)$results$Accuracy })
plot(nodesize, acc)
```



Random forests

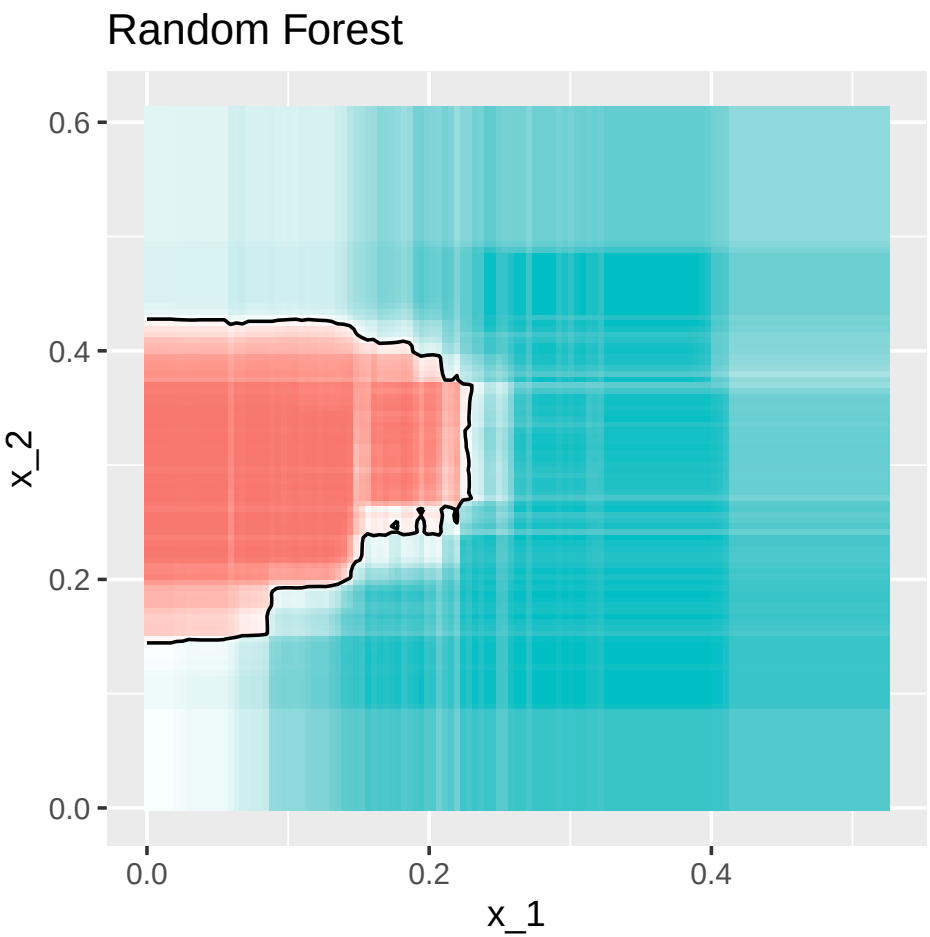
We can now fit the random forest with the optimized minimum node size to the entire training data and evaluate performance on the test data.

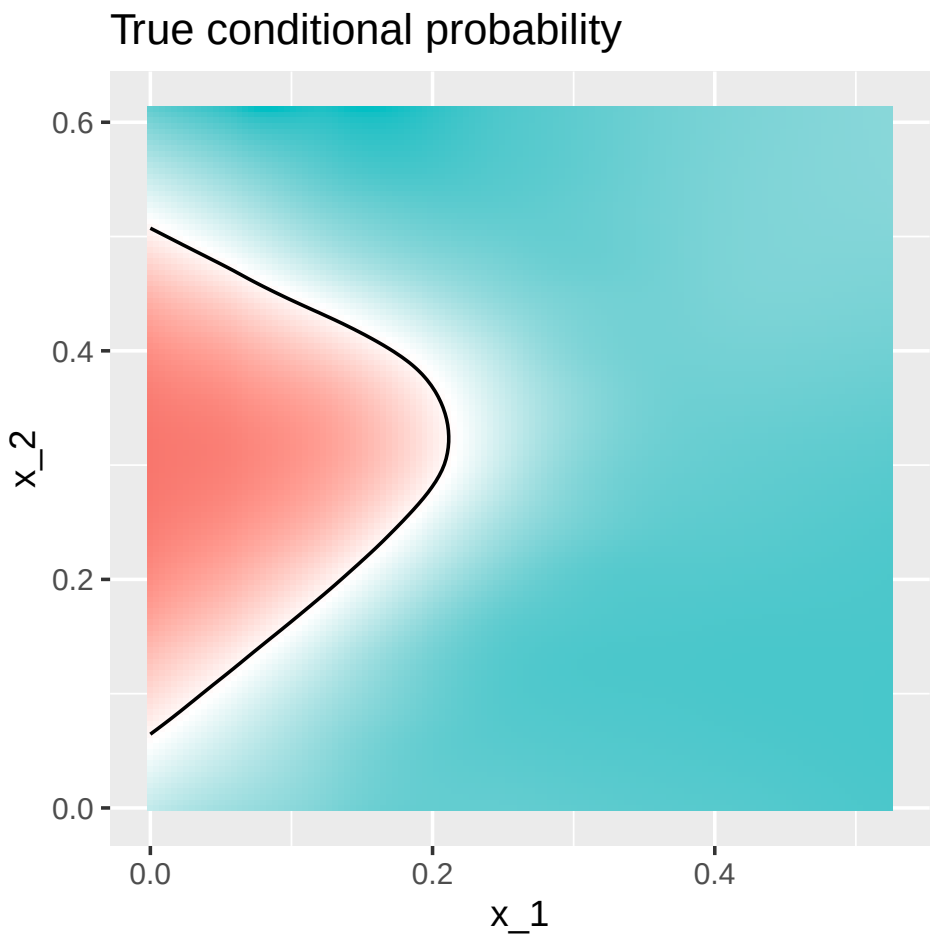
```
train_rf_2 <- randomForest(y ~ ., data=mnist_27$train,
                           nodesize = nodesize[which.max(acc)])

confusionMatrix(predict(train_rf_2, mnist_27$test),
                mnist_27$test$y)$overall["Accuracy"]
```

```
## Accuracy
##     0.84
```

Random forests

The selected model improves accuracy and provides a smoother estimate.





Random forests

Note that we can avoid writing our own code by using other random forest implementations as described in the **caret manual**.

Random forest often performs well in many examples, however, a disadvantage of random forests is that we lose interpretability.

An approach that helps with interpretability is to examine **variable importance**. To define *variable importance*, we count how often a predictor is used in the individual trees. You can learn more about it by clicking here.

The `caret package` includes the function `varImp` that extracts variable importance from any model in which the calculation is implemented.

Session Info

```
sessionInfo()
```

```
## R version 4.4.2 (2024-10-31)
## Platform: aarch64-apple-darwin20
## Running under: macOS Sequoia 15.5
##
## Matrix products: default
## BLAS:   /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRblas.0.dylib
## LAPACK: /Library/Frameworks/R.framework/Versions/4.4-arm64/Resources/lib/libRlapack.dylib;  LAPACK version 3
##
## locale:
## [1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8
##
## time zone: America/Denver
## tzcode source: internal
##
## attached base packages:
## [1] stats     graphics  grDevices utils     datasets  methods   base
##
## other attached packages:
##  [1] rafalib_1.0.4        randomForest_4.7-1.2 rpart_4.1.24
##  [4] dslabs_0.8.0         gridExtra_2.3        kableExtra_1.4.0
##  [7] lubridate_1.9.4      forcats_1.0.0        stringr_1.5.1
## [10] dplyr_1.1.4          purrr_1.0.4          readr_2.1.5
## [13] tidyr_1.3.1          tibble_3.2.1         tidyverse_2.0.0
## [16] caret_7.0-1          lattice_0.22-7       ggplot2_3.5.2
##
## loaded via a namespace (and not attached):
##  [1] tidyselect_1.2.1     viridisLite_0.4.2    timeDate_4041.110
##  [4] farver_2.1.2         fastmap_1.2.0        pROC_1.18.5
##  [7] digest_0.6.37        timechange_0.3.0     lifecycle_1.0.4
## [10] survival_3.8-3       magrittr_2.0.3       compiler_4.4.2
```