Chapter 31 Examples of algorithms

There are dozens of machine learning algorithms. Here we provide a few examples spanning rather different approaches. Throughout the chapter we will be using the two predictor digits data introduced in Section 27.8 to demonstrate how the algorithms work.

```
library(tidyverse)
library(dslabs)
library(caret)
data("mnist_27")
```

31.1 Linear regression

Linear regression can be considered a machine learning algorithm. In Section 27.8 we demonstrated how linear regression can be too rigid to be useful. This is generally true, but for some challenges it works rather well. It also serves as a baseline approach: if you can't beat it with a more complex approach, you probably want to stick to linear regression. To quickly make the connection between regression and machine learning, we will reformulate Galton's study with heights, a continuous outcome.

```
library(HistData)

set.seed(1983)
galton_heights <- GaltonFamilies %>%
  filter(gender == "male") %>%
  group_by(family) %>%
  sample_n(1) %>%
  ungroup() %>%
  select(father, childHeight) %>%
  rename(son = childHeight)
```

Suppose you are tasked with building a machine learning algorithm that predicts the son's height Y using the father's height X. Let's generate testing and training sets:

```
y <- galton_heights$son
test_index <- createDataPartition(y, times = 1, p = 0.5, list = FALSE)

train_set <- galton_heights %>% slice(-test_index)
test_set <- galton_heights %>% slice(test_index)
```

In this case, if we were just ignoring the father's height and guessing the son's height, we would guess the average height of sons.

```
m <- mean(train_set$son)
m
#> [1] 69.2
```

Our squared loss is:

```
mean((m - test_set$son)^2)
#> [1] 7.65
```

Can we do better? In the regression chapter, we learned that if the pair (X, Y) follow a bivariate normal distribution, the conditional expectation (what we want to estimate) is equivalent to the regression line:

$$f(x) = \mathrm{E}(Y \mid X = x) = \beta_0 + \beta_1 x$$

In Section 18.3 we introduced least squares as a method for estimating the slope β_0 and intercept β_1:

```
fit <- lm(son ~ father, data = train_set)
fit$coef
#> (Intercept)      father
#>      35.976       0.482
```

This gives us an estimate of the conditional expectation:

$$\hat{f}(x) = 52 + 0.25x$$

? Typos I think.

We can see that this does indeed provide an improvement over our guessing approach.

```
y_hat <- fit$coef[1] + fit$coef[2]*test_set$father
mean((y_hat - test_set$son)^2)
#> [1] 6.47
```

31.1.1 The `predict` function

The `predict` function is very useful for machine learning applications. This function takes a fitted object from functions such as `lm` or `glm` (we learn about `glm` soon) and a data frame with the new predictors for which to predict. So in our current example, we would use `predict` like this:

```
y_hat <- predict(fit, test_set)
```

Using `predict`, we can get the same results as we did previously:

```
y_hat <- predict(fit, test_set)
mean((y_hat - test_set$son)^2)
#> [1] 6.47
```

By the way, I prefer ROOT mean square error since it can be interpreted as a typical error size. sqrt(6.47) = 2.54.

`predict` does not always return objects of the same types; it depends on what type of object is sent to it. To learn about the specifics, you need to look at the help file specific for the type of fit object that is being used. The `predict` is actually a special type of function in R (called a *generic function*) that calls other functions depending on what kind of object it receives. So if `predict` receives an object coming out of the `lm` function, it will call `predict.lm`. If it receives an object coming out of `glm`, it calls `predict.glm`. These two functions are similar but different. You can learn more about the differences by reading the help files:

```
?predict.lm
?predict.glm
```

There are many other versions of `predict` and many machine learning algorithms have a `predict` function.

31.2 Exercises

1. Create a dataset using the following code.

```
n <- 100
Sigma <- 9*matrix(c(1.0, 0.5, 0.5, 1.0), 2, 2)
dat <- MASS::mvrnorm(n = 100, c(69, 69), Sigma) %>%
  data.frame() %>% setNames(c("x", "y"))
```

Use the **caret** package to partition into a test and training set of equal size. Train a linear model and report the RMSE. Repeat this exercise 100 times and make a histogram of the RMSEs and report the average and standard deviation. Hint: adapt the code shown earlier like this:

```
y <- dat$y
test_index <- createDataPartition(y, times = 1, p = 0.5, list = FALSE)
train_set <- dat %>% slice(-test_index)
test_set <- dat %>% slice(test_index)
fit <- lm(y ~ x, data = train_set)
y_hat <- fit$coef[1] + fit$coef[2]*test_set$x
mean((y_hat - test_set$y)^2)
```

and put it inside a call to `replicate`.

2. Now we will repeat the above but using larger datasets. Repeat exercise 1 but for datasets with `n <- c(100, 500, 1000, 5000, 10000)`. Save the average and standard deviation of RMSE from the 100 repetitions. Hint: use the `sapply` or `map` functions.

3. Describe what you observe with the RMSE as the size of the dataset becomes larger.

   a. On average, the RMSE does not change much as `n` gets larger, while the variability of RMSE does decrease.
   b. Because of the law of large numbers, the RMSE decreases: more data, more precise estimates.

c. `n = 10000` is not sufficiently large. To see a decrease in RMSE, we need to make it larger.
d. The RMSE is not a random variable.

4. Now repeat exercise 1, but this time make the correlation between `x` and `y` larger by changing `Sigma` like this:

```
n <- 100
Sigma <- 9*matrix(c(1, 0.95, 0.95, 1), 2, 2)
dat <- MASS::mvrnorm(n = 100, c(69, 69), Sigma) %>%
  data.frame() %>% setNames(c("x", "y"))
```

Repeat the exercise and note what happens to the RMSE now.

5. Which of the following best explains why the RMSE in exercise 4 is so much lower than exercise 1.

a. It is just luck. If we do it again, it will be larger.
b. The Central Limit Theorem tells us the RMSE is normal.
c. When we increase the correlation between `x` and `y`, `x` has more predictive power and thus provides a better estimate of `y`. This correlation has a much bigger effect on RMSE than `n`. Large `n` simply provide us more precise estimates of the linear model coefficients.
d. These are both examples of regression, so the RMSE has to be the same.

6. Create a dataset using the following code:

```
n <- 1000
Sigma <- matrix(c(1, 3/4, 3/4, 3/4, 1, 0, 3/4, 0, 1), 3, 3)
dat <- MASS::mvrnorm(n = 100, c(0, 0, 0), Sigma) %>%
  data.frame() %>% setNames(c("y", "x_1", "x_2"))
```

Note that `y` is correlated with both `x_1` and `x_2`, but the two predictors are independent of each other.

```
cor(dat)
```

Use the **caret** package to partition into a test and training set of equal size. Compare the RMSE when using just `x_1` , just `x_2` , and both `x_1` and `x_2` . Train a linear model and report the RMSE.

7. Repeat exercise 6 but now create an example in which `x_1` and `x_2` are highly correlated:

```
n <- 1000
Sigma <- matrix(c(1.0, 0.75, 0.75, 0.75, 1.0, 0.95, 0.75, 0.95, 1.0), 3, 3)
dat <- MASS::mvrnorm(n = 100, c(0, 0, 0), Sigma) %>%
  data.frame() %>% setNames(c("y", "x_1", "x_2"))
```

Use the **caret** package to partition into a test and training set of equal size. Compare the RMSE when using just `x_1` , just `x_2` , and both `x_1` and `x_2` Train a linear model and report the RMSE.

8. Compare the results in 6 and 7 and choose the statement you agree with:

a. Adding extra predictors can improve RMSE substantially, but not when they are highly correlated with another predictor.
b. Adding extra predictors improves predictions equally in both exercises.
c. Adding extra predictors results in over fitting.
d. Unless we include all predictors, we have no predicting power.

31.3 Logistic regression

The regression approach can be extended to categorical data. In this section we first illustrate how, for binary data, one can simply assign numeric values of 0 and 1 to the outcomes y, and apply regression as if the data were continuous. We will then point out a limitation with this approach and introduce *logistic regression* as a solution. Logistic regression is a specific case of a set of *generalized linear models*. To illustrate logistic regression, we will apply it to our previous predicting sex example:

If we define the outcome Y as 1 for females and 0 for males, and X as the height, we are interested in the conditional probability:

$$\Pr(Y = 1 \mid X = x)$$

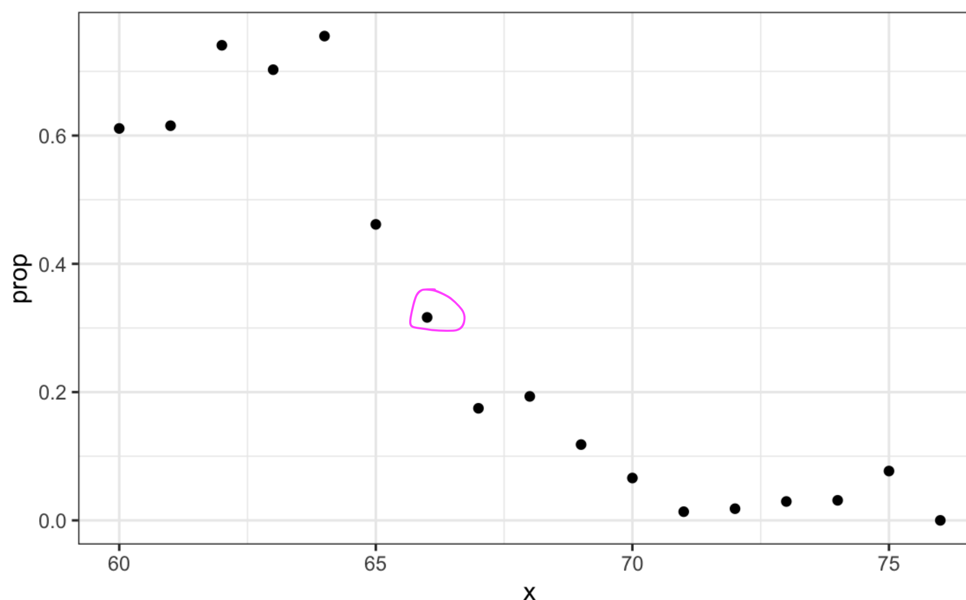
As an example, let's provide a prediction for a student that is 66 inches tall. What is the conditional probability of being female if you are 66 inches tall? In our dataset, we can estimate this by rounding to the nearest inch and computing:

```
train_set %>%
  filter(round(height)==66) %>%
  summarize(y_hat = mean(sex=="Female"))
#>   y_hat
#> 1 0.327
```

this is a crude estimate.

To construct a prediction algorithm, we want to estimate the proportion of the population that is female for any given height $X = x$, which we write as the conditional probability described above: $\Pr(Y = 1|X = x)$. Let's see what this looks like for several values of x (we will remove strata of x with few data points):

```
heights %>%
  mutate(x = round(height)) %>%
  group_by(x) %>%
  filter(n() >= 10) %>%
  summarize(prop = mean(sex == "Female")) %>%
  ggplot(aes(x, prop)) +
  geom_point()
#> `summarise()` ungrouping output (override with `.groups` argument)
```



Since the results from the plot above look close to linear, and it is the only approach we currently know, we will try regression. We assume that:

$$p(x) = \Pr(Y = 1 | X = x) = \beta_0 + \beta_1 x$$

Note: because $p_0(x) = 1 - p_1(x)$, we will only estimate $p_1(x)$ and drop the $_1$ index.

If we convert the factors to 0s and 1s, we can estimate β_0 and β_1 with least squares.

```
lm_fit <- mutate(train_set, y = as.numeric(sex == "Female")) %>%
  lm(y ~ height, data = .)
```

Once we have estimates $\hat{\beta}_0$ and $\hat{\beta}_1$, we can obtain an actual prediction. Our estimate of the conditional probability $p(x)$ is:

$$\hat{p}(x) = \hat{\beta}_0 + \hat{\beta}_1 x$$

To form a prediction, we define a *decision rule*: predict female if $\hat{p}(x) > 0.5$. We can compare our predictions to the outcomes using:

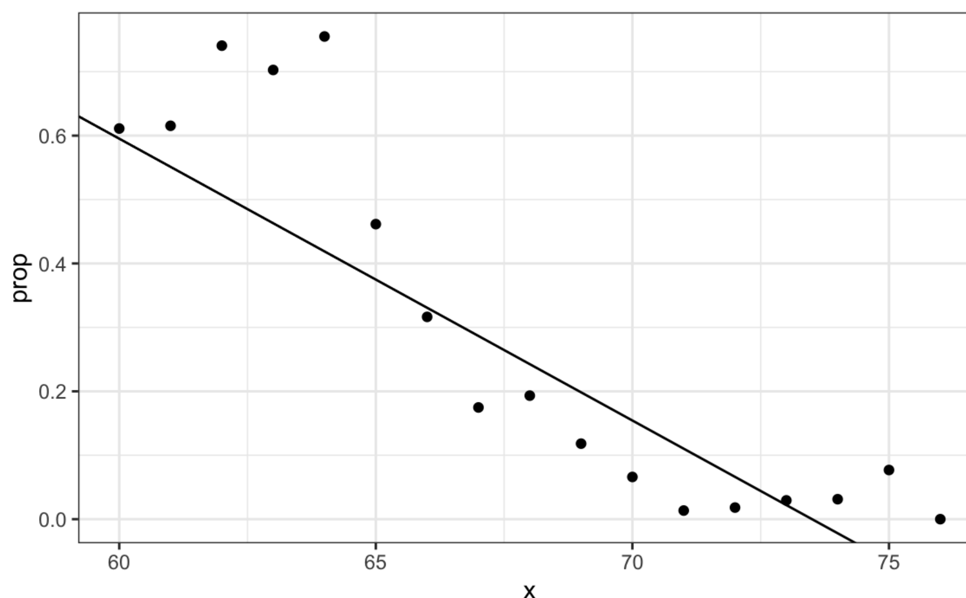
```
p_hat <- predict(lm_fit, test_set)
y_hat <- ifelse(p_hat > 0.5, "Female", "Male") %>% factor()
confusionMatrix(y_hat, test_set$sex)[["Accuracy"]]
#> NULL
```

We see this method does substantially better than guessing.

31.3.1 Generalized linear models

The function $\beta_0 + \beta_1 x$ can take any value including negatives and values larger than 1. In fact, the estimate $\hat{p}(x)$ computed in the linear regression section does indeed become negative at around 76 inches.

```
heights %>%
  mutate(x = round(height)) %>%
  group_by(x) %>%
  filter(n() >= 10) %>%
  summarize(prop = mean(sex == "Female")) %>%
  ggplot(aes(x, prop)) +
  geom_point() +
  geom_abline(intercept = lm_fit$coef[1], slope = lm_fit$coef[2])
#> `summarise()` ungrouping output (override with `.groups` argument)
```



The range is:

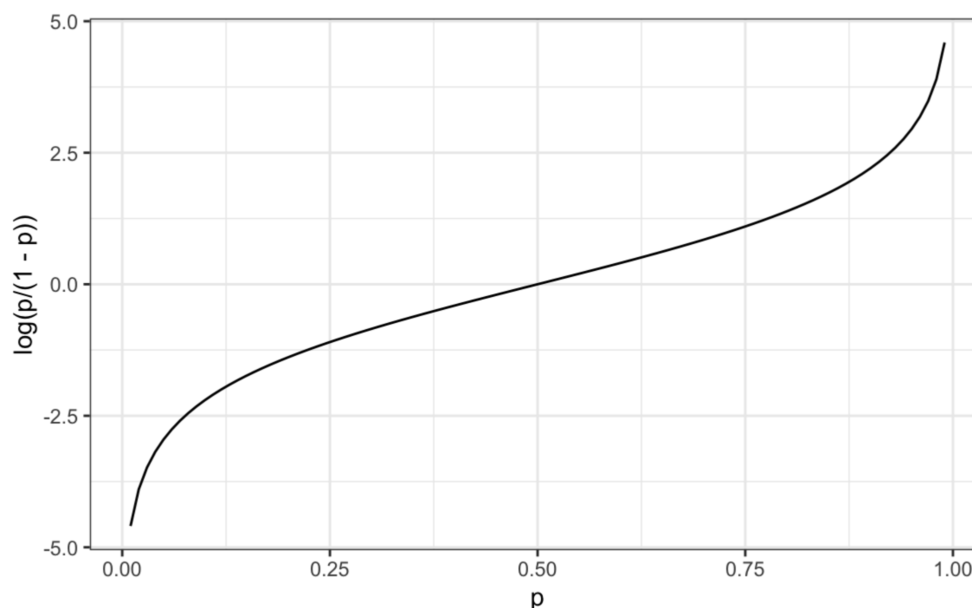
```
range(p_hat)
#> [1] -0.331  1.036
```

But we are estimating a probability: $\Pr(Y = 1 \mid X = x)$ which is constrained between 0 and 1.

The idea of generalized linear models (GLM) is to 1) define a distribution of Y that is consistent with it's possible outcomes and 2) find a function g so that $g(\Pr(Y = 1 \mid X = x))$ can be modeled as a linear combination of predictors. Logistic regression is the most commonly used GLM. It is an extension of linear regression that assures that the estimate of $\Pr(Y = 1 \mid X = x)$ is between 0 and 1. This approach makes use of the *logistic* transformation introduced in Section 9.8.1:

$$g(p) = \log \frac{p}{1-p}$$

This logistic transformation converts probability to log odds. As discussed in the data visualization lecture, the odds tell us how much more likely it is something will happen compared to not happening. $p = 0.5$ means the odds are 1 to 1, thus the odds are 1. If $p = 0.75$, the odds are 3 to 1. A nice characteristic of this transformation is that it converts probabilities to be symmetric around 0. Here is a plot of $g(p)$ versus p:



With *logistic regression*, we model the conditional probability directly with:

$$g\{\Pr(Y = 1 \mid X = x)\} = \beta_0 + \beta_1 x$$

With this model, we can no longer use least squares. Instead we compute the *maximum likelihood estimate* (MLE). You can learn more about this concept in a statistical theory textbook[107].

In R, we can fit the logistic regression model with the function `glm` : generalized linear models. This function is more general than logistic regression so we need to specify the model we want through the `family` parameter:

```
glm_fit <- train_set %>%
  mutate(y = as.numeric(sex == "Female")) %>%
  glm(y ~ height, data=., family = "binomial")
```

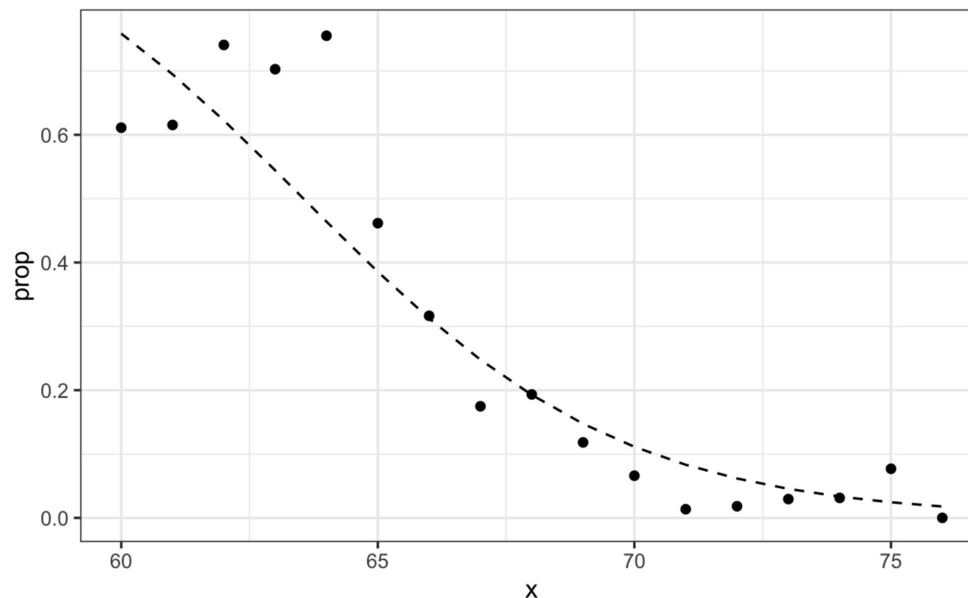
We can obtain prediction using the predict function:

```
p_hat_logit <- predict(glm_fit, newdata = test_set, type = "response")
```

When using `predict` with a `glm` object, we have to specify that we want `type="response"` if we want the conditional probabilities, since the default is to return the logistic transformed values.

This model fits the data slightly better than the line:

```
#> `summarise()` ungrouping output (override with `.groups` argument)
```

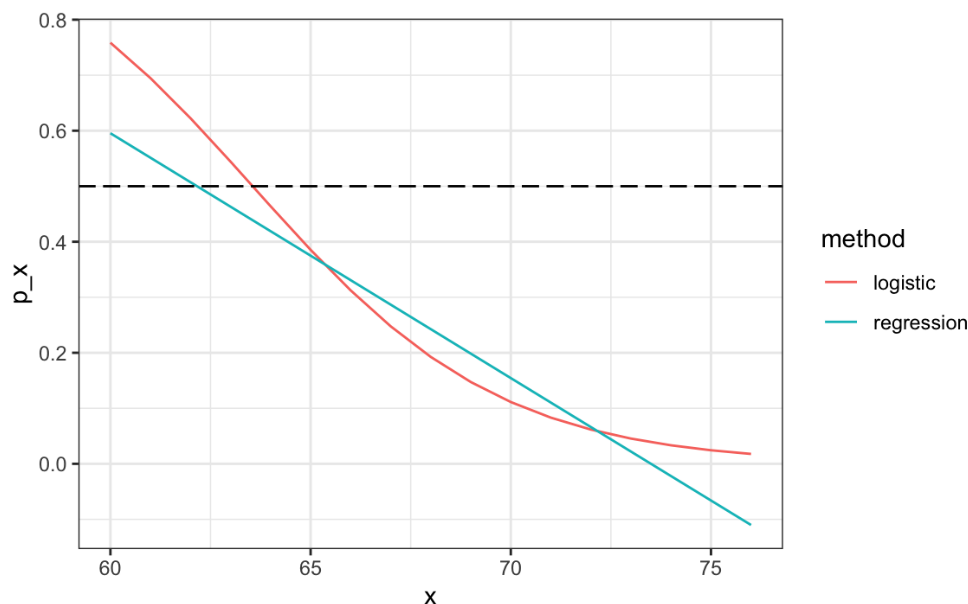


Because we have an estimate $\hat{p}(x)$, we can obtain predictions:

```
y_hat_logit <- ifelse(p_hat_logit > 0.5, "Female", "Male") %>% factor
confusionMatrix(y_hat_logit, test_set$sex)[["Accuracy"]]
#> NULL
```

The resulting predictions are similar. This is because the two estimates of $p(x)$ are larger than 1/2 in about the same region of x:

```
data.frame(x = seq(min(tmp$x), max(tmp$x))) %>%
  mutate(logistic = plogis(glm_fit$coef[1] + glm_fit$coef[2]*x),
         regression = lm_fit$coef[1] + lm_fit$coef[2]*x) %>%
  gather(method, p_x, -x) %>%
  ggplot(aes(x, p_x, color = method)) +
  geom_line() +
  geom_hline(yintercept = 0.5, lty = 5)
```



Both linear and logistic regressions provide an estimate for the conditional expectation:

$$\mathrm{E}(Y \mid X = x)$$

which in the case of binary data is equivalent to the conditional probability:

$$\Pr(Y = 1 \mid X = x)$$

31.3.2 Logistic regression with more than one predictor

In this section we apply logistic regression to the two or seven data introduced in Section 27.8. In this case, we are interested in estimating a conditional probability that depends on two variables. The standard logistic regression model in this case will assume that

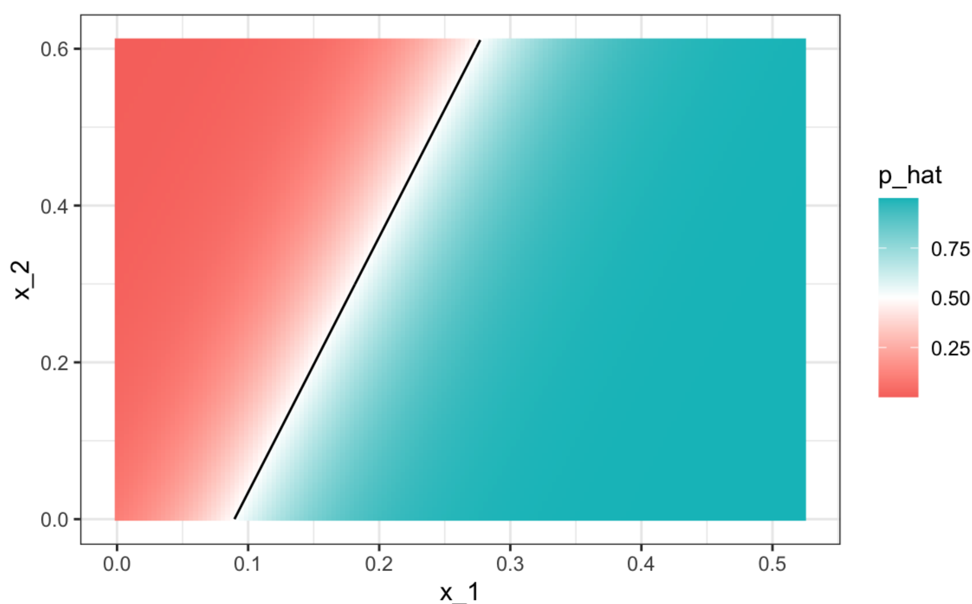
$$g\{p(x_1, x_2)\} = g\{\Pr(Y = 1 \mid X_1 = x_1, X_2 = x_2)\} = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

with $g(p) = \log \frac{p}{1-p}$ the logistic function described in the previous section. To fit the model we use the following code:

```
fit_glm <- glm(y ~ x_1 + x_2, data=mnist_27$train, family = "binomial")
p_hat_glm <- predict(fit_glm, mnist_27$test, type="response")
y_hat_glm <- factor(ifelse(p_hat_glm > 0.5, 7, 2))
confusionMatrix(y_hat_glm, mnist_27$test$y)$overall["Accuracy"]
#> Accuracy
#>     0.75
```

Comparing to the results we obtained in Section 27.8, we see that logistic regression performs similarly to regression. This is not surprising, given that the estimate of $\hat{p}(x_1, x_2)$ looks similar as well:

```
p_hat <- predict(fit_glm, newdata = mnist_27$true_p, type = "response")
mnist_27$true_p %>% mutate(p_hat = p_hat) %>%
  ggplot(aes(x_1, x_2,  z=p_hat, fill=p_hat)) +
  geom_raster() +
  scale_fill_gradientn(colors=c("#F8766D","white","#00BFC4")) +
  stat_contour(breaks=c(0.5), color="black")
```



Just like regression, the decision rule is a line, a fact that can be corroborated mathematically since

$$g^{-1}(\hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2) = 0.5 \implies \hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 = g(0.5) = 0 \implies x_2 = -\hat{\beta}_0/\hat{\beta}_2 - \hat{\beta}_1$$

Thus x_2 is a linear function of x_1. This implies that, just like regression, our logistic regression approach has no chance of capturing the non-linear nature of the true $p(x_1, x_2)$. Once we move on to more complex examples, we will see that linear regression and generalized linear regression are limited and not flexible enough to be useful for most machine learning challenges. The new techniques we learn are essentially approaches to estimating the conditional probability in a way that is more flexible.

31.4 Exercises

1. Define the following dataset:

```
make_data <- function(n = 1000, p = 0.5,
                      mu_0 = 0, mu_1 = 2,
                      sigma_0 = 1,  sigma_1 = 1){
  y <- rbinom(n, 1, p)
  f_0 <- rnorm(n, mu_0, sigma_0)
  f_1 <- rnorm(n, mu_1, sigma_1)
  x <- ifelse(y == 1, f_1, f_0)
  test_index <- createDataPartition(y, times = 1, p = 0.5, list = FALSE)
  list(train = data.frame(x = x, y = as.factor(y)) %>%
         slice(-test_index),
       test = data.frame(x = x, y = as.factor(y)) %>%
         slice(test_index))
}
dat <- make_data()
```

Note that we have defined a variable `x` that is predictive of a binary outcome `y`.

```
dat$train %>% ggplot(aes(x, color = y)) + geom_density()
```

Compare the accuracy of linear regression and logistic regression.

2. Repeat the simulation from exercise 1 100 times and compare the average accuracy for each method and notice they give practically the same answer.

3. Generate 25 different datasets changing the difference between the two class: `delta <- seq(0, 3, len = 25)`. Plot accuracy versus `delta`.

31.5 k-nearest neighbors

We introduced the kNN algorithm in Section 29.1 and demonstrated how we use cross validation to pick k in Section 30.2. Here we quickly review how we fit a kNN model using the **caret** package. In Section 30.2 we introduced the following code to fit a kNN model:

```
train_knn <- train(y ~ ., method = "knn",
                   data = mnist_27$train,
                   tuneGrid = data.frame(k = seq(9, 71, 2)))
```

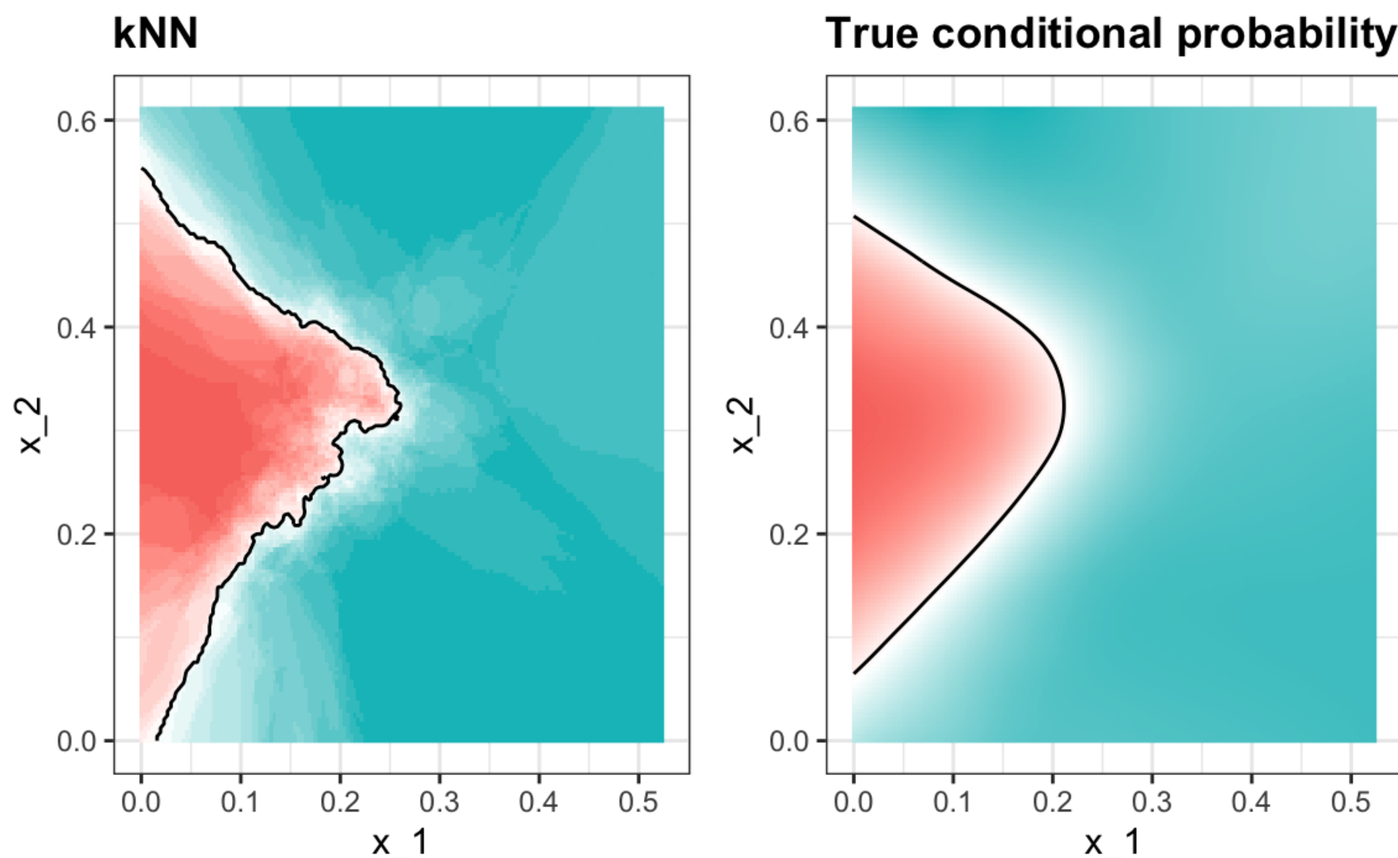
We saw that the parameter that maximized the estimated accuracy was:

```
train_knn$bestTune
#>     k
#> 10 27
```

This model improves the accuracy over regression and logistic regression:

```
confusionMatrix(predict(train_knn, mnist_27$test, type = "raw"),
                mnist_27$test$y)$overall["Accuracy"]
#> Accuracy
#>    0.835
```

A plot of the estimated conditional probability shows that the kNN estimate is flexible enough and does indeed capture the shape of the true conditional probability.



31.6 Exercises

1. Earlier we used logistic regression to predict sex from height. Use kNN to do the same. Use the code described in this chapter to select the F_1 measure and plot it against k. Compare to the F_1 of about 0.6 we obtained with regression.

2. Load the following dataset:

```
data("tissue_gene_expression")
```

This dataset includes a matrix `x` :

```
dim(tissue_gene_expression$x)
```

with the gene expression measured on 500 genes for 189 biological samples representing seven different tissues. The tissue type is stored in `y` :

```
table(tissue_gene_expression$y)
```

Split the data in training and test sets, then use kNN to predict tissue type and see what accuracy you obtain. Try it for $k = 1, 3, \dots, 11$.

31.7 Generative models

We have described how, when using squared loss, the conditional expectation/probabilities provide the best approach to developing a decision rule. In a binary case, the smallest true error we can achieve is determined by Bayes' rule, which is a decision rule based on the true conditional probability:

$$p(\mathbf{x}) = \mathrm{Pr}(Y = 1 \mid \mathbf{X} = \mathbf{x})$$

We have described several approaches to estimating $p(\mathbf{x})$. In all these approaches, we estimate the conditional probability directly and do not consider the distribution of the predictors. In machine learning, these are referred to as *discriminative* approaches.

However, Bayes' theorem tells us that knowing the distribution of the predictors $\mathbf{X}$ may be useful. Methods that model the joint distribution of Y and $\mathbf{X}$ are referred to as *generative models* (we model how the entire data, $\mathbf{X}$ and Y, are generated). We start by describing the most general generative model, Naive Bayes, and then proceed to describe two specific cases, quadratic discriminant analysis (QDA) and linear discriminant analysis (LDA).

31.7.1 Naive Bayes

Recall that Bayes rule tells us that we can rewrite $p(\mathbf{x})$ like this:

$$p(\mathbf{x}) = \mathrm{Pr}(Y = 1|\mathbf{X} = \mathbf{x}) = \frac{f_{\mathbf{X}|Y=1}(\mathbf{x})\mathrm{Pr}(Y = 1)}{f_{\mathbf{X}|Y=0}(\mathbf{x})\mathrm{Pr}(Y = 0) + f_{\mathbf{X}|Y=1}(\mathbf{x})\mathrm{Pr}(Y = 1)}$$

with $f_{\mathbf{X}|Y=1}$ and $f_{\mathbf{X}|Y=0}$ representing the distribution functions of the predictor $\mathbf{X}$ for the two classes $Y = 1$ and $Y = 0$. The formula implies that if we can estimate these conditional distributions of the predictors, we can develop a powerful decision rule. However, this is a big *if*. As we go forward, we will encounter examples in which $\mathbf{X}$ has many dimensions and we do not have much information about the distribution. In these cases, Naive Bayes will be practically

impossible to implement. However, there are instances in which we have a small number of predictors (not much more than 2) and many categories in which generative models can be quite powerful. We describe two specific examples and use our previously described case studies to illustrate them.

Let's start with a very simple and uninteresting, yet illustrative, case: the example related to predicting sex from height.

```
library(tidyverse)
library(caret)

library(dslabs)
data("heights")

y <- heights$height
set.seed(1995)
test_index <- createDataPartition(y, times = 1, p = 0.5, list = FALSE)
train_set <- heights %>% slice(-test_index)
test_set <- heights %>% slice(test_index)
```

In this case, the Naive Bayes approach is particularly appropriate because we know that the normal distribution is a good approximation for the conditional distributions of height given sex for both classes $Y = 1$ (female) and $Y = 0$ (male). This implies that we can approximate the conditional distributions $f_{X|Y=1}$ and $f_{X|Y=0}$ by simply estimating averages and standard deviations from the data:

```
params <- train_set %>%
  group_by(sex) %>%
  summarize(avg = mean(height), sd = sd(height))
#> `summarise()` ungrouping output (override with `.groups` argument)
params
#> # A tibble: 2 x 3
#>   sex      avg    sd
#>   <fct>  <dbl> <dbl>
#> 1 Female  64.8  4.14
#> 2 Male    69.2  3.57
```

The prevalence, which we will denote with $\pi = \Pr(Y = 1)$, can be estimated from the data with:

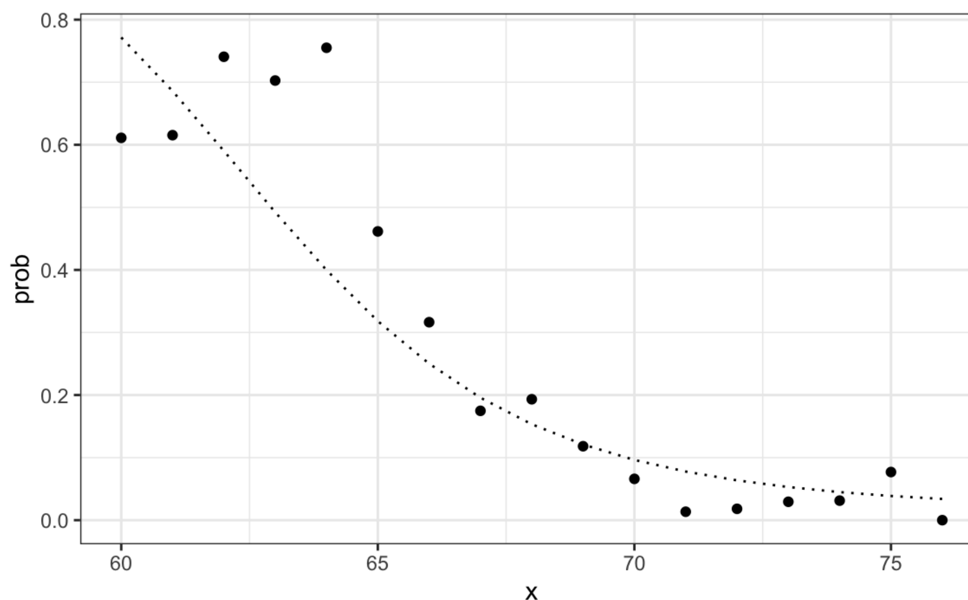
```
pi <- train_set %>% summarize(pi=mean(sex=="Female")) %>% pull(pi)
pi
#> [1] 0.212
```

Now we can use our estimates of average and standard deviation to get an actual rule:

```
x <- test_set$height

f0 <- dnorm(x, params$avg[2], params$sd[2])
f1 <- dnorm(x, params$avg[1], params$sd[1])

p_hat_bayes <- f1*pi / (f1*pi + f0*(1 - pi))
```

Our Naive Bayes estimate $\hat{p}(x)$ looks a lot like our logistic regression estimate:

```
#> `summarise()` ungrouping output (override with `.groups` argument)
```



In fact, we can show that the Naive Bayes approach is similar to the logistic regression prediction mathematically. However, we leave the demonstration to a more advanced text, such as the Elements of Statistical Learning[108]. We can see that they are similar empirically by comparing the two resulting curves.

31.7.2 Controlling prevalence

One useful feature of the Naive Bayes approach is that it includes a parameter to account for differences in prevalence. Using our sample, we estimated $f_{X|Y=1}$, $f_{X|Y=0}$ and π. If we use hats to denote the estimates, we can write $\hat{p}(x)$ as:

$$\hat{p}(x) = \frac{\hat{f}_{X|Y=1}(x)\hat{\pi}}{\hat{f}_{X|Y=0}(x)(1-\hat{\pi}) + \hat{f}_{X|Y=1}(x)\hat{\pi}}$$

As we discussed earlier, our sample has a much lower prevalence, 0.21, than the general population. So if we use the rule $\hat{p}(x) > 0.5$ to predict females, our accuracy will be affected due to the low sensitivity:

```
y_hat_bayes <- ifelse(p_hat_bayes > 0.5, "Female", "Male")
sensitivity(data = factor(y_hat_bayes), reference = factor(test_set$sex))
#> [1] 0.213
```

Again, this is because the algorithm gives more weight to specificity to account for the low prevalence:

```
specificity(data = factor(y_hat_bayes), reference = factor(test_set$sex))
#> [1] 0.967
```

This is due mainly to the fact that $\hat{\pi}$ is substantially less than 0.5, so we tend to predict `Male` more often. It makes sense for a machine learning algorithm to do this in our sample because we do have a higher percentage of males. But if we were to extrapolate this to a general population, our overall accuracy would be affected by the low sensitivity.

The Naive Bayes approach gives us a direct way to correct this since we can simply force $\hat{\pi}$ to be whatever value we want it to be. So to balance specificity and sensitivity, instead of changing the cutoff in the decision rule, we could simply change $\hat{\pi}$ to 0.5 like this:

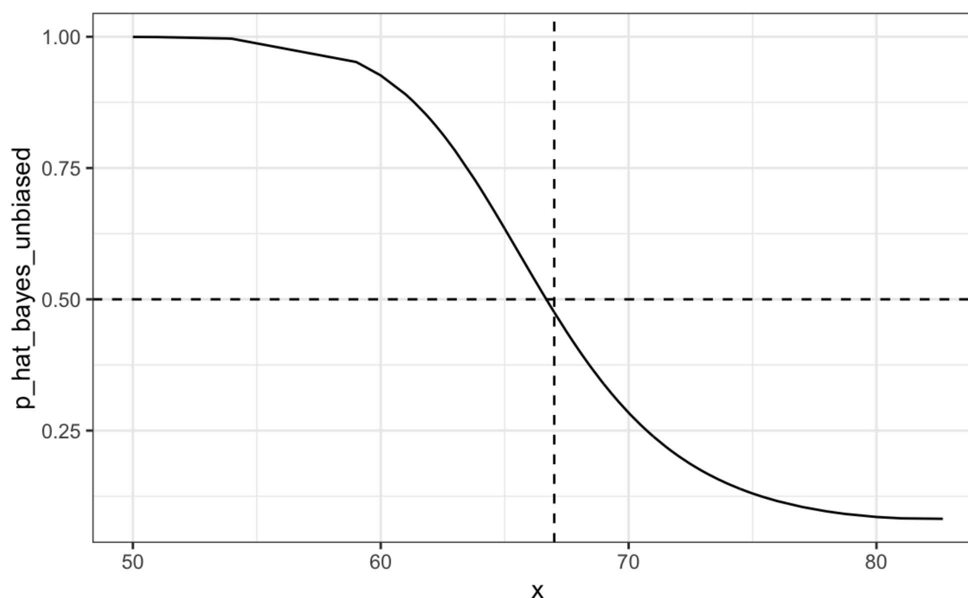
```
p_hat_bayes_unbiased <- f1 * 0.5 / (f1 * 0.5 + f0 * (1 - 0.5))
y_hat_bayes_unbiased <- ifelse(p_hat_bayes_unbiased> 0.5, "Female", "Male")
```

Note the difference in sensitivity with a better balance:

```
sensitivity(factor(y_hat_bayes_unbiased), factor(test_set$sex))
#> [1] 0.693
specificity(factor(y_hat_bayes_unbiased), factor(test_set$sex))
#> [1] 0.832
```

The new rule also gives us a very intuitive cutoff between 66-67, which is about the middle of the female and male average heights:

```
qplot(x, p_hat_bayes_unbiased, geom = "line") +
  geom_hline(yintercept = 0.5, lty = 2) +
  geom_vline(xintercept = 67, lty = 2)
```



31.7.3 Quadratic discriminant analysis

Quadratic Discriminant Analysis (QDA) is a version of Naive Bayes in which we assume that the distributions $p_{\mathbf{X}|Y=1}(x)$ and $p_{\mathbf{X}|Y=0}(\mathbf{x})$ are multivariate normal. The simple example we described in the previous section is actually QDA. Let's now look at a slightly more complicated case: the 2 or 7 example.

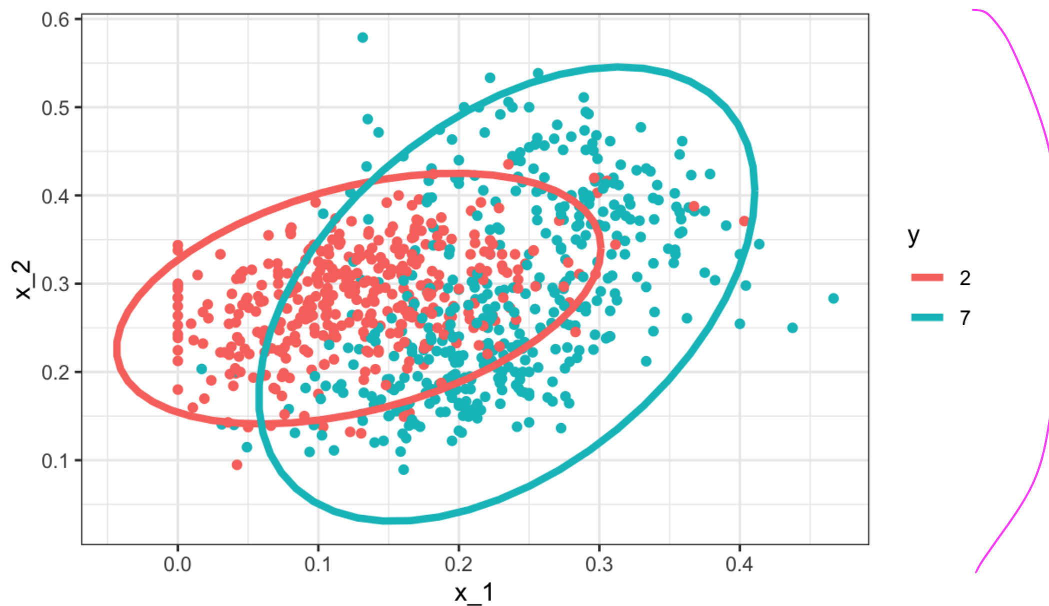
```
data("mnist_27")
```

In this case, we have two predictors so we assume each one is bivariate normal. This implies that we need to estimate two averages, two standard deviations, and a correlation for each case $Y = 1$ and $Y = 0$. Once we have these, we can approximate the distributions $f_{X_1,X_2|Y=1}$ and $f_{X_1,X_2|Y=0}$. We can easily estimate parameters from the data:

```
params <- mnist_27$train %>%
  group_by(y) %>%
  summarize(avg_1 = mean(x_1), avg_2 = mean(x_2),
            sd_1= sd(x_1), sd_2 = sd(x_2),
            r = cor(x_1, x_2))
#> `summarise()` ungrouping output (override with `.groups` argument)
params
#> # A tibble: 2 x 6
#>   y     avg_1 avg_2   sd_1   sd_2     r
#>   <fct> <dbl> <dbl>  <dbl>  <dbl> <dbl>
#> 1 2     0.129 0.283 0.0702 0.0578 0.401
#> 2 7     0.234 0.288 0.0719 0.105  0.455
```

Here we provide a visual way of showing the approach. We plot the data and use contour plots to give an idea of what the two estimated normal densities look like (we show the curve representing a region that includes 95% of the points):

```
mnist_27$train %>% mutate(y = factor(y)) %>%
  ggplot(aes(x_1, x_2, fill = y, color=y)) +
  geom_point(show.legend = FALSE) +
  stat_ellipse(type="norm", lwd = 1.5)
```



This defines the following estimate of $f(x_1, x_2)$.

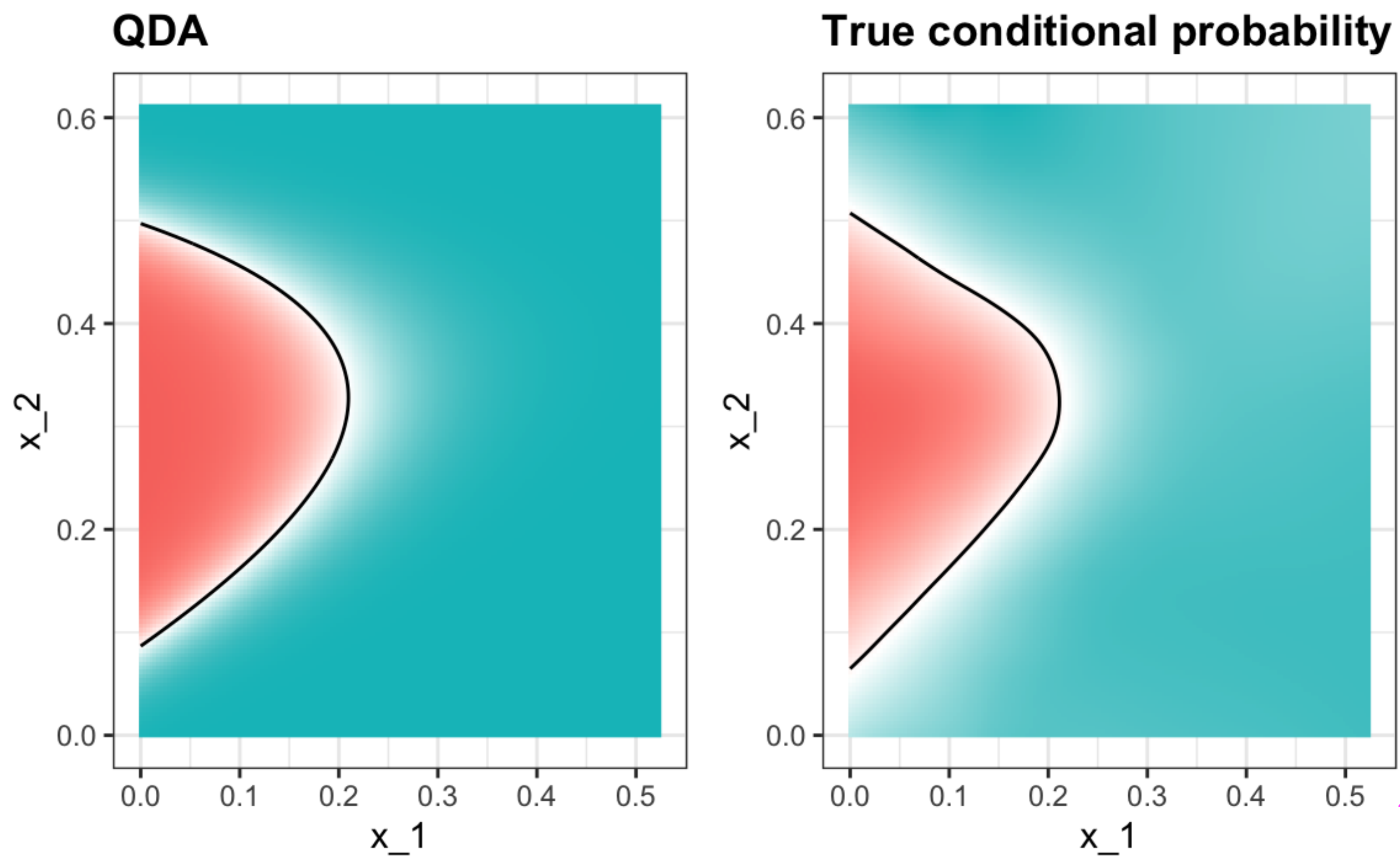
We can use the `train` function from the **caret** package to fit the model and obtain predictors:

```
library(caret)
train_qda <- train(y ~ ., method = "qda", data = mnist_27$train)
```

We see that we obtain relatively good accuracy:

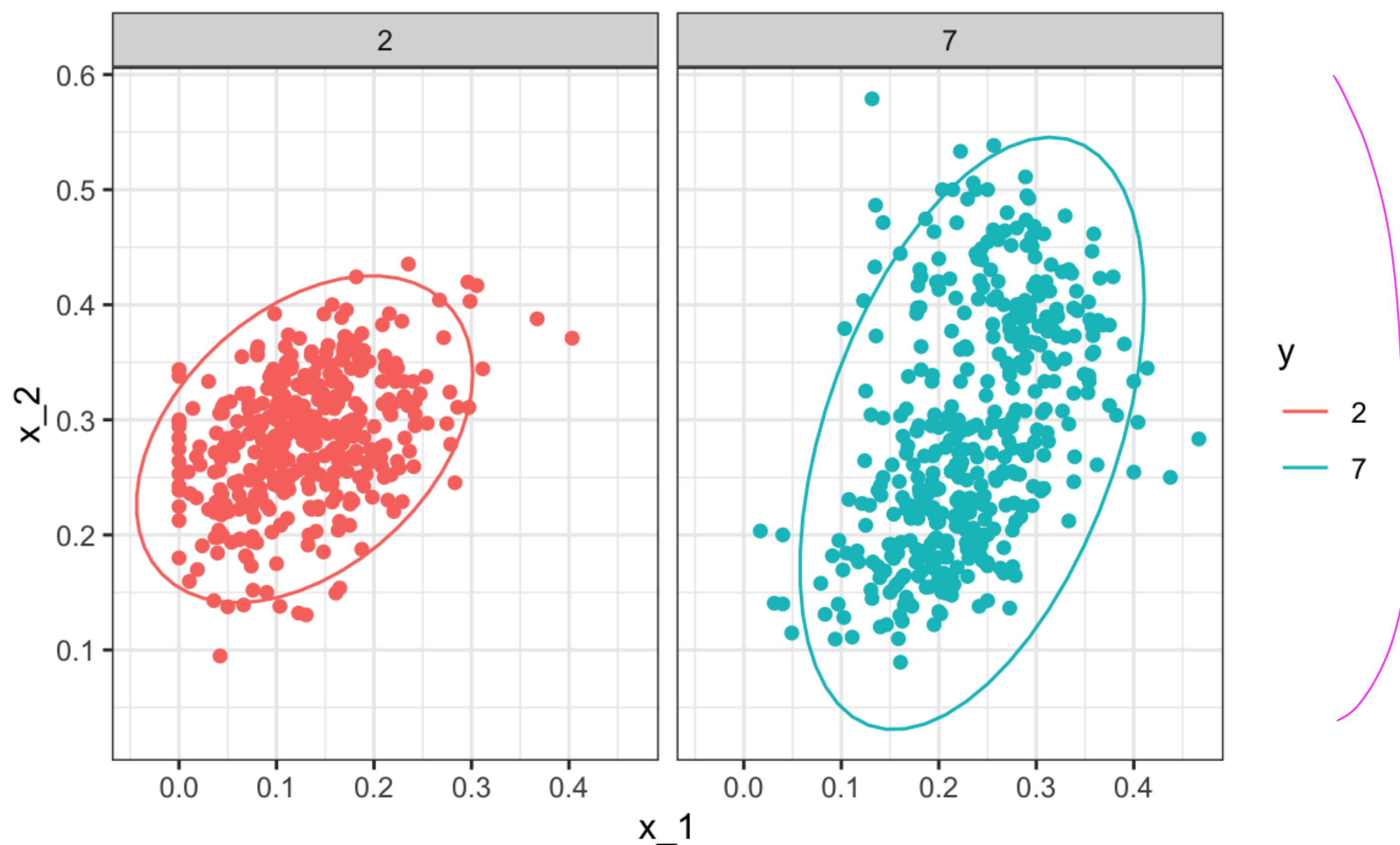
```
y_hat <- predict(train_qda, mnist_27$test)
confusionMatrix(y_hat, mnist_27$test$y)$overall["Accuracy"]
#> Accuracy
#>     0.82
```

The estimated conditional probability looks relatively good, although it does not fit as well as the kernel smoothers:



One reason QDA does not work as well as the kernel methods is perhaps because the assumption of normality does not quite hold. Although for the 2s it seems reasonable, for the 7s it does seem to be off. Notice the slight curvature in the points for the 7s:

```
mnist_27$train %>% mutate(y = factor(y)) %>%
  ggplot(aes(x_1, x_2, fill = y, color=y)) +
  geom_point(show.legend = FALSE) +
  stat_ellipse(type="norm") +
  facet_wrap(~y)
```

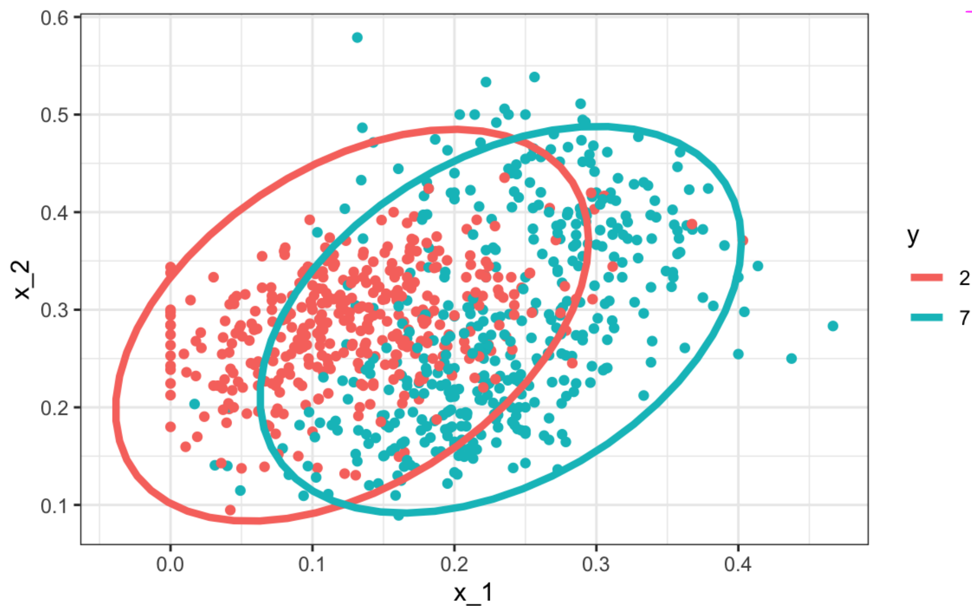


QDA can work well here, but it becomes harder to use as the number of predictors increases. Here we have 2 predictors and had to compute 4 means, 4 SDs, and 2 correlations. How many parameters would we have if instead of 2 predictors, we had 10? The main problem comes from estimating correlations for 10 predictors. With 10, we have 45 correlations for each class. In general, the formula is $K \times p(p-1)/2$, which gets big fast. Once the number of parameters approaches the size of our data, the method becomes impractical due to overfitting.

31.7.4 Linear discriminant analysis

A relatively simple solution to the problem of having too many parameters is to assume that the correlation structure is the same for all classes, which reduces the number of parameters we need to estimate.

In this case, we would compute just one pair of standard deviations and one correlation, and the distributions looks like this:



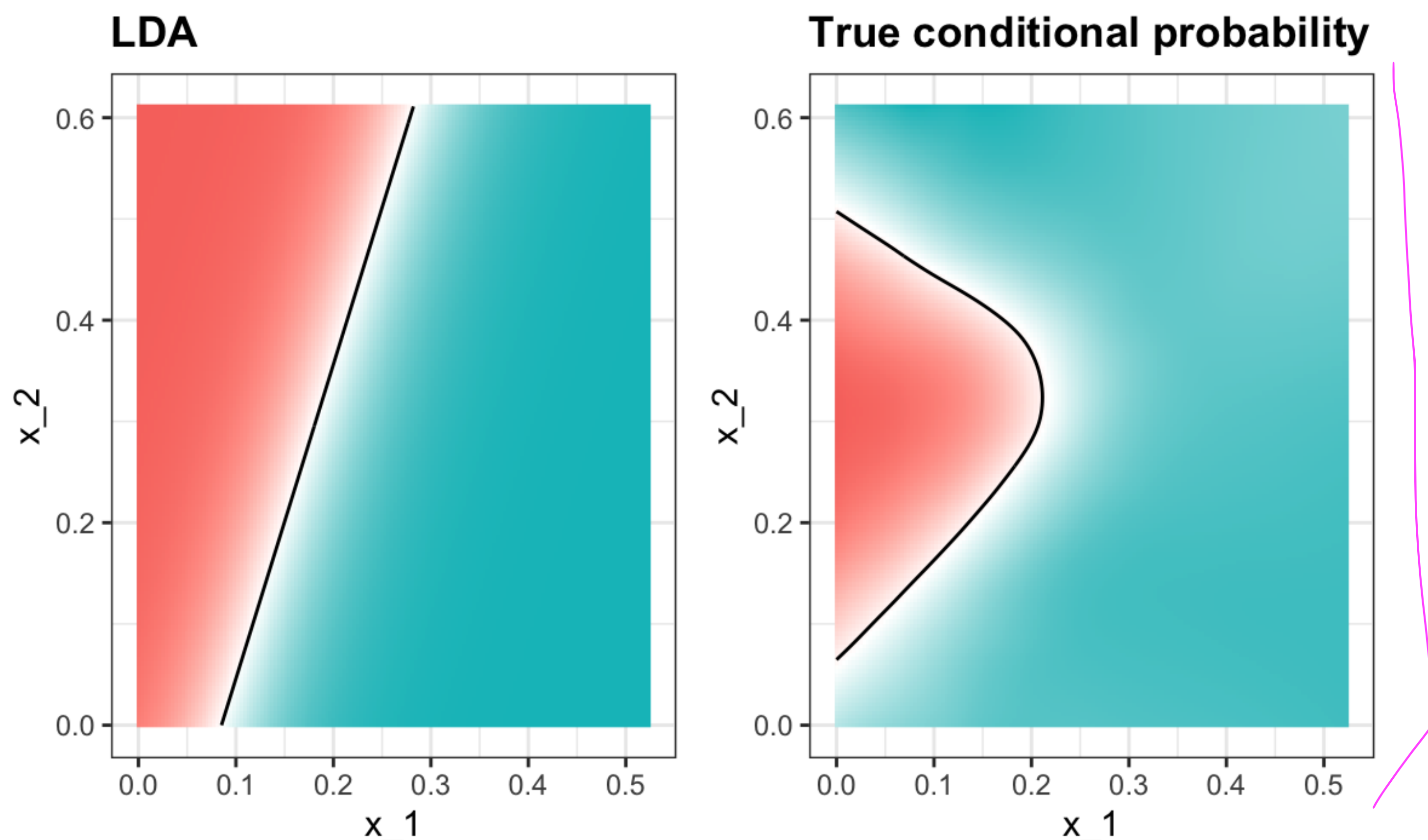
Now the size of the ellipses as well as the angle are the same. This is because they have the same standard deviations and correlations.

We can fit the LDA model using **caret**:

```
train_lda <- train(y ~ ., method = "lda", data = mnist_27$train)
y_hat <- predict(train_lda, mnist_27$test)
confusionMatrix(y_hat, mnist_27$test$y)$overall["Accuracy"]
#> Accuracy
#>     0.75
```

It was .82 for QDA.

When we force this assumption, we can show mathematically that the boundary is a line, just as with logistic regression. For this reason, we call the method *linear* discriminant analysis (LDA). Similarly, for QDA, we can show that the boundary must be a quadratic function.



In the case of LDA, the lack of flexibility does not permit us to capture the non-linearity in the true conditional probability function.

31.7.5 Connection to distance

The normal density is:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{ -\frac{(x-\mu)^2}{\sigma^2} \right\}$$

If we remove the constant $1/(\sqrt{2\pi}\sigma)$ and then take the log, we get:

$$-\frac{(x-\mu)^2}{\sigma^2}$$

which is the negative of a distance squared scaled by the standard deviation. For higher dimensions, the same is true except the scaling is more complex and involves correlations.

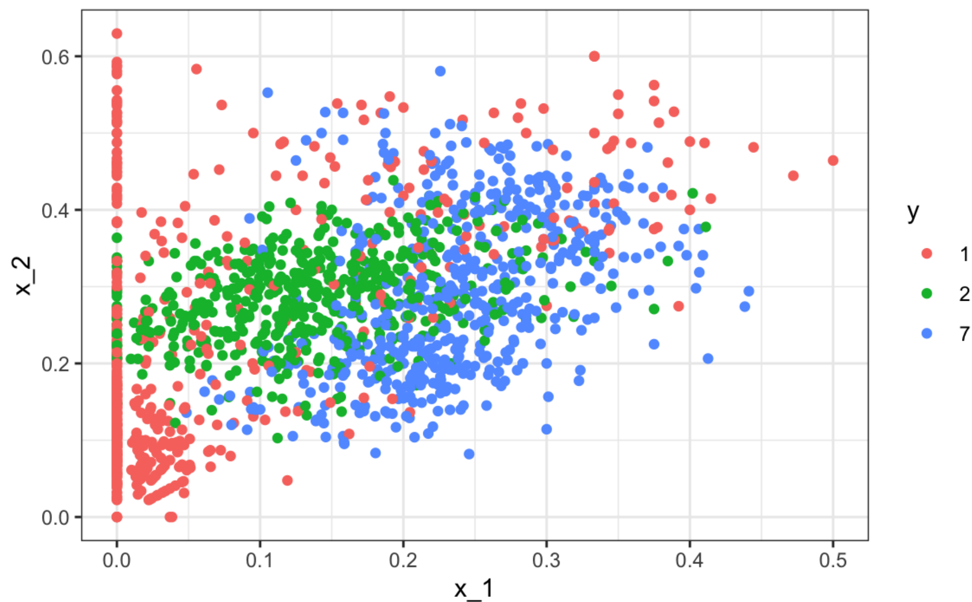
31.8 Case study: more than three classes

We can generate an example with three categories like this:

```
if(!exists("mnist")) mnist <- read_mnist()
set.seed(3456)
index_127 <- sample(which(mnist$train$labels %in% c(1,2,7)), 2000)
y <- mnist$train$labels[index_127]
x <- mnist$train$images[index_127,]
index_train <- createDataPartition(y, p=0.8, list = FALSE)
## get the quadrants
row_column <- expand.grid(row=1:28, col=1:28)
upper_left_ind <- which(row_column$col <= 14 & row_column$row <= 14)
lower_right_ind <- which(row_column$col > 14 & row_column$row > 14)
## binarize the values. Above 200 is ink, below is no ink
x <- x > 200
## proportion of pixels in lower right quadrant
x <- cbind(rowSums(x[ ,upper_left_ind])/rowSums(x),
           rowSums(x[ ,lower_right_ind])/rowSums(x))
##save data
train_set <- data.frame(y = factor(y[index_train]),
                        x_1 = x[index_train,1], x_2 = x[index_train,2])
test_set <- data.frame(y = factor(y[-index_train]),
                       x_1 = x[-index_train,1], x_2 = x[-index_train,2])
```

Here is the training data:

```
train_set %>% ggplot(aes(x_1, x_2, color=y)) + geom_point()
```



We can use the **caret** package to train the QDA model:

```
train_qda <- train(y ~ ., method = "qda", data = train_set)
```

Now we estimate three conditional probabilities (although they have to add to 1):

```
predict(train_qda, test_set, type = "prob") %>% head()
#>        1       2       7
#> 1 0.7655 0.23043 0.00405
#> 2 0.2031 0.72514 0.07175
#> 3 0.5396 0.45909 0.00132
#> 4 0.0393 0.09419 0.86655
#> 5 0.9600 0.00936 0.03063
#> 6 0.9865 0.00724 0.00623
```

Our predictions are one of the three classes:

```
predict(train_qda, test_set) %>% head()
#> [1] 1 2 1 7 1 1
#> Levels: 1 2 7
```

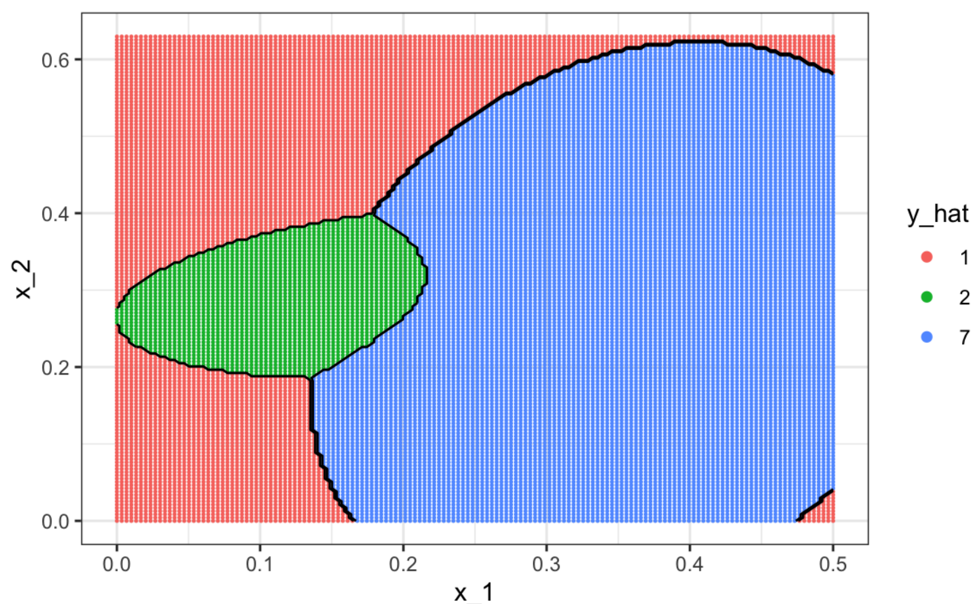
The confusion matrix is therefore a 3 by 3 table:

```
confusionMatrix(predict(train_qda, test_set), test_set$y)$table
#>           Reference
#> Prediction   1   2   7
#>          1 111   9  11
#>          2  10  86  21
#>          7  21  28 102
```

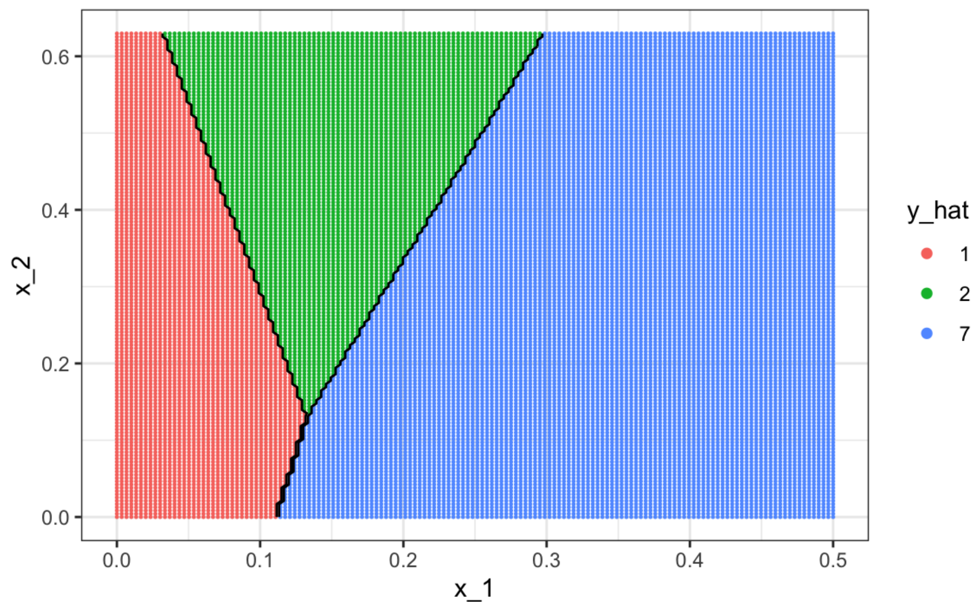
The accuracy is 0.749

Note that for sensitivity and specificity, we have a pair of values for **each** class. To define these terms, we need a binary outcome. We therefore have three columns: one for each class as the positives and the other two as the negatives.

To visualize what parts of the region are called 1, 2, and 7 we now need three colors:



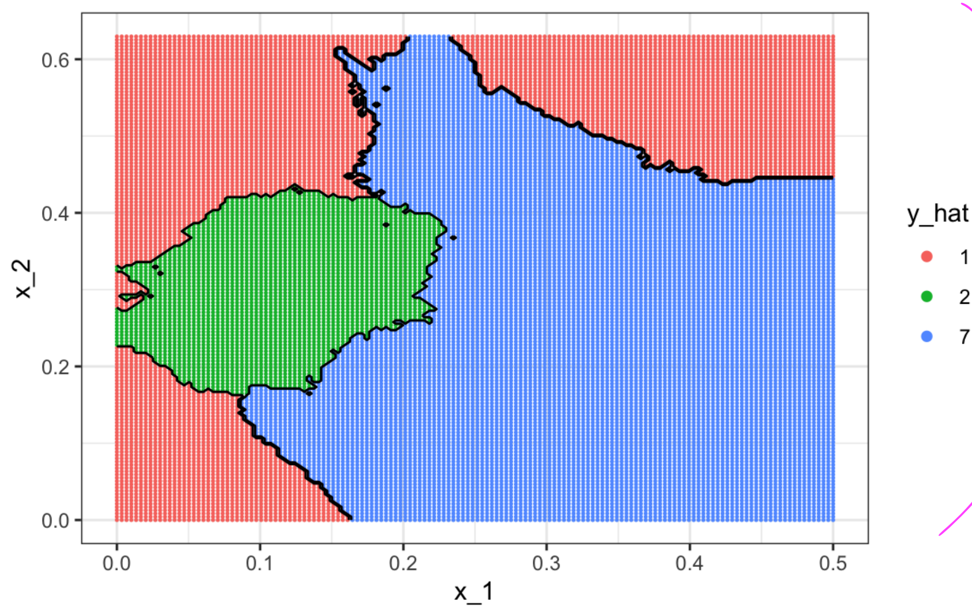
The accuracy for LDA, 0.629, is much worse because the model is more rigid. This is what the decision rule looks like:



The results for kNN

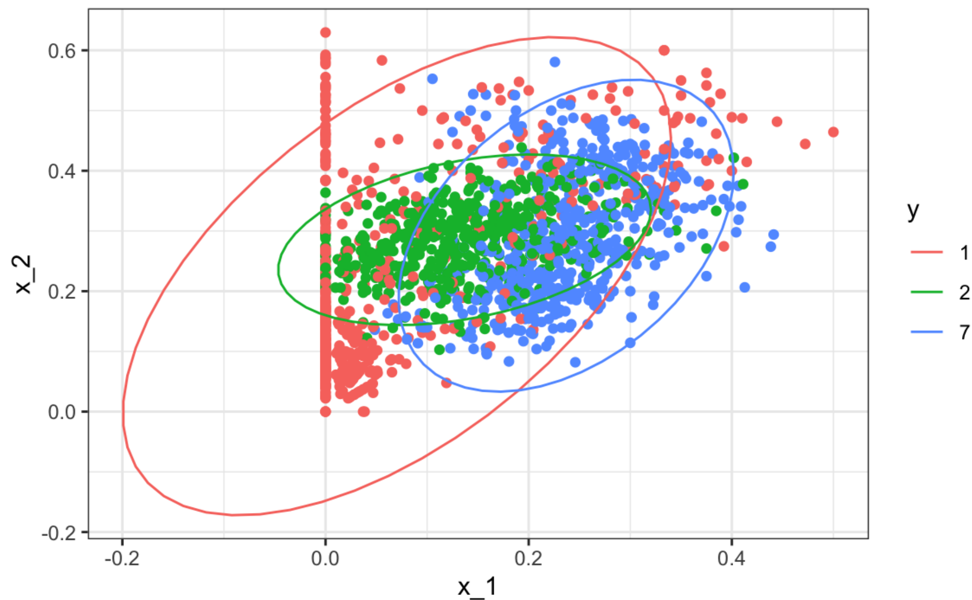
```
train_knn <- train(y ~ ., method = "knn", data = train_set,
                   tuneGrid = data.frame(k = seq(15, 51, 2)))
```

are much better with an accuracy of 0.749. The decision rule looks like this:



Note that one of the limitations of generative models here is due to the lack of fit of the normal assumption, in particular for class 1.

```
train_set %>% mutate(y = factor(y)) %>%
  ggplot(aes(x_1, x_2, fill = y, color=y)) +
  geom_point(show.legend = FALSE) +
  stat_ellipse(type="norm")
```



Generative models can be very powerful, but only when we are able to successfully approximate the joint distribution of predictors conditioned on each class.

31.9 Exercises

We are going to apply LDA and QDA to the `tissue_gene_expression` dataset. We will start with simple examples based on this dataset and then develop a realistic example.

1. Create a dataset with just the classes "cerebellum" and "hippocampus" (two parts of the brain) and a predictor matrix with 10 randomly selected columns.

```
set.seed(1993)
data("tissue_gene_expression")
tissues <- c("cerebellum", "hippocampus")
ind <- which(tissue_gene_expression$y %in% tissues)
y <- droplevels(tissue_gene_expression$y[ind])
x <- tissue_gene_expression$x[ind, ]
x <- x[, sample(ncol(x), 10)]
```

Use the `train` function to estimate the accuracy of LDA.

2. In this case, LDA fits two 10-dimensional normal distributions. Look at the fitted model by looking at the `finalModel` component of the result of train. Notice there is a component called `means` that includes the estimate `means` of both distributions. Plot the mean vectors against each other and determine which predictors (genes) appear to be driving the algorithm.

3. Repeat exercises 1 with QDA. Does it have a higher accuracy than LDA?

4. Are the same predictors (genes) driving the algorithm? Make a plot as in exercise 2.

5. One thing we see in the previous plot is that the value of predictors correlate in both groups: some predictors are low in both groups while others are high in both groups. The mean value of each predictor, `colMeans(x)`, is not informative or useful for prediction, and often for interpretation purposes it is useful to center or scale each column. This can be achieved with the `preProcessing` argument in `train`. Re-run LDA with `preProcessing = "scale"`. Note that accuracy does not change but see how it is easier to identify the predictors that differ more between groups in the plot made in exercise 4.

6. In the previous exercises we saw that both approaches worked well. Plot the predictor values for the two genes with the largest differences between the two groups in a scatterplot to see how they appear to follow a bivariate distribution as assumed by the LDA and QDA approaches. Color the points by the outcome.

7. Now we are going to increase the complexity of the challenge slightly: we will consider all the tissue types.

```
set.seed(1993)
data("tissue_gene_expression")
y <- tissue_gene_expression$y
x <- tissue_gene_expression$x
x <- x[, sample(ncol(x), 10)]
```

What accuracy do you get with LDA?

8. We see that the results are slightly worse. Use the `confusionMatrix` function to learn what type of errors we are making.

9. Plot an image of the centers of the seven 10-dimensional normal distributions.

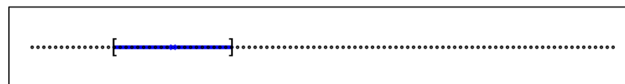
31.10 Classification and regression trees (CART)

31.10.1 The curse of dimensionality

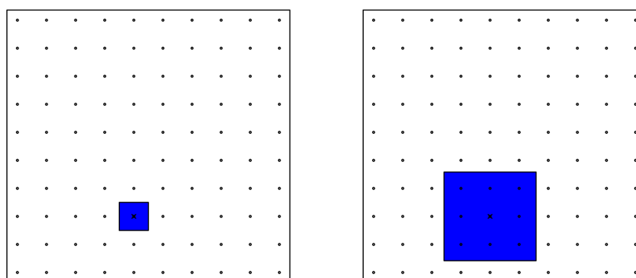
We described how methods such as LDA and QDA are not meant to be used with many predictors p because the number of parameters that we need to estimate becomes too large. For example, with the digits example $p = 784$, we would have over 600,000 parameters with LDA, and we would multiply that by the number of classes for QDA. Kernel methods such as kNN or local regression do not have model parameters to estimate. However, they also face a challenge when multiple predictors are used due to what is referred to as the *curse of dimensionality*. The *dimension* here refers to the fact that when we have p predictors, the distance between two observations is computed in p-dimensional space.

A useful way of understanding the curse of dimensionality is by considering how large we have to make a span/neighborhood/window to include a given percentage of the data. Remember that with larger neighborhoods, our methods lose flexibility.

For example, suppose we have one continuous predictor with equally spaced points in the [0,1] interval and we want to create windows that include 1/10th of data. Then it’s easy to see that our windows have to be of size 0.1:

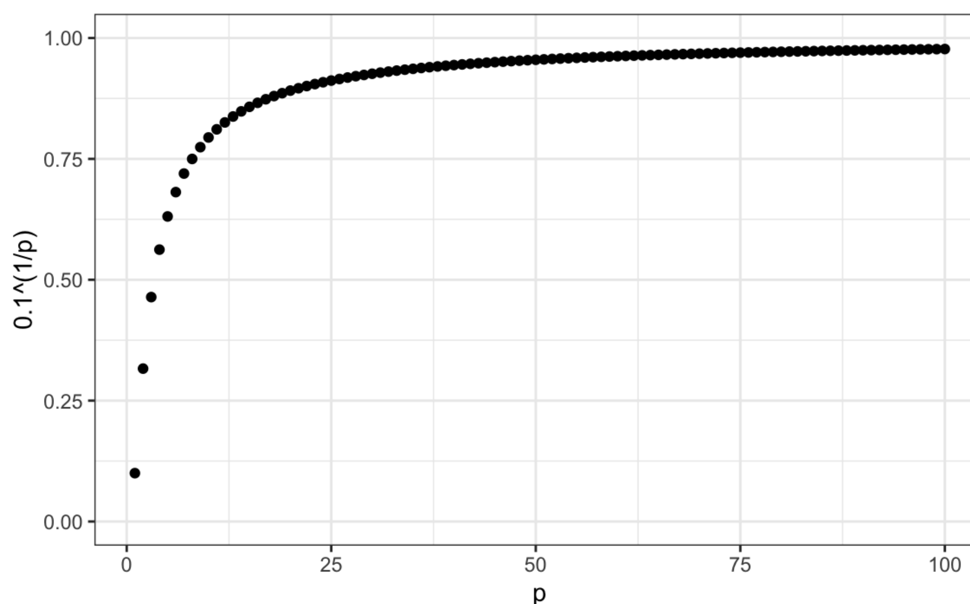


Now, for two predictors, if we decide to keep the neighborhood just as small, 10% for each dimension, we include only 1 point. If we want to include 10% of the data, then we need to increase the size of each side of the square to $\sqrt{.10} \approx .316$:



Using the same logic, if we want to include 10% of the data in a three-dimensional space, then the side of each cube is $\sqrt[3]{.10} \approx 0.464$. In general, to include 10% of the data in a case with p dimensions, we need an interval with each side of size $\sqrt[p]{.10}$ of the total. This proportion gets close to 1 quickly, and if the proportion is 1 it means we include all the data and are no longer smoothing.

```
library(tidyverse)
p <- 1:100
qplot(p, .1^(1/p), ylim = c(0,1))
```



By the time we reach 100 predictors, the neighborhood is no longer very local, as each side covers almost the entire dataset.

Here we look at a set of elegant and versatile methods that adapt to higher dimensions and also allow these regions to take more complex shapes while still producing models that are interpretable. These are very popular, well-known and studied methods. We will concentrate on regression and decision trees and their extension to random forests.

31.10.2 CART motivation

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")
names(olive)
#>  [1] "region"      "area"        "palmitic"    "palmitoleic"
#>  [5] "stearic"     "oleic"       "linoleic"    "linolenic"
#>  [9] "arachidic"   "eicosenoic"
```

For illustrative purposes, we will try to predict the region using the fatty acid composition values as predictors.

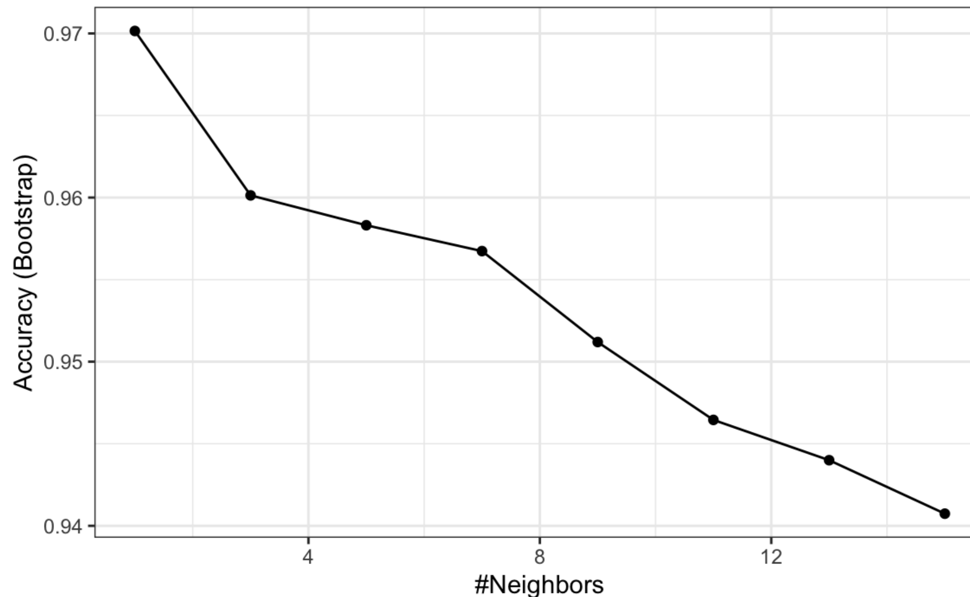
```
table(olive$region)
#> 
#> Northern Italy       Sardinia Southern Italy 
#>            151             98            323
```

We remove the `area` column because we won’t use it as a predictor.

```
olive <- select(olive, -area)
```

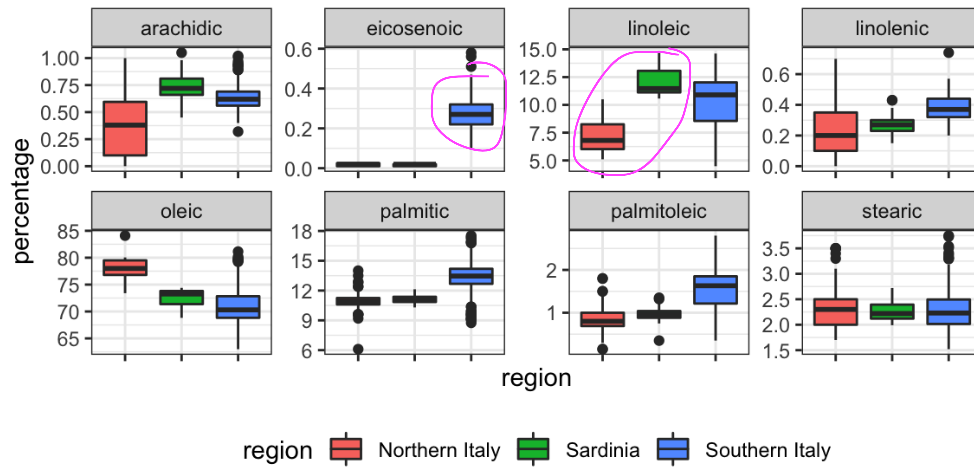
Let’s very quickly try to predict the region using kNN:

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



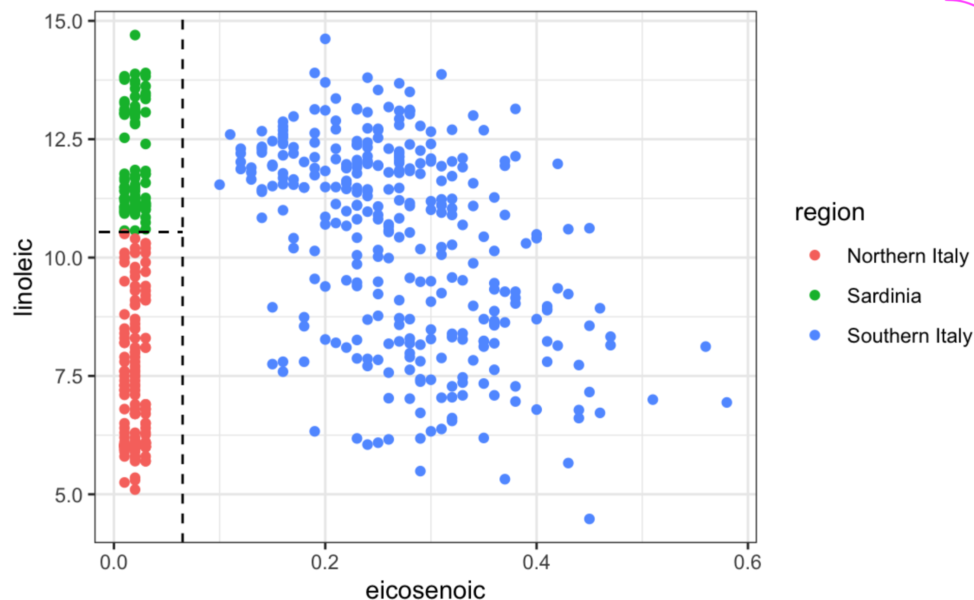
We see that using just one neighbor, we can predict relatively well. However, a bit of data exploration reveals that we should be able to do even better. For example, if we look at the distribution of each predictor stratified by region we see that eicosenoic is only present in Southern Italy and that linoleic separates Northern Italy from Sardinia.

```
olive %>% gather(fatty_acid, percentage, -region) %>%
  ggplot(aes(region, percentage, fill = region)) +
  geom_boxplot() +
  facet_wrap(~fatty_acid, scales = "free", ncol = 4) +
  theme(axis.text.x = element_blank(), legend.position="bottom")
```



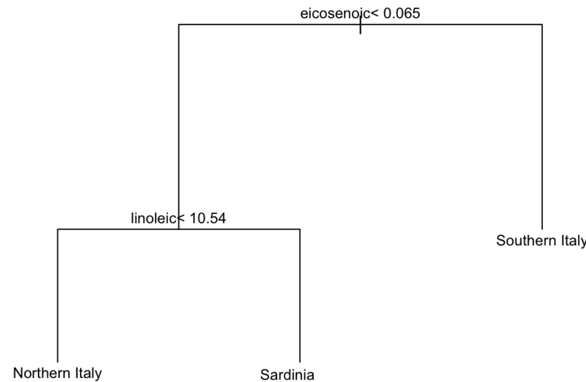
This implies that we should be able to build an algorithm that predicts perfectly! We can see this clearly by plotting the values for eicosenoic and linoleic.

```
olive %>%
  ggplot(aes(eicosenoic, linoleic, color = region)) +
  geom_point() +
  geom_vline(xintercept = 0.065, lty = 2) +
  geom_segment(x = -0.2, y = 10.54, xend = 0.065, yend = 10.54,
               color = "black", lty = 2)
```

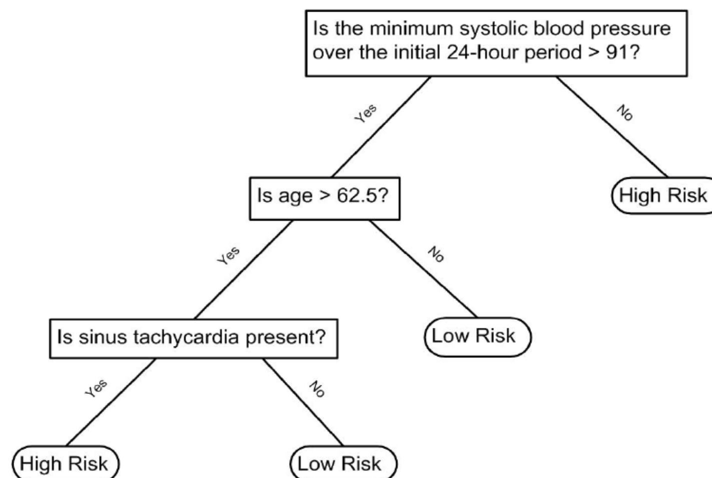


In Section 33.3.4 we define predictor spaces. The predictor space here consists of eight-dimensional points with values between 0 and 100. In the plot above, we show the space defined by the two predictors eicosenoic and linoleic, and, by eye, we can construct a prediction rule that partitions the predictor space so that each partition contains only outcomes of a one

category. This in turn can be used to define an algorithm with perfect accuracy. Specifically, we define the following decision rule. If eicosenoic is larger than 0.065, predict Southern Italy. If not, then if linoleic is larger than 10.535, predict Sardinia, and if lower, predict Northern Italy. We can draw this decision tree like this:



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



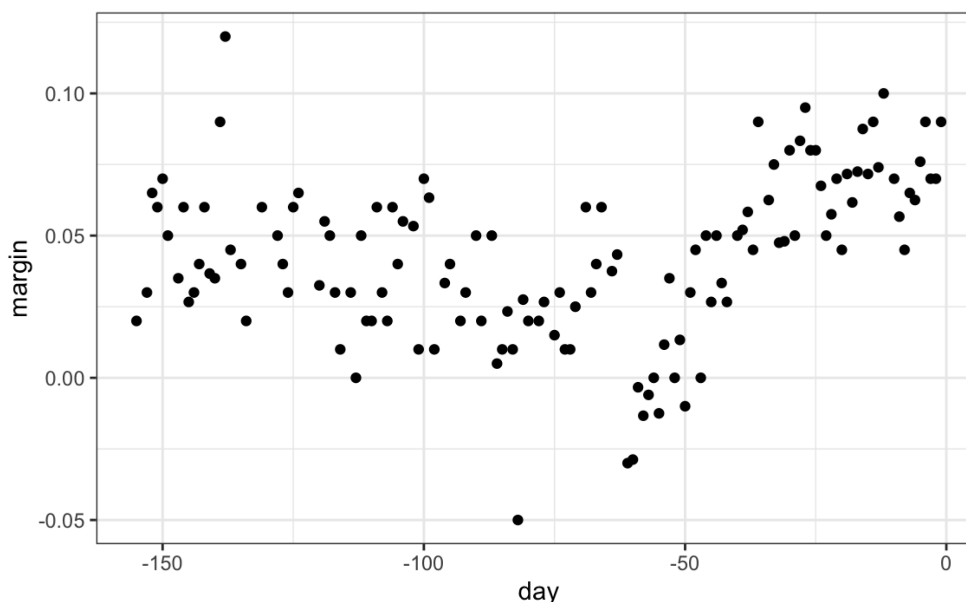
(Source: Walton 2010 Informal Logic, Vol. 30, No. 2, pp. 159-184[109].)

A tree is basically a flow chart of yes or no questions. The general idea of the methods we are describing is to define an algorithm that uses data to create these trees with predictions at the ends, referred to as *nodes*. Regression and decision trees operate by predicting an outcome variable Y by partitioning the predictors.

31.10.3 Regression trees

When the outcome is continuous, we call the method a *regression* tree. To introduce regression trees, we will use the 2008 poll data used in previous sections to describe the basic idea of how we build these algorithms. As with other machine learning algorithms, we will try to estimate the conditional expectation $f(x) = \mathrm{E}(Y|X = x)$ with Y the poll margin and x the day.

```
data("polls_2008")
qplot(day, margin, data = polls_2008)
```



The general idea here is to build a decision tree and, at the end of each *node*, obtain a predictor $\hat{y}$. A mathematical way to describe this is to say that we are partitioning the predictor space into J non-overlapping regions, $R_1, R_2, \dots, R_J$, 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.

But how do we decide on the partition $R_1, R_2, \dots, R_J$ and how do we choose J? Here is where the algorithm gets a bit complicated.

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]. But after the first step we will have two partitions. After the second step we will split one of these partitions into two and will have three partitions, then four, then five, and so on. We describe how we pick the partition to further partition, and when to stop, later.

Once we select a partition $\mathbf{x}$ to split in order to create the new partitions, we find a predictor j and value s that define two new partitions, which we will call $R_1(j,s)$ and $R_2(j,s)$, that split our observations in the current partition by asking if x_j is bigger than s:

$$R_1(j,s) = \{\mathbf{x} \mid x_j < s\} \text{ and } R_2(j,s) = \{\mathbf{x} \mid x_j \geq s\}$$

In our current example we only have one predictor, so we will always choose $j = 1$, but in general this will not be the case. Now, after we define the new partitions R_1 and R_2, and we decide to stop the partitioning, we compute predictors by taking the average of all the observations y for which the associated $\mathbf{x}$ is in R_1 and R_2. We refer to these two as $\hat{y}_{R_1}$ and $\hat{y}_{R_2}$ respectively.

But how do we pick j and s? Basically we find the pair that minimizes the residual sum of square (RSS):

$$\sum_{i:\, x_i \in R_1(j,s)} (y_i - \hat{y}_{R_1})^2 + \sum_{i:\, x_i \in R_2(j,s)} (y_i - \hat{y}_{R_2})^2$$

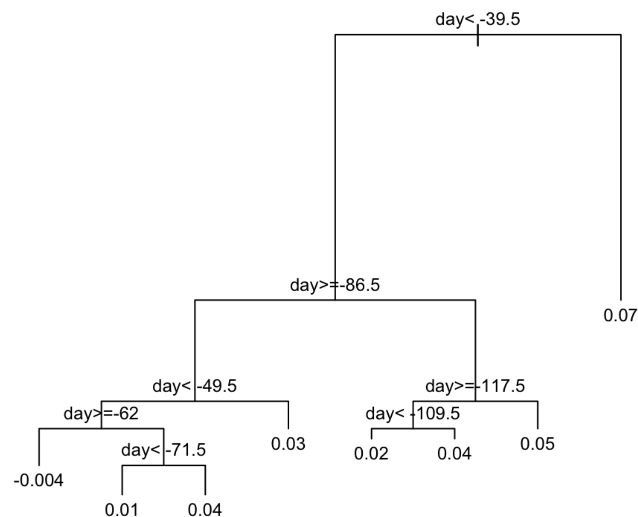
This is then applied recursively to the new regions R_1 and R_2. We describe how we stop later, but once we are done partitioning the predictor space into regions, in each region a prediction is made using the observations in that region.

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)
```

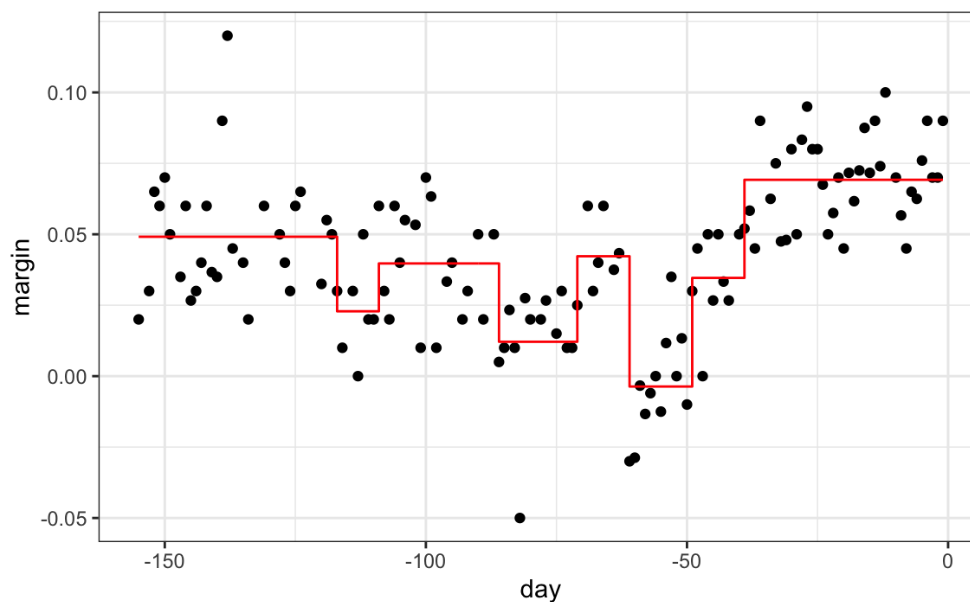
Here, there is only one predictor. Thus we do not have to decide which predictor j to split by, we simply have to decide what value s we use to split. We can visually see where the splits were made:

```
plot(fit, margin = 0.1)
text(fit, cex = 0.75)
```



The first split is made on day 39.5. One of those regions is then split at day 86.5. The two resulting new partitions are split on days 49.5 and 117.5, respectively, and so on. We end up with 8 partitions. 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")
```



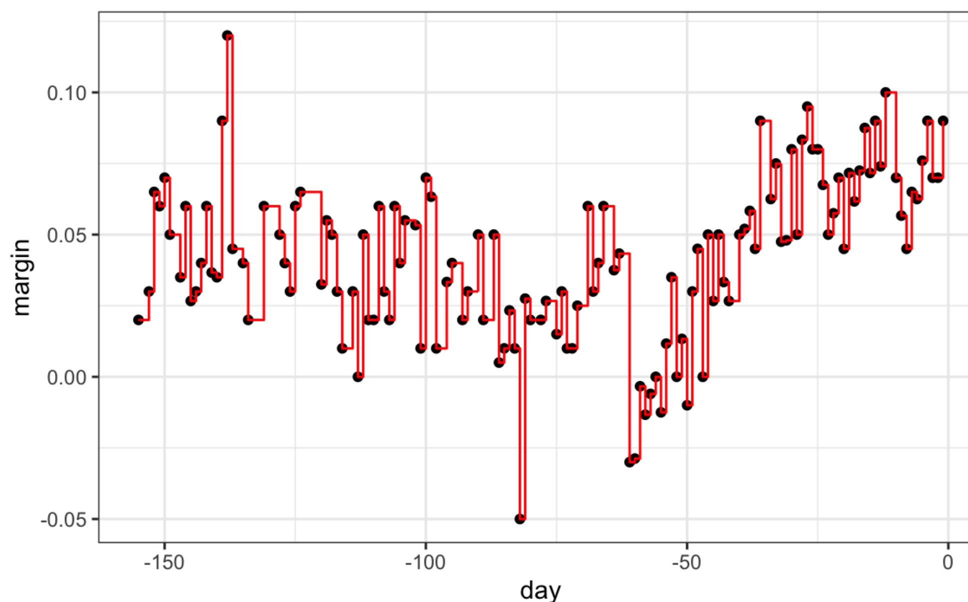
Note that the algorithm stopped partitioning at 8. Now we explain how this decision is made.

First we need to define the term *complexity parameter* (cp). Every time we split and define two new partitions, our training set RSS decreases. This is because with more partitions, 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 since the average of one value is that same value. 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). The RSS must improve by a factor of cp for the new partition to be added. Large values of cp will therefore force the algorithm to stop earlier which results in fewer nodes.

However, cp is not the only parameter used to decide if we should partition a current partition or not. Another common parameter is the minimum number of observations required in a partition before partitioning it further. 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)`.

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

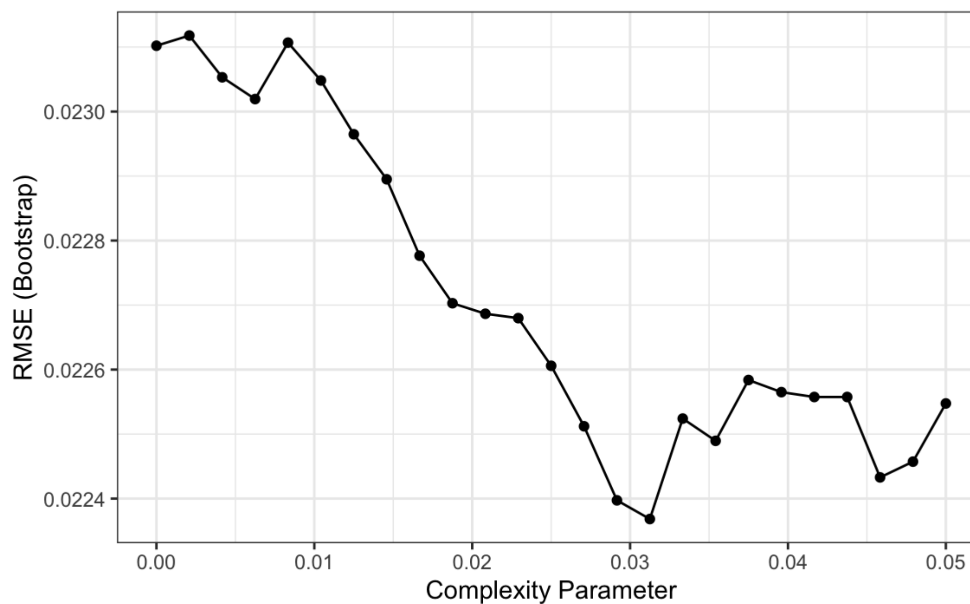
```
fit <- rpart(margin ~ ., data = polls_2008,
             control = rpart.control(cp = 0, minsplit = 2))
polls_2008 %>%
  mutate(y_hat = predict(fit)) %>%
  ggplot() +
  geom_point(aes(day, margin)) +
  geom_step(aes(day, y_hat), col="red")
```



Intuitively we know that this is not a good approach as it will generally result in over-training. These `cp` , `minsplit` , and `minbucket` , three parameters can be used to control the variability of the final predictors. The larger these values are the more data is averaged to compute a predictor and thus reduce variability. The drawback is that it restricts flexibility.

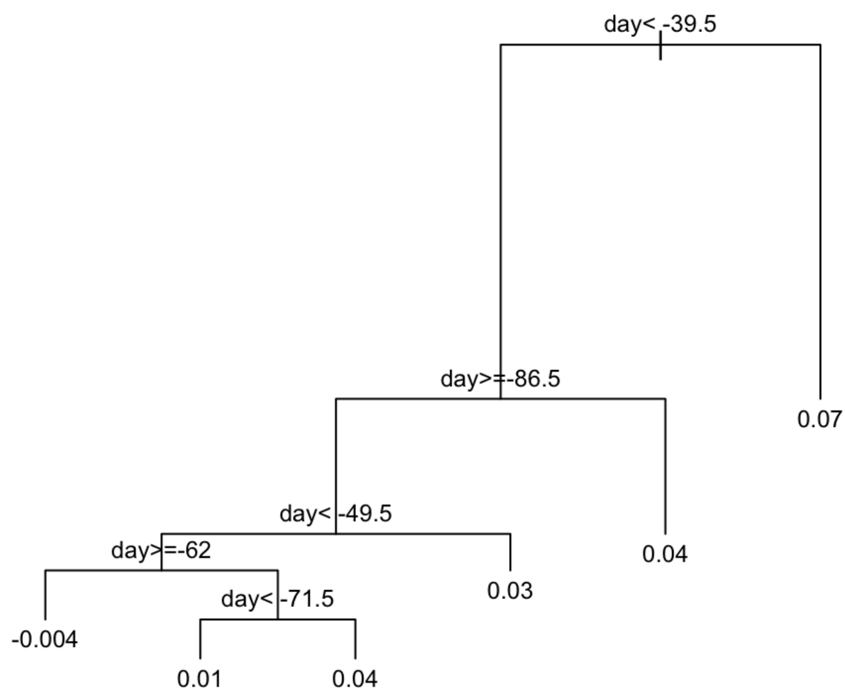
So how do we pick these parameters? We can use cross validation, described in Chapter 29, just like with any tuning parameter. Here is an example of using cross validation to chose cp.

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



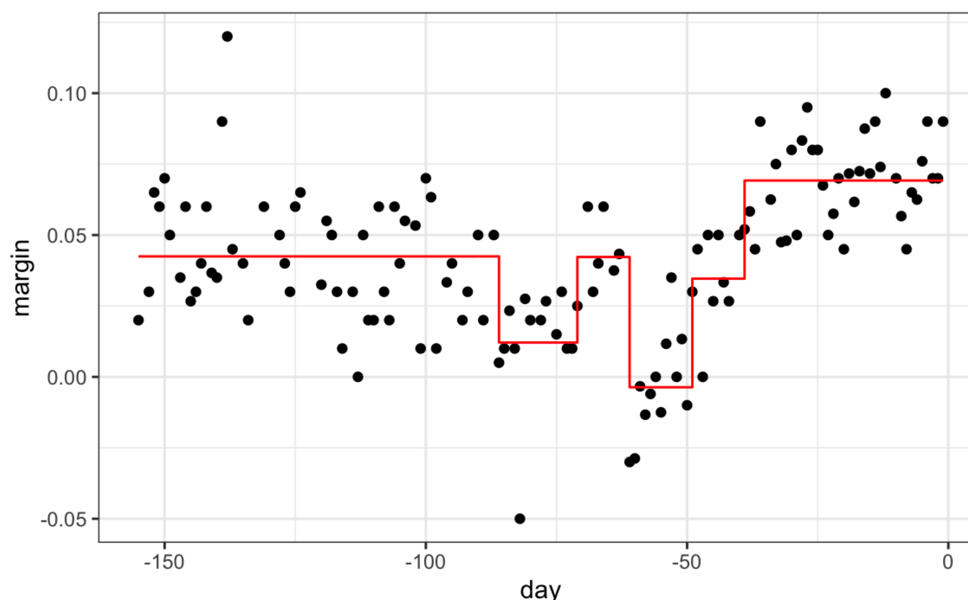
To see the resulting tree, we access the `finalModel` and plot it:

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



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")
```



Note that if we already have a tree and want to apply a higher cp value, we can use the `prune` function. We call this *pruning* a tree because we are snipping off partitions that do not meet a `cp` criterion. We previously created a tree that used a `cp = 0` and saved it to `fit`. We can prune it like this:

```
pruned_fit <- prune(fit, cp = 0.01)
```

31.10.4 Classification (decision) trees

Classification trees, or decision trees, are used in prediction problems where the outcome is categorical. We use the same partitioning principle with some differences to account for the fact that we are now working with a categorical outcome.

The first difference is that 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).

The second is that we can no longer use RSS to choose the partition. While we could use the naive approach of looking for partitions that minimize training error, better performing approaches use more sophisticated metrics. Two of the more popular ones are the *Gini Index* and *Entropy*.

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* is going to be 0 in this scenario, and become larger the more we deviate from this scenario. 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

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

If you study the formula carefully you will see that it is in fact 0 in the perfect scenario described above.

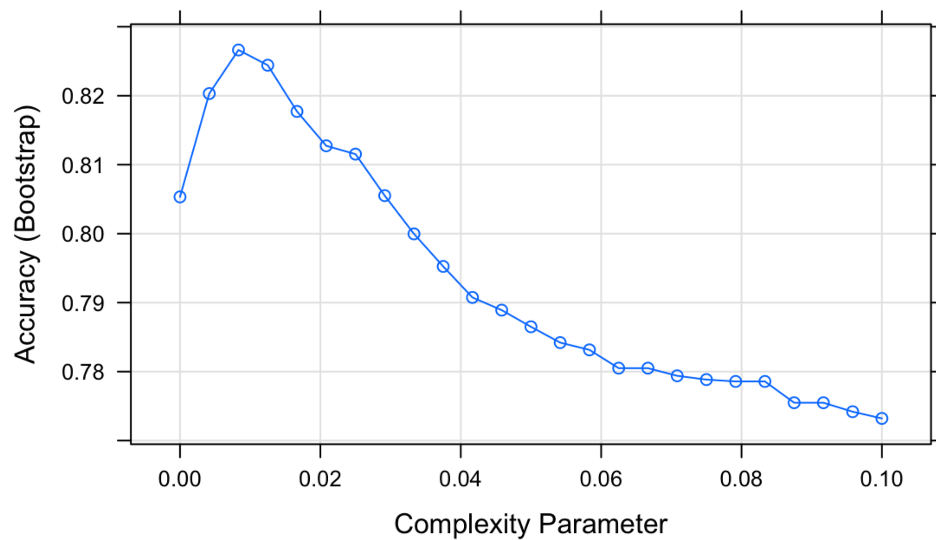
Entropy is a very similar quantity, defined as

$$\text{entropy}(j) = -\sum_{k=1}^{K} \hat{p}_{j,k} \log(\hat{p}_{j,k}), \text{ with } 0 \times \log(0) \text{ defined as } 0$$

Let us look at how a classification tree performs on the digits example we examined before:

We can use this code to run the algorithm and plot the resulting tree:

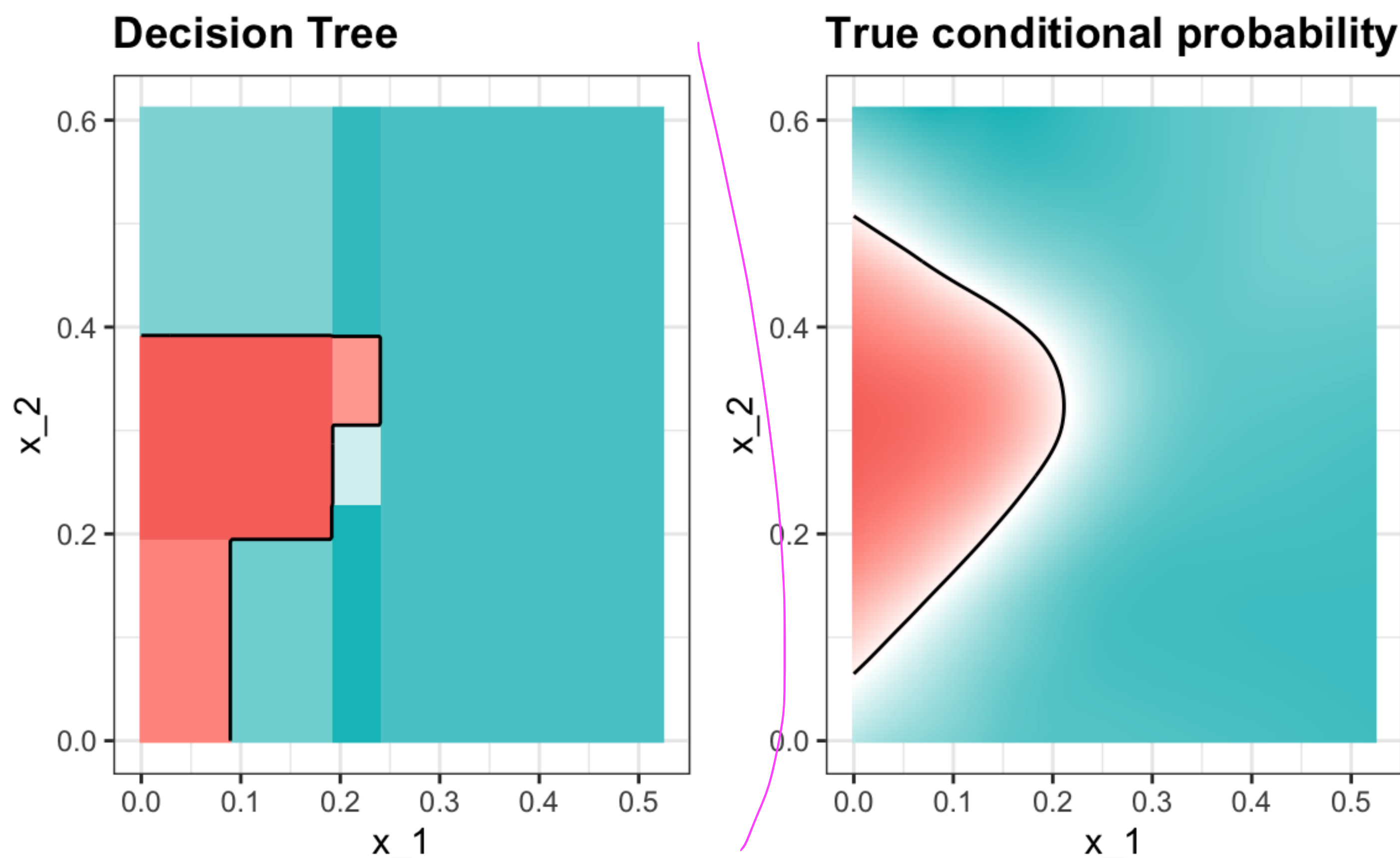
```
train_rpart <- train(y ~ .,
                     method = "rpart",
                     tuneGrid = data.frame(cp = seq(0.0, 0.1, len = 25)),
                     data = mnist_27$train)
plot(train_rpart)
```



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
#>     0.82
```

The plot of the estimated conditional probability shows us the limitations of classification trees:



Note that with decision trees, it is difficult to make the boundaries smooth since each partition creates a discontinuity.

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). Finally, 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. Furthermore, 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.

31.11 Random forests

Random forests are a **very popular** machine learning approach that addresses the shortcomings of decision trees using a clever idea. The goal is to improve prediction performance and reduce instability by *averaging* multiple decision trees (a forest of trees constructed with randomness). It has two features that help accomplish this.

The first step is *bootstrap aggregation* or *bagging*. The general idea is to generate many predictors, each using regression or classification trees, and then forming a final prediction based on the average prediction of all these trees. To assure that the individual trees are not the same, we use the bootstrap to induce randomness. These two features combined explain the name: the bootstrap makes the individual trees **randomly** different, and the combination of trees is the **forest**. The specific steps are as follows.

1. Build B decision trees using the training set. We refer to the fitted models as $T_1, T_2, \dots, T_B$. 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 $\hat{y}_1, \dots, \hat{y}_T$).

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

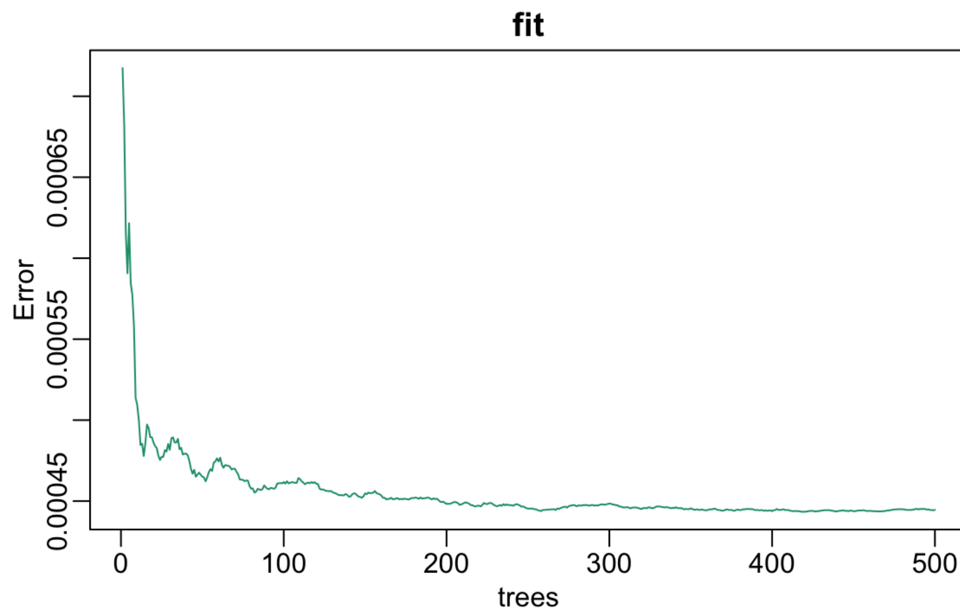
2. A large number of features is typical in machine learning challenges. Often, many features can be informative but including them all in the model may result in overfitting. The second way random forests induce randomness is by randomly selecting features to be included in the building of each tree. A different random subset is selected for each tree. This reduces correlation between trees in the forest, thereby improving prediction accuracy.

To illustrate how the first steps can result in smoother estimates we will demonstrate by fitting a random forest to the 2008 polls data. We will use the `randomForest` function in the **randomForest** package:

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

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

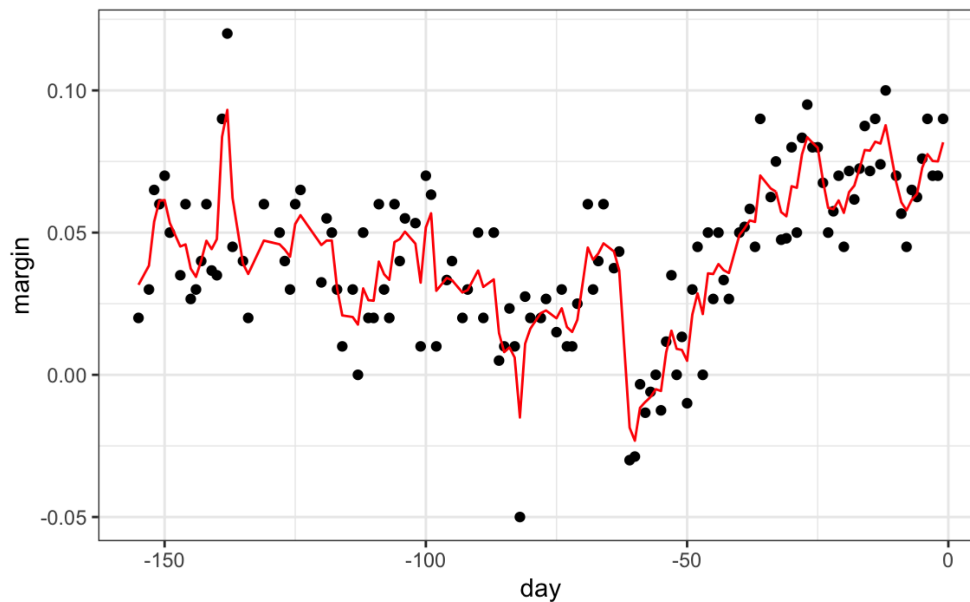
```
rafalib::mypar()
plot(fit)
```



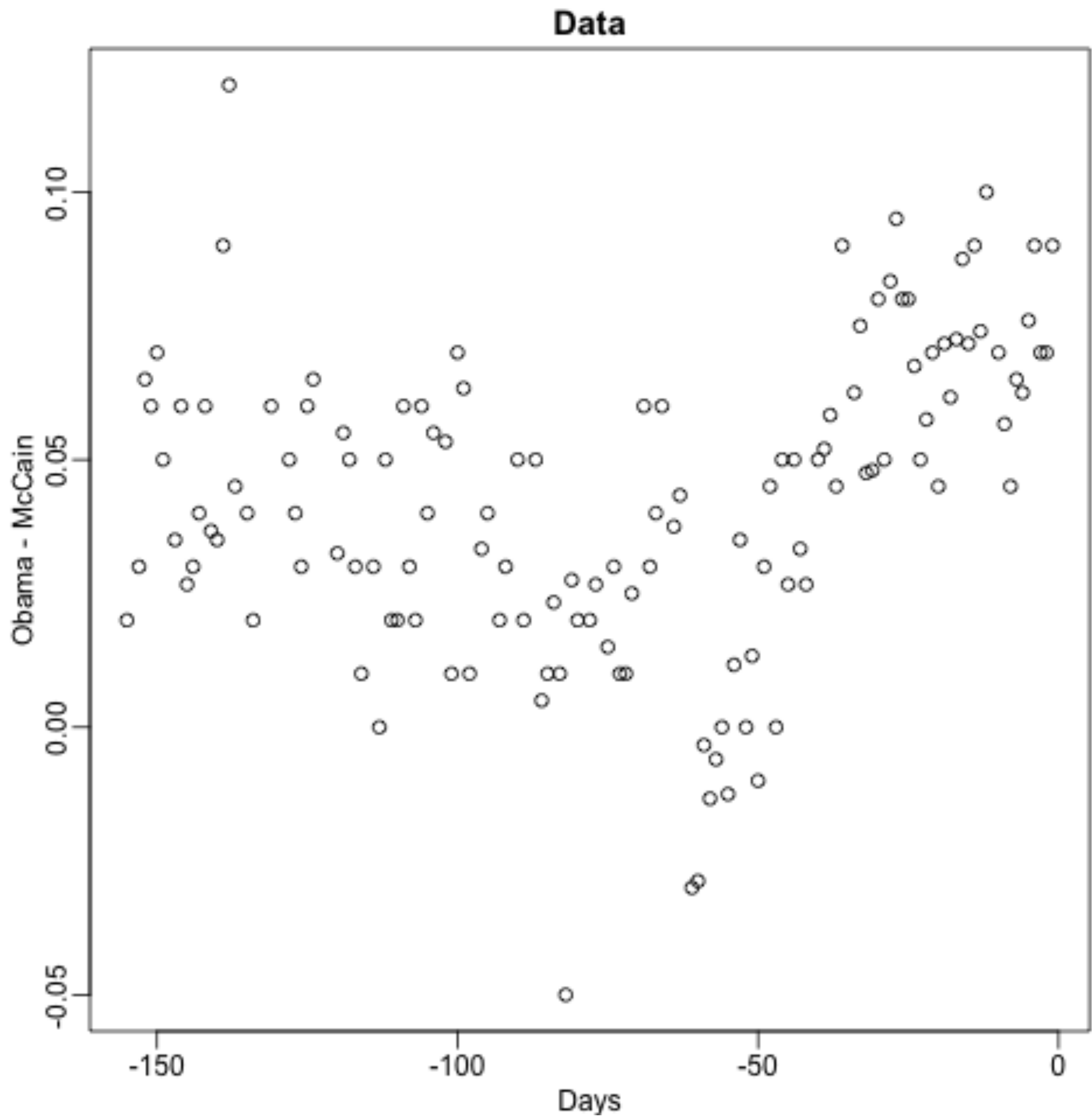
We can see that in this case, the accuracy improves as we add more trees until about 30 trees where accuracy stabilizes.

The resulting estimate for this random forest can be seen 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")
```



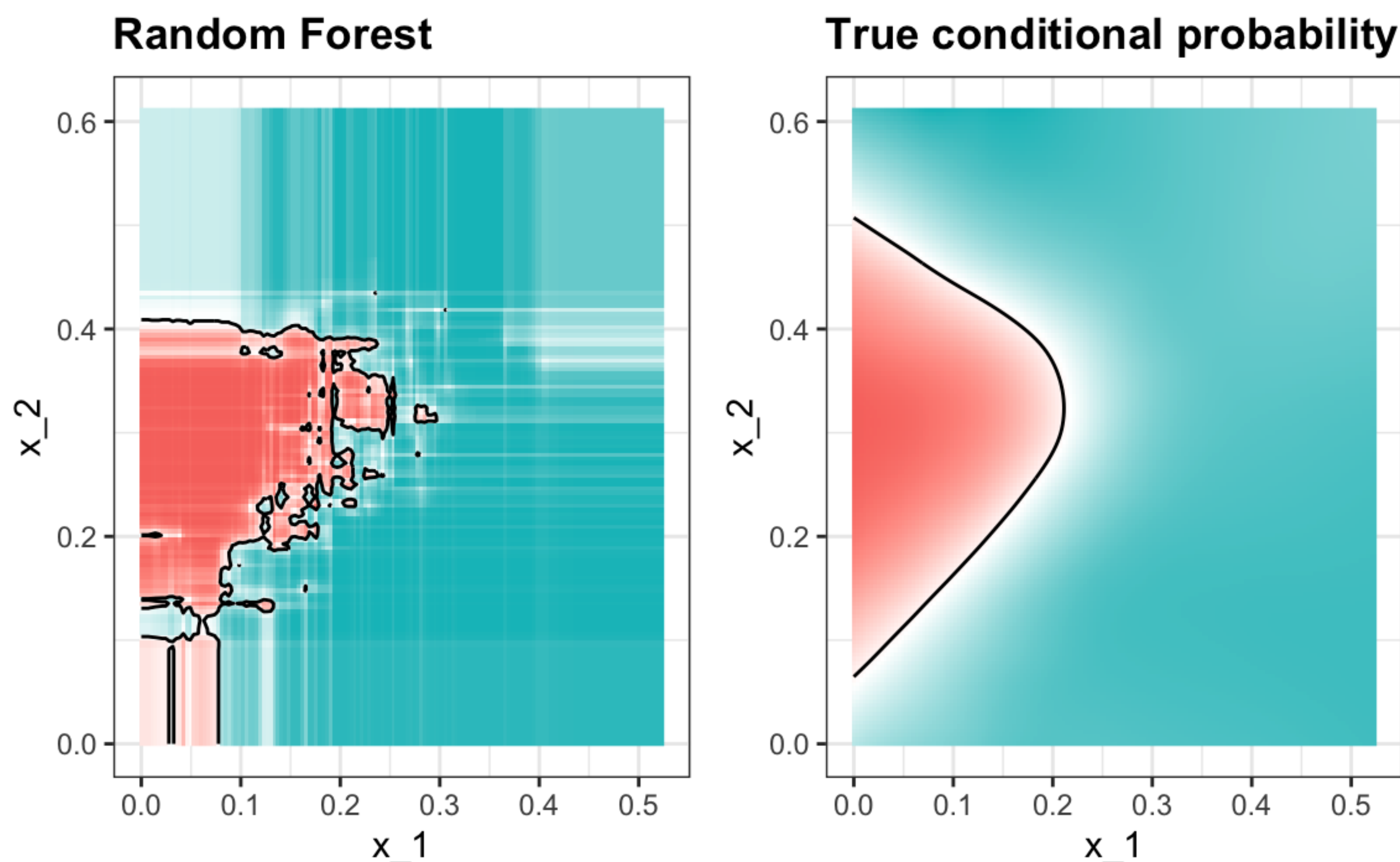
Notice that the random forest estimate is much smoother than what we achieved with the regression tree in the previous section. This is possible because the average of many step functions can be smooth. We can see this by visually examining how the estimate changes as we add more trees. In the following figure you see each of the bootstrap samples for several values of b and for each one we see the tree that is fitted in grey, the previous trees that were fitted in lighter grey, and the result of averaging all the trees estimated up to that point.



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

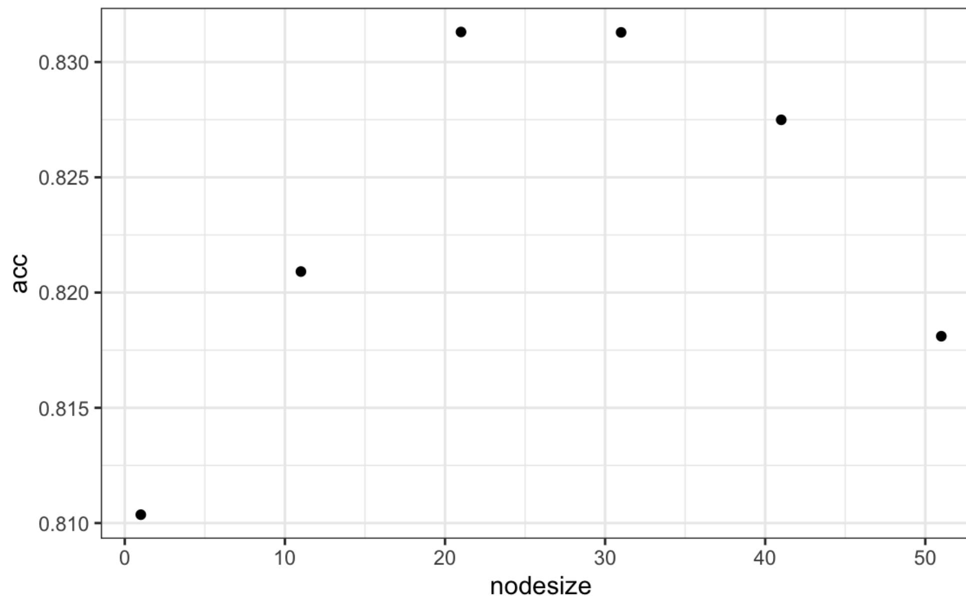
```
library(randomForest)
train_rf <- randomForest(y ~ ., data=mnist_27$train)

confusionMatrix(predict(train_rf, mnist_27$test),
                mnist_27$test$y)$overall["Accuracy"]
#> Accuracy
#>     0.79
```

Here is what the conditional probabilities look like:



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. Below, we use the **caret** package to optimize over the minimum node size. Because, this is not one of the parameters that the **caret** package optimizes by default we will write our own code:

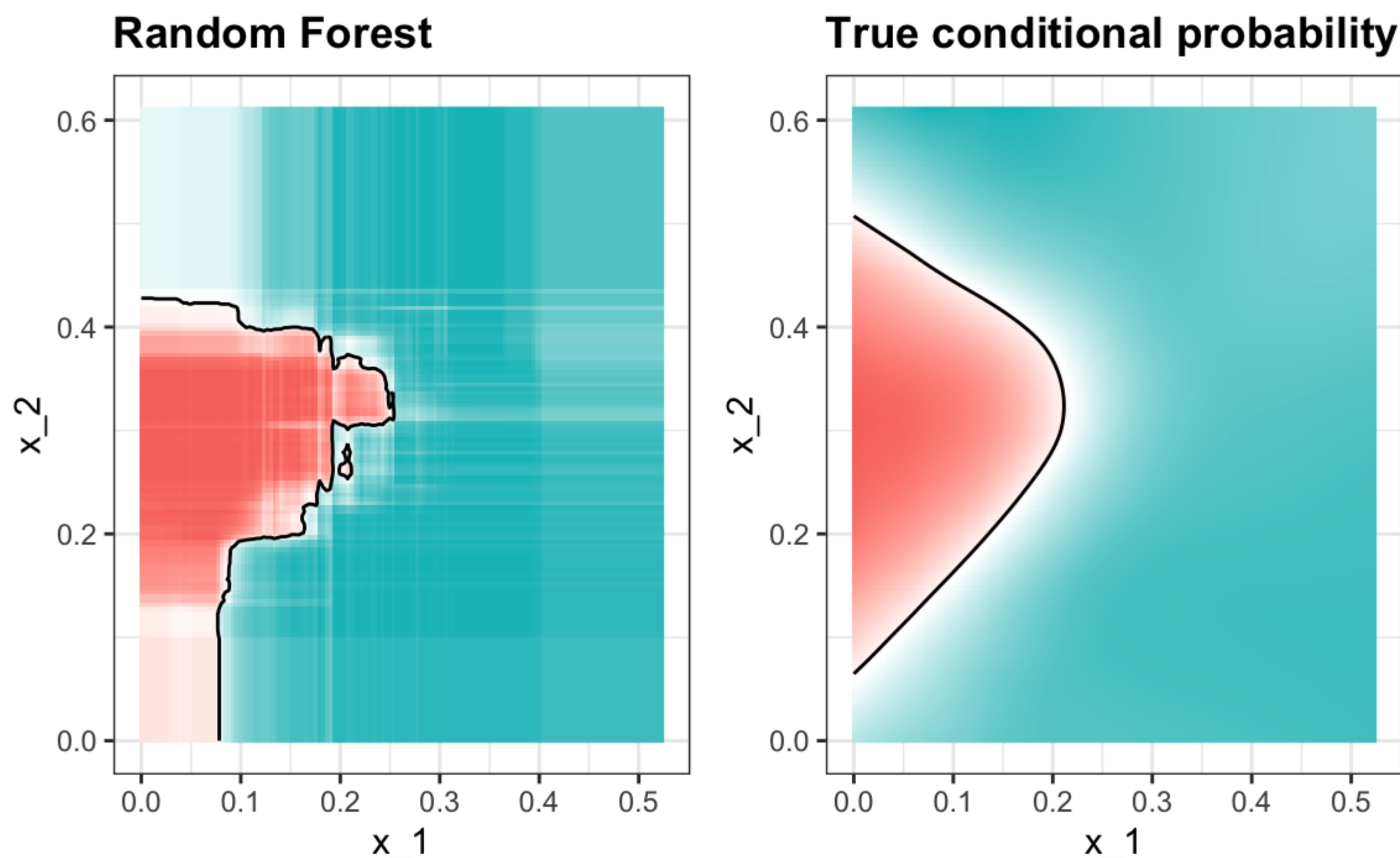
```
nodesize <- seq(1, 51, 10)
acc <- sapply(nodesize, function(ns){
  train(y ~ ., method = "rf", data = mnist_27$train,
             tuneGrid = data.frame(mtry = 2),
             nodesize = ns)$results$Accuracy
})
qplot(nodesize, acc)
```



We can now fit the random forest with the optimized minimun 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
#>    0.815
```

The selected model improves accuracy and provides a smoother estimate.



Note that we can avoid writing our own code by using other random forest implementations as described in the **caret** manual[110].

Random forest performs better in all the examples we have considered. 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 *variable importance* in an advanced machine learning book[111]. The **caret** package includes the function `varImp` that extracts variable importance from any model in which the calculation is implemented. We give an example on how we use variable importance in the next section.

31.12 Exercises

1. Create a simple dataset where the outcome grows 0.75 units on average for every increase in a predictor:

```
n <- 1000
sigma <- 0.25
x <- rnorm(n, 0, 1)
y <- 0.75 * x + rnorm(n, 0, sigma)
dat <- data.frame(x = x, y = y)
```

Use `rpart` to fit a regression tree and save the result to `fit` .

2. Plot the final tree so that you can see where the partitions occurred.

3. Make a scatterplot of `y` versus `x` along with the predicted values based on the fit.

4. Now model with a random forest instead of a regression tree using `randomForest` from the **randomForest** package, and remake the scatterplot with the prediction line.

5. Use the function `plot` to see if the random forest has converged or if we need more trees.

6. It seems that the default values for the random forest result in an estimate that is too flexible (not smooth). Re-run the random forest but this time with `nodesize` set at 50 and `maxnodes` set at 25. Remake the plot.

7. We see that this yields smoother results. Let's use the `train` function to help us pick these values. From the **caret** manual[112] we see that we can't tune the `maxnodes` parameter or the `nodesize` argument with `randomForest` , so we will use the **Rborist** package and tune the `minNode` argument. Use the `train` function to try values `minNode <- seq(5, 250, 25)` . See which value minimizes the estimated RMSE.

8. Make a scatterplot along with the prediction from the best fitted model.

9. Use the `rpart` function to fit a classification tree to the `tissue_gene_expression` dataset. Use the `train` function to estimate the accuracy. Try out `cp` values of `seq(0, 0.05, 0.01)` . Plot the accuracy to report the results of the best model.

10. Study the confusion matrix for the best fitting classification tree. What do you observe happening for placenta?

11. Notice that placentas are called endometrium more often than placenta. Note also that the number of placentas is just six, and that, by default, `rpart` requires 20 observations before splitting a node. Thus it is not possible with these parameters to have a node in which placentas

are the majority. Rerun the above analysis but this time permit `rpart` to split any node by using the argument `control = rpart.control(minsplit = 0)` . Does the accuracy increase? Look at the confusion matrix again.

12. Plot the tree from the best fitting model obtained in exercise 11.

13. We can see that with just six genes, we are able to predict the tissue type. Now let's see if we can do even better with a random forest. Use the `train` function and the `rf` method to train a random forest. Try out values of `mtry` ranging from, at least, `seq(50, 200, 25)` . What `mtry` value maximizes accuracy? To permit small `nodesize` to grow as we did with the classification trees, use the following argument: `nodesize = 1` . This will take several seconds to run. If you want to test it out, try using smaller values with `ntree` . Set the seed to 1990.

14. Use the function `varImp` on the output of `train` and save it to an object called `imp` .

15. The `rpart` model we ran above produced a tree that used just six predictors. Extracting the predictor names is not straightforward, but can be done. If the output of the call to train was `fit_rpart` , we can extract the names like this:

```
ind <- !(fit_rpart$finalModel$frame$var == "<leaf>")
tree_terms <-
  fit_rpart$finalModel$frame$var[ind] %>%
  unique() %>%
  as.character()
tree_terms
```

What is the variable importance in the random forest call for these predictors? Where do they rank?

16. Advanced: Extract the top 50 predictors based on importance, take a subset of `x` with just these predictors and apply the function `heatmap` to see how these genes behave across the tissues. We will introduce the `heatmap` function in Chapter 34.

107. http://www.amazon.com/Mathematical-Statistics-Analysis-Available-Enhanced/dp/0534399428↩

108. https://web.stanford.edu/~hastie/Papers/ESLII.pdf↩

109. https://papers.ssrn.com/sol3/Delivery.cfm/SSRN_ID1759289_code1486039.pdf?abstractid=1759289&mirid=1&type=2↩

110. http://topepo.github.io/caret/available-models.html↩

111. https://web.stanford.edu/~hastie/Papers/ESLII.pdf↩

112. https://topepo.github.io/caret/available-models.html↩