Machine Learning Day 1

Dr. Osita Onyejekwe

10/21/2022

Machine Learning

Introduction

Perhaps the most popular data science methodologies come from *Machine Learning*. Machine learning success stories include the hand writing zip code readers implemented by the postal service, speech recognition such as Apple's Siri, movie recommendation systems, spam and malware detectors, housing prices, stock market outcomes, and driver-less cars. While artificial intelligence algorithms, such as those used by chess playing machines, implement decision making based on programmable rules derived from theory or first principles, in Machine Learning decisions are based on algorithms built on data.

Notation

Data comes in the form of

1. the ***outcome*** we want to predict and
2. the ***features*** that we will use to predict the outcome.

We want to build an algorithm that takes feature values as input and returns a prediction for the outcome when we don't know it. The machine learning approach is to use a dataset for which we do know the outcome to ***train*** the algorithm for future use.

Here we will use Y to denote the outcome and $X_1, \dots, X_p$ to denote features. **Features are sometimes referred to as predictors or covariates**.

```
#install.packages("BiocManager")
#BiocManager::install("EBImage")
library(EBImage)
```

```
##
## Attaching package: 'EBImage'

## The following object is masked from 'package:purrr':
##
##     transpose
```

```r
#img = readImage("/Users/osita/Desktop/supervised_ml.png")
#display(img, method = "raster")
```

```r
#BiocManager::install(c("GenomicFeatures", "AnnotationDbi"))
```

Prediction problems can be divided into categorical and continuous outcomes. For categorical outcomes, the outcome Y can be one of K classes. The number of classes can vary greatly across applications. For example, for handwritten digits, $K = 10$ with the classes being the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. In speech recognition, the outcome is all possible words we are trying to detect. Spam detection has two outcomes: spam or not spam. Here we denote the K categories with indexes $k = 1, \ldots, K$. However, for binary data we will use $k = 0, 1$.

The general setup is as follows. We have a series of features and an unknown outcome we want to predict:

outcome	feature_1	feature_2	feature_3	feature_4	feature_5
?	X_1	X_2	X_3	X_4	X_5

To *build a model* that provides a prediction for any set of values $X_1 = x_1, X_2 = x_2, \ldots, X_5 = x_5$, we collect data for which we know the outcome.

outcome	feature_1	feature_2	feature_3	feature_4	feature_5
Y1	X_1,1	X_1,2	X_1,3	X_1,4	X_1,5
Y2	X_2,1	X_2,2	X_2,3	X_2,4	X_2,5
Y3	X_3,1	X_3,2	X_3,3	X_3,4	X_3,5
Y4	X_4,1	X_4,2	X_4,3	X_4,4	X_4,5
Y5	X_5,1	X_5,2	X_5,3	X_5,4	X_5,5
Y6	X_6,1	X_6,2	X_6,3	X_6,4	X_6,5
Y7	X_7,1	X_7,2	X_7,3	X_7,4	X_7,5
Y8	X_8,1	X_8,2	X_8,3	X_8,4	X_8,5
Y9	X_9,1	X_9,2	X_9,3	X_9,4	X_9,5
Y10	X_10,1	X_10,2	X_10,3	X_10,4	X_10,5

We use the notation $\hat{Y}$ to denote the prediction. We use the term *actual outcome* to denote what we ended up observing. So we want the prediction $\hat{Y}$ to match the *actual outcome*.

Categorical versus Continuous

The outcome Y can be categorical (which digit, what word, spam or not spam, pedestrian or empty road ahead) or continuous (movie ratings, housing prices, stock value, distance to pedestrian). The concepts and algorithms we learn here apply to both. However, there are some differences in how we approach each case so it is important to distinguish between the two.

When the outcome is categorical we refer to the task as ***classification***. Our predictions will be categorical just like our outcomes and they will be either correct or incorrect. When the outcome is continuous we will refer to the task as ***prediction***. In this case our predictions will not be either right or wrong but some distance away from the actual outcome. This term can be confusing since we call $\hat{Y}$ our prediction even when it is a categorical outcome. However, throughout the lecture, the context will make the meaning clear.

Note that these terms vary among courses, textbooks, and other publications. Often *prediction* is used for both categorical and continuous and *regression* is used for the continuous case. Here we avoid using *regression* to avoid confusion with our previous use of the term *linear regression*. In most cases it will be clear if our outcomes are categorical or continuous so we will avoid using these terms when possible.

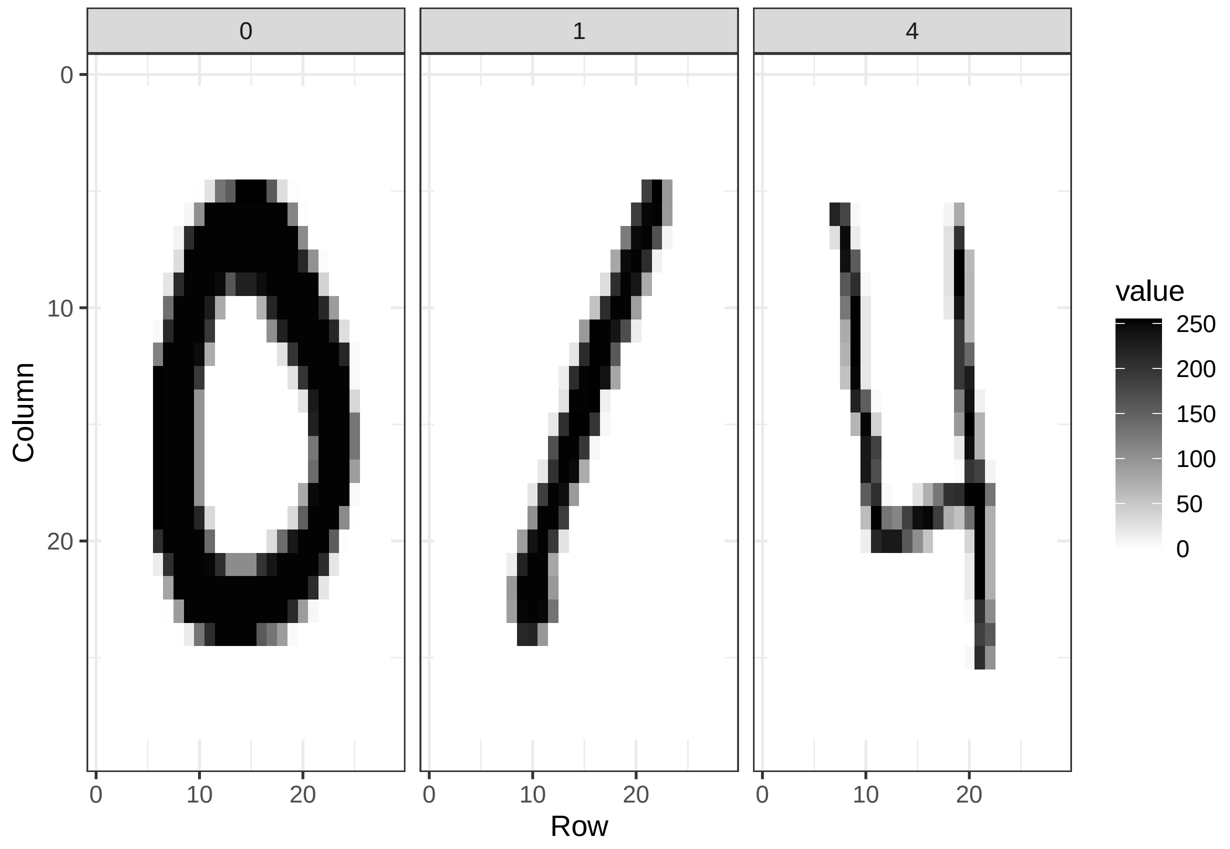
The first part of this module deals with categorical values and the second with continuous ones.

Case Study 1: Digit reader:

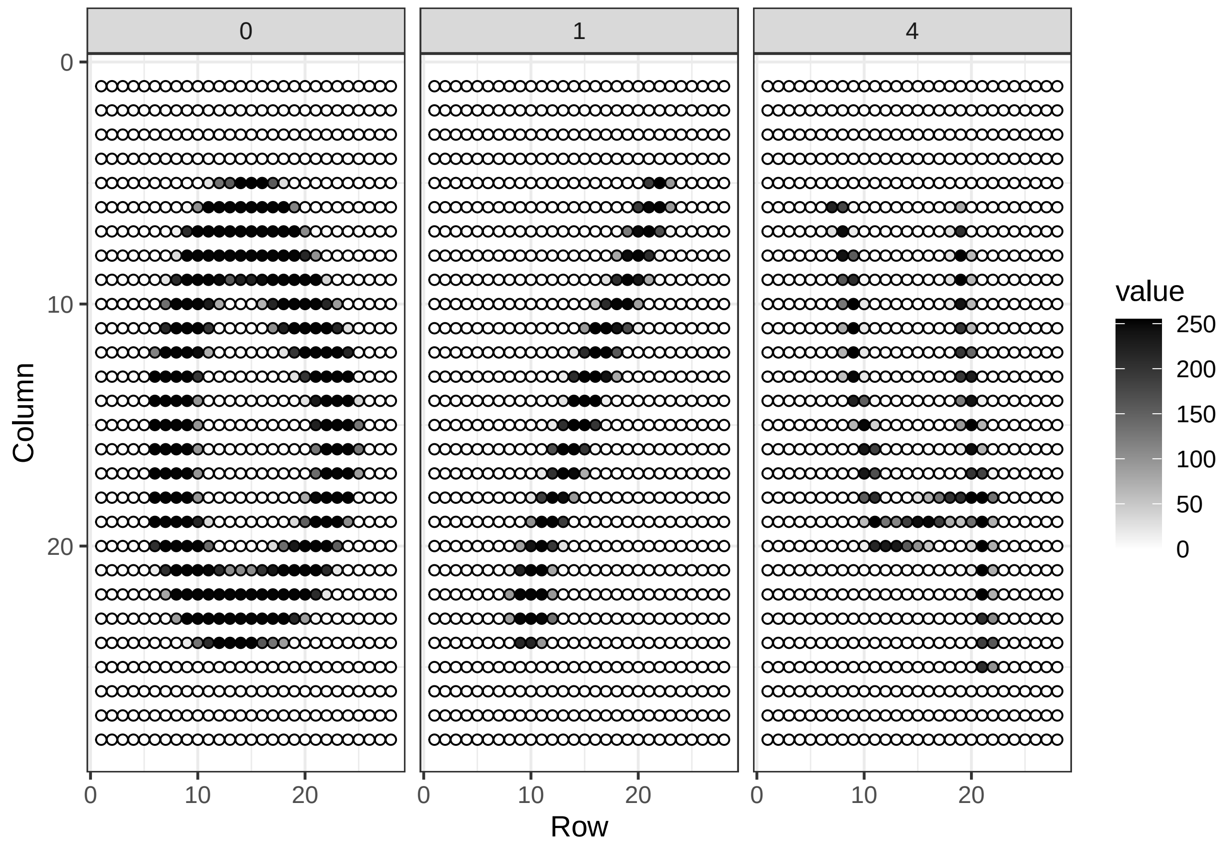
Let's consider an example. The first thing that happens to a letter when they are received in the post office is that they are sorted by zip code:

Originally humans had to sort these by hand. To do this they had to read the zip codes on each letter. Today thanks to machine learning algorithms, a computer can read zip codes and then a robot sorts the letters. In this lecture we will learn how to build algorithms that can read a digit.

The first step in building an algorithm is to understand what the outcomes and features are. Below are three images of written digits. These have already been read by a human and assigned an outcome y. These are considered known and serve as the **training set**.



The images are converted into $28 \times 28 = 784$ pixels and for each pixel we obtain a grey scale intensity between 0 (white) and 255 (black) which we consider continuous for now. We can see these values like this:



For each digit i we have a categorical outcome Y_i which can be one of 10 values: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$ and features $X_{i,1}, \dots, X_{i,784}$. We use bold face $\mathbf{X}_i = (X_{i,1}, \dots, X_{i,784})$ to distinguish the vector from the individual predictors. When referring to an arbitrary set of features we drop the index i and use Y and $\mathbf{X} = (X_1, \dots, X_{784})$. We use upper case variables because, in general, we think of the predictors as random variables. We use lower case, for example $\mathbf{X} = \mathbf{x}$, to denote observed values.

The machine learning task is to build an algorithm that returns a prediction for any of the possible values of the features. Here we will learn several approaches to building these algorithms. Although at this point it might seem impossible to achieve this, we will start with a simpler example, and build up our knowledge until we can attack this more complex example.

Accuracy and the confusion matrix

Here we consider a prediction task based on the height data.

```
library(dslabs)
data(heights)
```

We want to predict sex based on height. It is not a realistic example but rather one we use as an illustration that will help us start to understand the main concepts. We start by defining the outcome and predictor. In this example we have only one predictor.

```
y <- heights$sex
x <- heights$height
```

This is clearly a categorical outcome since Y can be `Male` or `Female`. Predicting Y based on X will be a hard task because male and female heights are not that different relative to within group variability. But can we do better than guessing? We can code a random guess like this:

```
set.seed(1)
N <- length(y)
y_hat <- sample(c("Male" , "Female"), N, replace = TRUE)
```

First, we will quantify what it means to do better. **The confusion matrix breaks down the correct and incorrect classifications**:

```
table(predicted = y_hat, actual = y)
```

```
##          actual
## predicted Female Male
##    Female    104  421
##    Male      134  391
```

The *accuracy* is simply defined as the overall proportion that is predicted correctly:

```
mean(y_hat == y)
```

```
## [1] 0.471
```

Not surprisingly, by guessing, our accuracy is about 50%. No matter the actual sex, we guess female half the time. Can we do better? We know males are slightly taller,

```
heights %>% group_by(sex) %>% summarize(mean(height), sd(height))
```

```
## # A tibble: 2 x 3
##   sex    `mean(height)` `sd(height)`
##   <fct>           <dbl>        <dbl>
## 1 Female           64.9         3.76
## 2 Male             69.3         3.61
```

so let's try to use our covariate. Let's try the approach "predict `Male` if height is within two standard deviations from the average male" (69.31 - 2(3.61)) $\approx$ 62:

```
y_hat <- ifelse( x >= 62, "Male", "Female")
```

The accuracy goes way up from 0.50 now:

```
mean(y == y_hat)
```

```
## [1] 0.779
```

But can we do better? We can examine the accuracy obtained for other cutoffs and then pick the value that provides the best results. However, if this optimization feels a bit like cheating to you, you are correct. By assessing our approach on the same dataset that we use to optimize our algorithm, we will end up with an over-optimistic view of our algorithm. In a later section we cover this issue, referred to as ***overtraining*** (or ***overfitting***) in more detail.

Training and test sets

The general solution to this problem is to split the data into **training and testing sets.** We build the algorithm using the training data and test the algorithm on the test set. In a later section we will learn more systematic ways to do this, but here we will split the data in half.

We now introduce the `caret` package that has several useful functions for building and assessing machine learning methods. For example, the `createDataPartition` automatically generates indexes. The argument `times` is used to define how many random samples of indexes to return, the argument `p` is used to define what proportion of the data to index, and the argument `list` is used to decide if we want the indexes returned as a list or not.

```r
library(caret)
set.seed(1)
train_index <- createDataPartition(y, times = 1, p =0.5, list = FALSE)
head(train_index)
```

```
##      Resample1
## [1,]         3
## [2,]         4
## [3,]         5
## [4,]         6
## [5,]        10
## [6,]        15
```

We can use this index to define the training and test sets:

```r
train_set <- heights[train_index,]
test_set <- heights[-train_index,]
head(train_set)
```

```
##       sex height
## 3    Male     68
## 4    Male     74
## 5    Male     61
## 6  Female     65
## 10   Male     67
## 15   Male     69
```

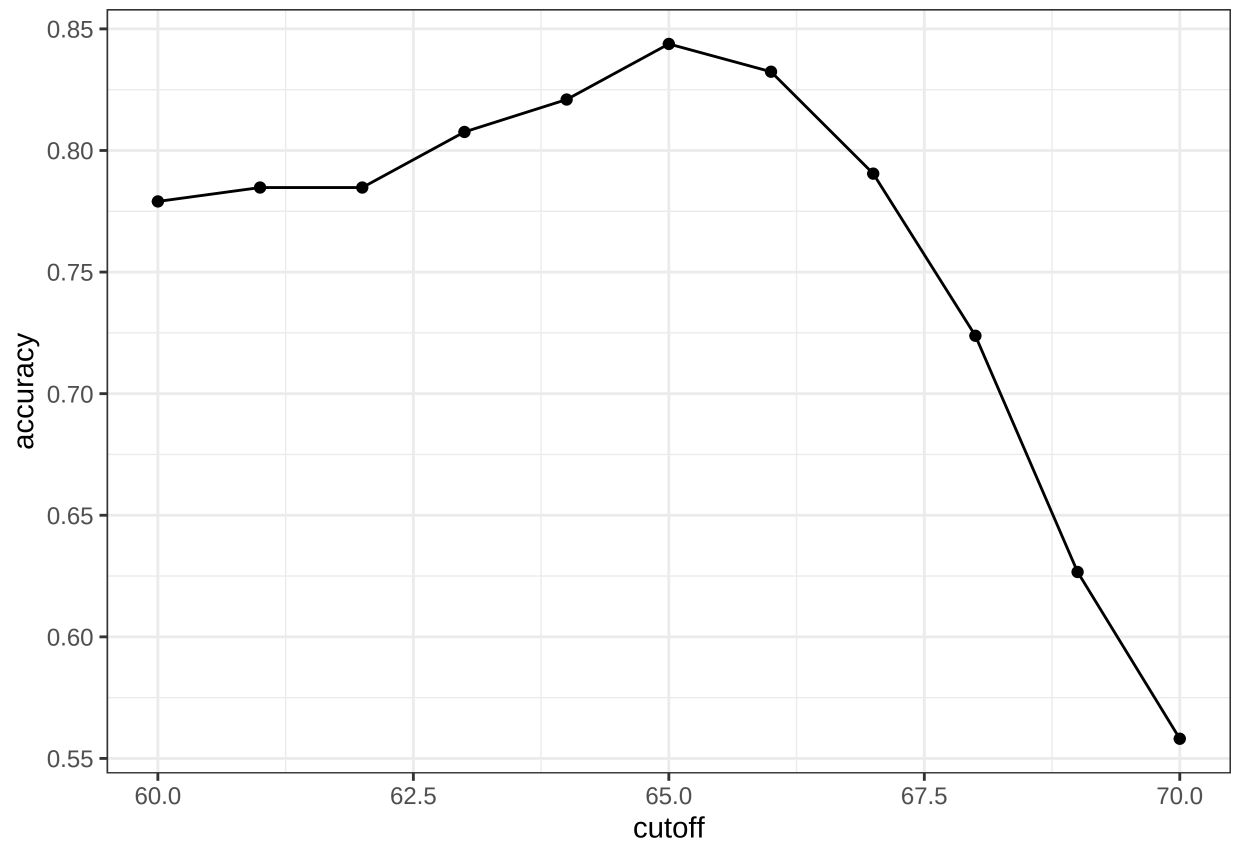
```r
head(test_set)
```

```
##       sex height
## 1    Male     75
## 2    Male     70
## 7  Female     66
## 8  Female     62
## 9  Female     66
## 11   Male     72
```

Now let's use the train set to examine the accuracy of 11 different cutoffs:

```
cutoff <- seq(60,70)
accuracy <- map_dbl(cutoff, function(x){
  y_hat <- ifelse(train_set$height >= x, "Male", "Female")
  mean(y_hat == train_set$sex)
  })
```

We can make a plot showing the accuracy on the training set for males and females.



We see that the maximum value is:

```
max(accuracy)
```

```
## [1] 0.844
```

much higher than 0.5, and it is maximized with the cutoff:

```
best_cutoff <- cutoff[which.max(accuracy)]
best_cutoff
```

```
## [1] 65
```

Now we can test this cutoff on our test set to make sure our accuracy is not overly optimistic:

```
y_hat <- ifelse(test_set$height > best_cutoff, "Male", "Female")
mean(y_hat == test_set$sex)
```

```
## [1] 0.817
```

We see that it is lower than the accuracy observed for the training set, but it is still better than guessing. And by testing on a dataset that we did not train on, we know we did not cheat.

Day2

Prevalence, Sensitivity and Specificity

Assuming we develop a prediction rule to predict `Male` if the student is taller than 65 inches. Given that the average female is about 65 inches, this prediction rule is bound to make many errors. If a student is the height of the average female, shouldn't we predict `Female`? A closer look at the confusion matrix reveals the problem. We look at the proportion of calls for each sex:

```
test_set %>%
  mutate(y_hat = y_hat) %>%
  group_by(sex) %>%
  summarize(accuracy = mean(y_hat == sex))
```

```
## # A tibble: 2 x 2
##   sex    accuracy
##   <fct>     <dbl>
## 1 Female    0.605
## 2 Male      0.879
```

There is an imbalance in the accuracy for males and females: too many females are predicted to be male. The reason this does not affect our overall accuracy is because the *prevalence* of males in this dataset is high:

```
prev <- mean(y == "Male")
prev
```

```
## [1] 0.773
```

So the high percentage of mistakes made for females is outweighed by the gains in correct calls for men. This can be a big problem in machine learning. **If your training data is biased in some way, you are likely to develop algorithms that are biased as well. This is one of the reasons we look at metrics other than overall accuracy when evaluating a machine learning algorithm**.

To evaluate an algorithm in a way that prevalence does not cloud our assessment, we can study *sensitivity* and *specificity* separately. These terms are defined for a specific category. Once we specify a category of interest then we can talk about positive outcomes, $Y = 1$, and negative outcomes, $Y = 0$.

In general, *sensitivity* is defined as the ability of an algorithm to predict a positive outcome when the actual outcome is $Y = 1$. Because an algorithm that calls everything $\hat{Y} = 1$ has perfect sensitivity, sensitivity on its own is not enough to judge an algorithm. For this reason we also examine *specificity*, which is generally defined as the ability of an algorithm to *only* call a $\hat{Y} = 1$ when the case is actually $Y = 1$. We can summarize in the following way:

- High sensitivity: $Y = 1 \implies \hat{Y} = 1$
- High specificity: $\hat{Y} = 1 \implies Y = 1$ or equivalently $Y = 0 \implies \hat{Y} = 0$

To provide a precise definition we name the four entries of the confusion matrix:

	Positive	Negative
Predicted positve	True positives (TP)	False positives (FP)
Predicted negative	False negatives (FN)	True negatives (TN)

Sensitivity is typically quantified by $TP/(TP + FN)$, or the proportion of positives `TP+FN` that are called positives `TP`. This quantity is referred to as the *true positive rate* (TPR) or ***recall***.

Specificity is typically quantified as $TN/(TN + FP)$ or the proportion of negatives `TN+FP` that are called negatives `TN`. This quantity is called the true negative rate (TNR). Specificity is sometimes quantified with $TP/(TP + FP)$, the proportion of outcomes called positives $TP + FP$ that are actaully positives TP. This quantity is referred to as ***precision***. Note that unlike TPR and TNR, precision depends on prevelance since higher prevalence implies you can get higher precision, even when guessing.

The caret function `confusionMatrix` computes all these metrics for us once we define what a positive is. The function expects factors as input and coerces characters into factors. The first level is considered the positives. Here `Female` is the first level because it comes before `Male` alphabetically.

```
library(e1071)
```

```
## Warning: package 'e1071' was built under R version 4.2.2
```

```
confusionMatrix(data = as.factor(y_hat), reference = test_set$sex)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Female Male
##     Female     72   49
##     Male       47  357
##
##                Accuracy : 0.817
##                  95% CI : (0.781, 0.849)
##     No Information Rate : 0.773
##     P-Value [Acc > NIR] : 0.00835
##
##                   Kappa : 0.481
##
##  Mcnemar's Test P-Value : 0.91871
##
##             Sensitivity : 0.605
##             Specificity : 0.879
##          Pos Pred Value : 0.595
##          Neg Pred Value : 0.884
##              Prevalence : 0.227
##          Detection Rate : 0.137
##    Detection Prevalence : 0.230
##       Balanced Accuracy : 0.742
```

```
##
##        'Positive' Class : Female
##
```

We can see that the high accuracy is possible despite relatively low sensitivity. The reason this can happen is the low prevalence: because the proportion of females is low, incorrectly classifying them as males does not lower the accuracy as much as it is increased by most males being predicted as males. This is an example of why it's important to examine sensitivity and specificity and not just accuracy.

However, it is often useful to have one number summary, for example for optimization purposes. One metric is simply the average of specificity and sensitivity, referred to as ***balanced accuracy***. However, because these are rates, it is more appropriate to compute the harmonic average of specificity and sensitivity. In fact the $\boldsymbol{F}_1$-score, a widely used one number summary, is the harmonic average of precision and recall:

$$\frac{1}{\frac{1}{2}\left(\frac{1}{\text{recall}}+\frac{1}{\text{precision}}\right)}$$

which can be rewritten as

$$2\frac{\text{precision}\cdot\text{recall}}{\text{precision}+\text{recall}}$$

However, depending on the context, some types of errors are more costly than others. For example, in the case of plane safety it is much more important to maximize sensitivity over specificity: failing to predict a plane will malfunction before it crashes is a much more costly error than grounding a plane when in fact the plane is in perfect condition. In a capital murder criminal case the opposite is true: a false positive can lead to killing an innocent person.

The F_1-score can be adapted to weigh specificity and sensitivity differently. The way it is implemented is by defining β to represent how much more important sensitivity is compared to specificity and consider a weighted harmonic average:

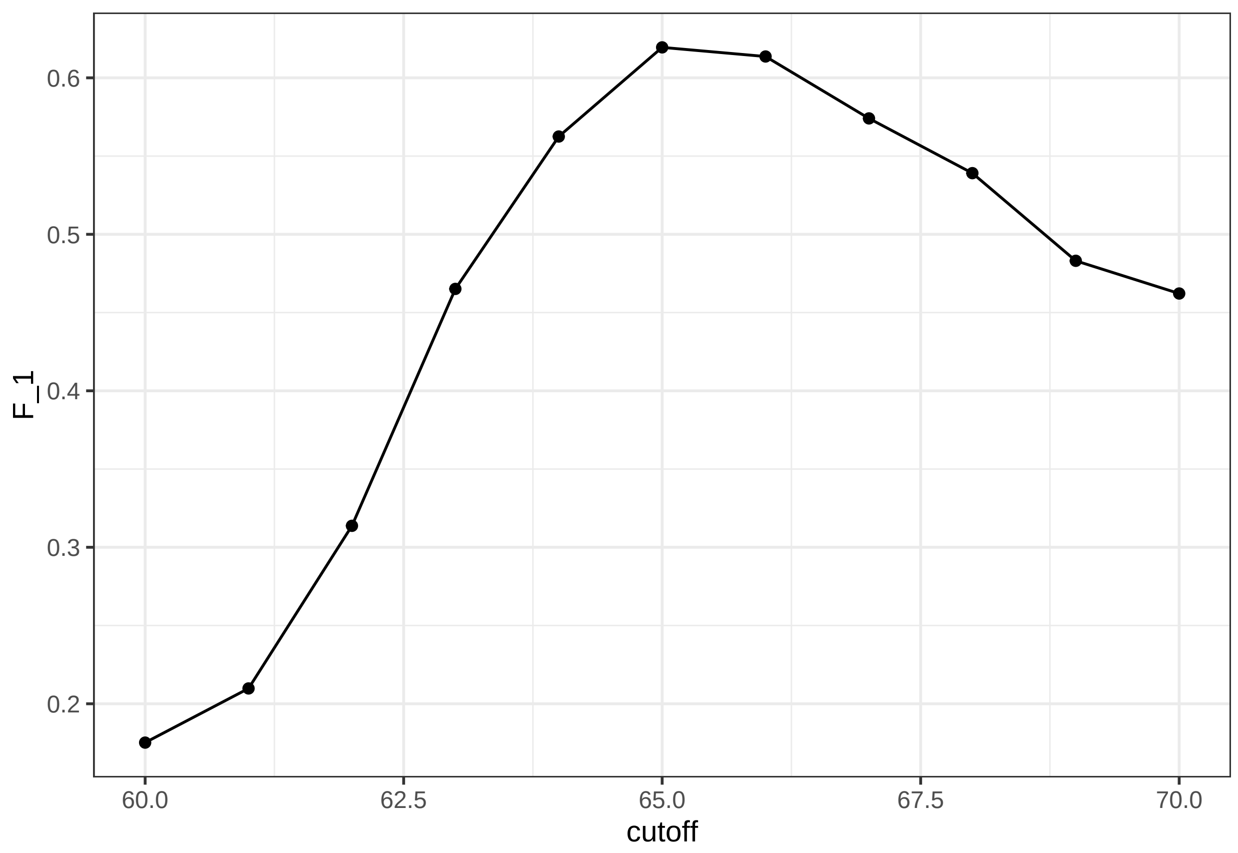
$$\frac{1}{\frac{\beta^2}{1+\beta^2}\frac{1}{\text{recall}}+\frac{1}{1+\beta^2}\frac{1}{\text{precision}}}$$

The `F_meas` function in the caret package computes this summary with `beta` defaulting to 1.

We can reassess our algorithm above using the F-score instead:

```r
cutoff <- seq(60, 70)
F_1 <- map_dbl(cutoff, function(x){
  y_hat <- ifelse(train_set$height > x, "Male", "Female")
  F_meas(data = factor(y_hat), reference = factor(train_set$sex))
})
```

As before, we can plot these F_1 measures versus the cutoffs:



We see that it is maximized at:

```
max(F_1)
```

```
## [1] 0.619
```

when we use cutoff:

```
best_cutoff <- cutoff[which.max(F_1)]
best_cutoff
```

```
## [1] 65
```

However, a cutoff of 66 makes more sense than 65. Furthermore, it balances the specificity and sensitivity of our confusion matrix a bit better:

```
y_hat <- ifelse(test_set$height > 66, "Male", "Female")
confusionMatrix(data = as.factor(y_hat), reference = test_set$sex)
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction Female Male
##     Female     82   69
##     Male       37  337
##
##                Accuracy : 0.798
##                  95% CI : (0.761, 0.832)
##     No Information Rate : 0.773
##     P-Value [Acc > NIR] : 0.0951
##
##                   Kappa : 0.474
##
##  Mcnemar's Test P-Value : 0.0026
##
##             Sensitivity : 0.689
##             Specificity : 0.830
##          Pos Pred Value : 0.543
##          Neg Pred Value : 0.901
##              Prevalence : 0.227
##          Detection Rate : 0.156
##    Detection Prevalence : 0.288
##       Balanced Accuracy : 0.760
##
##        'Positive' Class : Female
##
```

Conditional probabilities

The machine learning classification challenge can be thought of as trying to estimate the probability of Y being any of the K possible categories given a set of predictors $\mathbf{X} = (X_1, \dots, X_p)$. We can quantify this by saying that we are interested in the *conditional probabilities*:

$$p_k(x) = \Pr(Y = k \mid \mathbf{X} = \mathbf{x}),\, k = 1, \dots, K$$

If we know $p_k(x)$ then we can optimize our predictions by, for example, predicting the k with the largest probability:

$$\hat{Y} = \max_k p_k(x)$$

As discussed above, sensitivity and specificity may differ in importance in different contexts. For this reason, maximizing the probability is not always optimal in practice and depends on the context. But even in these cases, knowing $p_k(x)$ will suffice to build optimal prediction models, since we can inform our predictions based on these probabilities and control specificity and sensitivty however we wish. For example, we can simply change the cutoffs used to predict one outcome or the other. In the plane example given above, we may ground the plane anytime the probability of malfunction is higher than 1/1000 as opposed to the default 1/2 used when error types are equally undesired.

To simplify the expression below, let's consider the predicting sex example which is a binary data case.

For binary data, you can think of the probability $\Pr(Y = 1 \mid \mathbf{X} = \mathbf{x})$ as the proportion of 1s in the stratum of the population for which $\mathbf{X} = \mathbf{x}$.

Logistic Regression

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

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

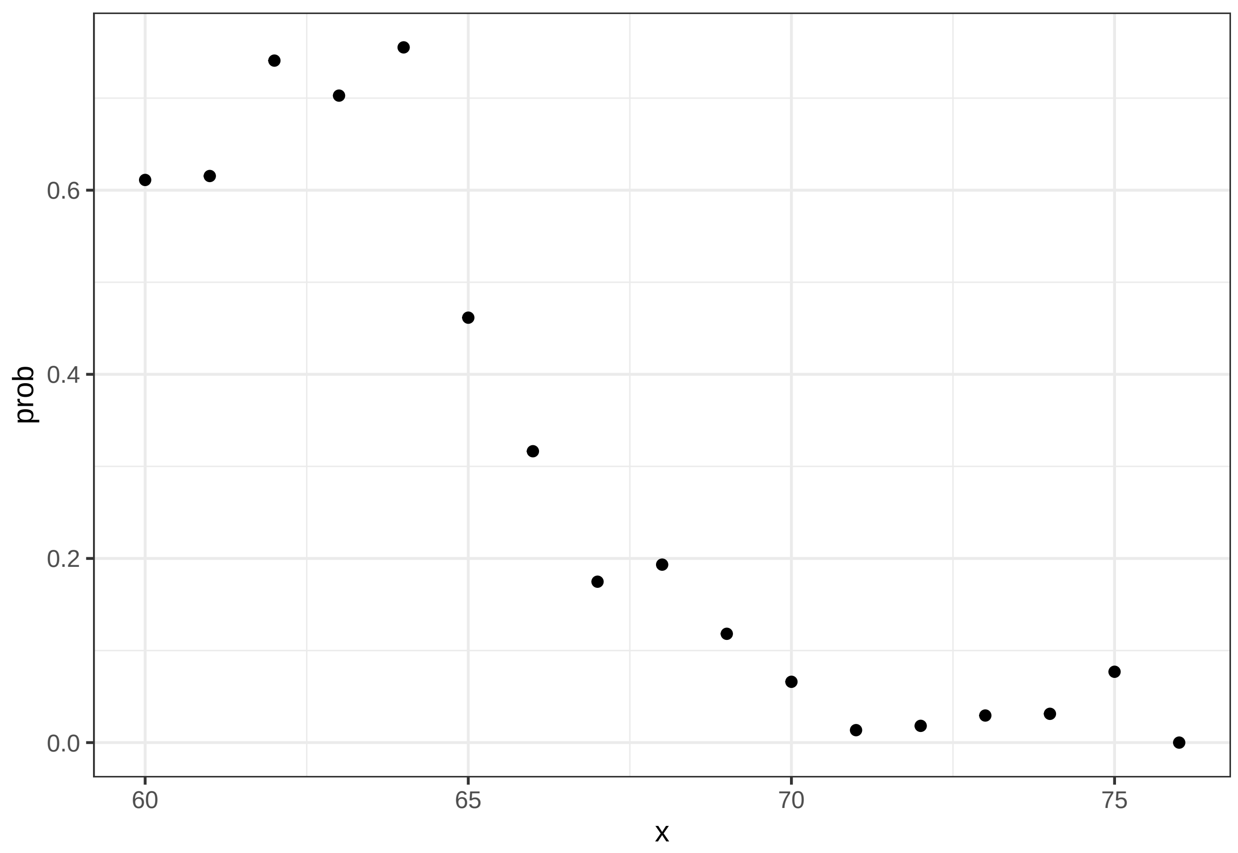
As an example, let's provide a prediction for a student that is 66 inches tall. What is the conditional probability of being female if you are 66 inches tall? In our dataset we can estimate this by rounding to the nearest inch and computing:

```
heights %>%
  filter(round(height)==66) %>%
  summarize(mean(sex=="Female"))
```

```
##   mean(sex == "Female")
## 1                 0.316
```

Using data exploration let's see what this estimate of our conditional probability looks like for several values of x. We will remove values of x with few data points using the `n()` function.

```
heights %>%
  mutate(x = round(height)) %>%
  group_by(x) %>%
  filter(n() >= 10) %>%
  summarize(prob = mean(sex == "Female")) %>%
  ggplot(aes(x, prob)) +
  geom_point()
```



This plot suggests that the conditional probability decreases with x and, at least in some parts, appears to be a linear function of x. Let's assume that this is the case for now and use the following model:

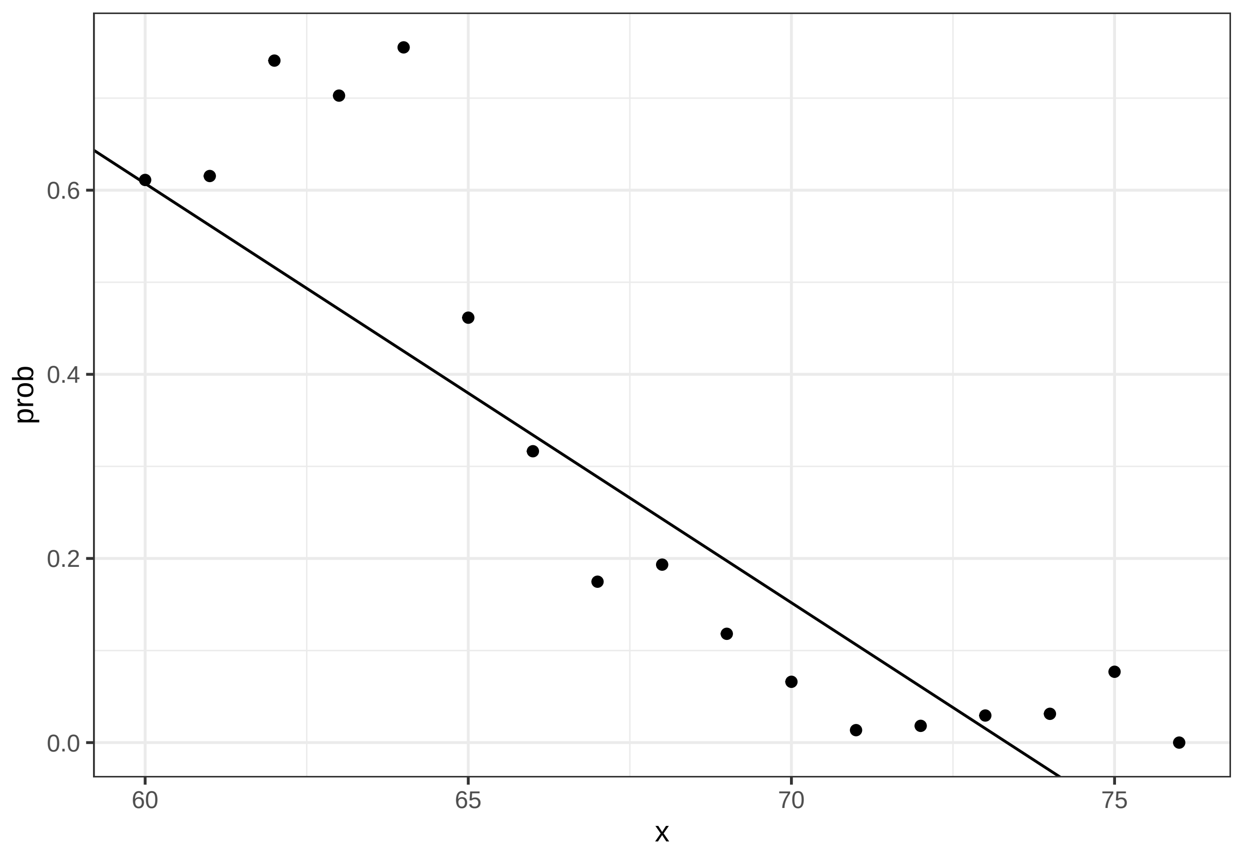
$$p(x) = \Pr(Y = 1|X = x) = \beta_0 + \beta_1 x$$

If we can estimate β_0 and β_1 we will have an estimate of $p(x)$ which will permit us to build a classification algorithm. We will use least squares (linear regression).

```r
library(broom)
betas <- train_set %>%
  mutate(y = as.numeric(sex == "Female")) %>%
  do(tidy(lm(y ~ height, data = .))) %>%
  .$estimate
```

The model fits the data relatively well:

```r
heights %>%
  mutate(x = round(height)) %>%
  group_by(x) %>%
  filter(n() >= 10) %>%
  summarize(prob = mean(sex == "Female")) %>%
  ggplot(aes(x, prob)) +
  geom_point() +
  geom_abline(intercept = betas[1], slope = betas[2])
```



and we can define an actual prediction rule, for example: define

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

and predict `Female` if $\hat{p}(x) > 0.5$. Here is the confusion matrix and relevant statistics:

```
p_hat <- betas[1] + betas[2]*test_set$height
prediction <- ifelse(p_hat > 0.5, "Female", "Male")
confusionMatrix(data = as.factor(prediction), reference = test_set$sex)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Female Male
##     Female     22   18
##     Male       97  388
##
##                Accuracy : 0.781
##                  95% CI : (0.743, 0.816)
##     No Information Rate : 0.773
##     P-Value [Acc > NIR] : 0.361
##
##                   Kappa : 0.184
##
##  Mcnemar's Test P-Value : 3.5e-13
```

```
##
##             Sensitivity : 0.1849
##             Specificity : 0.9557
##          Pos Pred Value : 0.5500
##          Neg Pred Value : 0.8000
##              Prevalence : 0.2267
##          Detection Rate : 0.0419
##    Detection Prevalence : 0.0762
##       Balanced Accuracy : 0.5703
##
##        'Positive' Class : Female
##
```

As before we see an imbalance of specificity and sensitivity. In this case we can fix this by changing the probability cutoff.

One problem with this approach is that $\hat{p}(x)$ can be outside the [0,1] range:

```
range(betas[1] + betas[2]*test_set$height)
```

```
## [1] -0.425  1.063
```

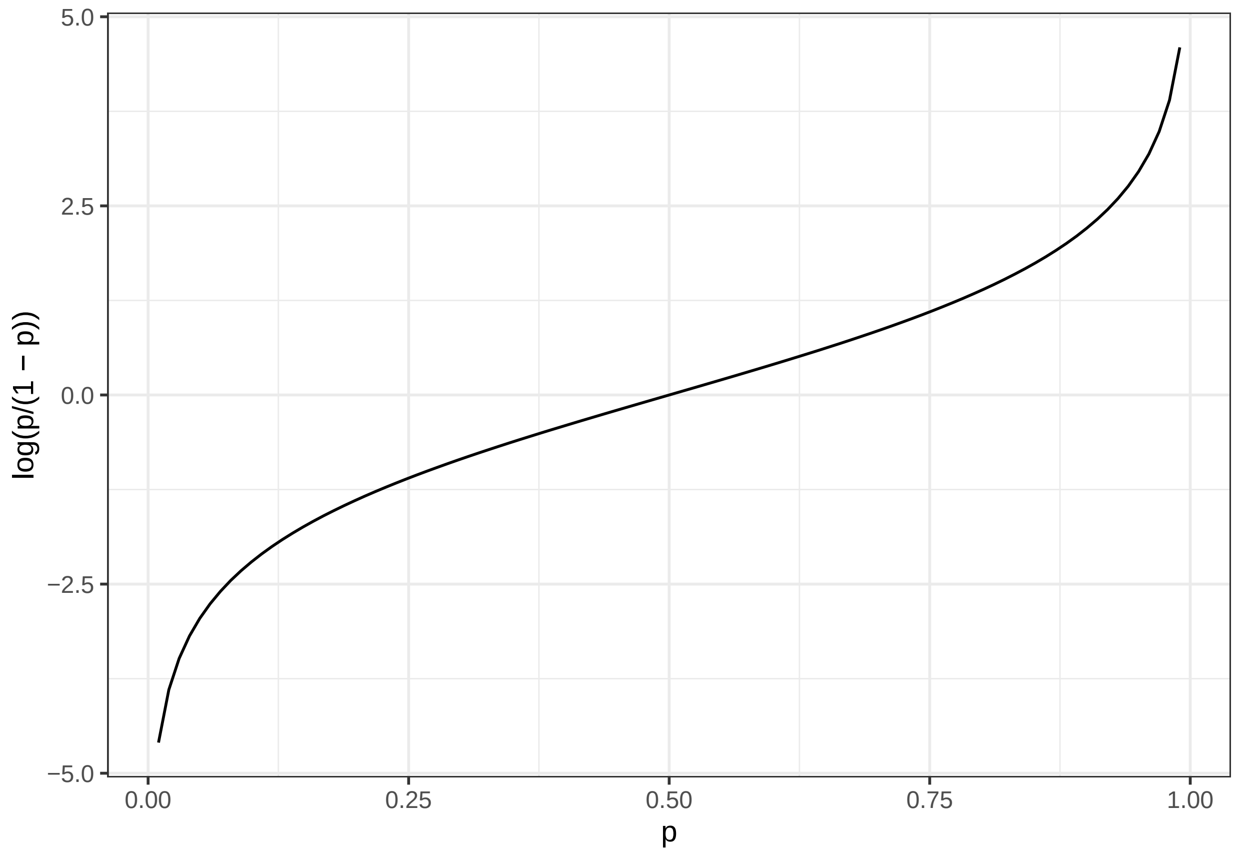
An extension of the regression model that permits us to continue using a regression-like approach is to apply transformations that guarantee that $\hat{p}(x)$ will be between 0 and 1.

In the case of binary data the most common approach is to fit a *logistic regression* model which makes use of the *logistic* transformation

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

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

```
## Warning: `qplot()` was deprecated in ggplot2 3.4.0.
```



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

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

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 text.

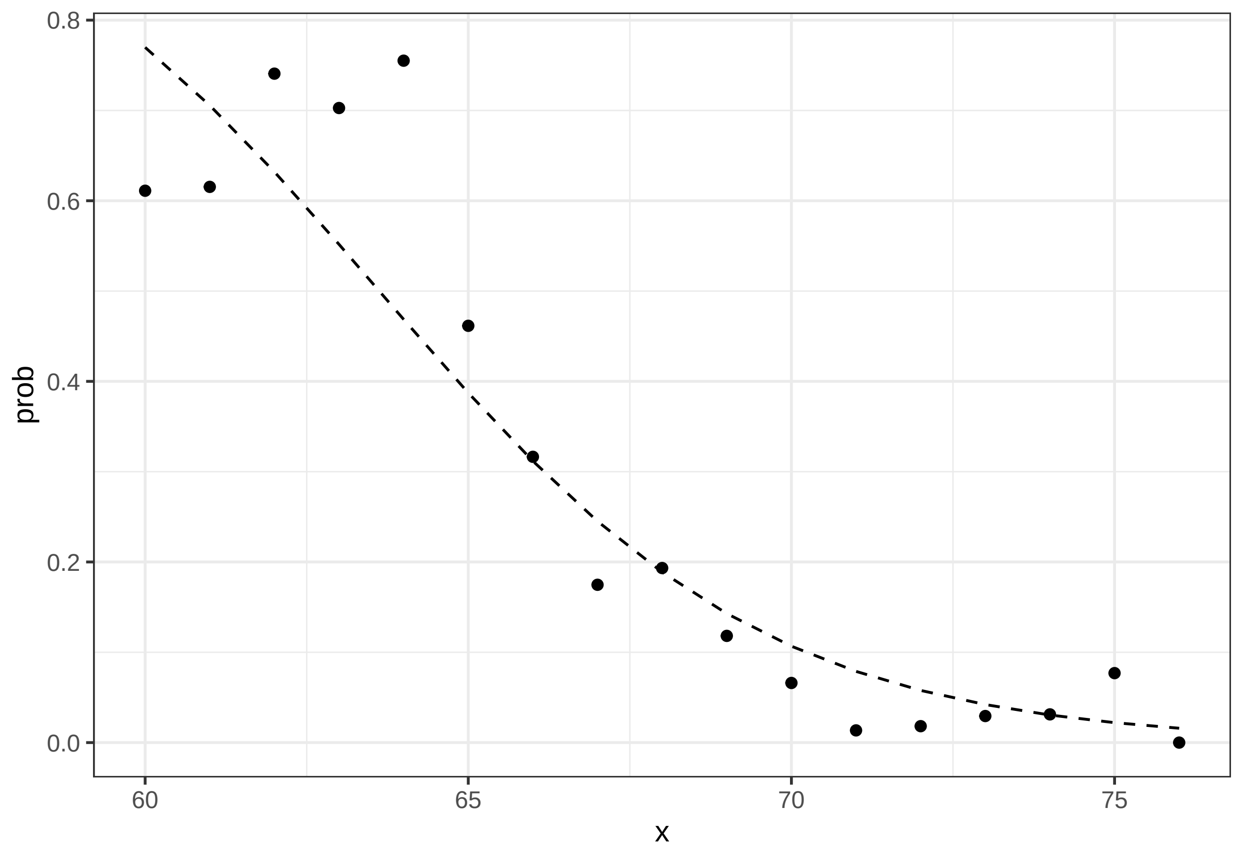
In R we can fit the logistic regression model with the function `glm`: generalized linear models. This function is more general than logistic regression so we need to specify the model we want through the `family` parameter:

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

We can obtain predictions using the predict function:

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

Note that this model fits the data slightly better than the line:



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

```r
y_hat_logit <- ifelse(p_hat_logit > 0.5, "Female", "Male")
confusionMatrix(data = as.factor(y_hat_logit), reference = test_set$sex)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Female Male
##     Female     35   26
##     Male       84  380
##
##                Accuracy : 0.79
##                  95% CI : (0.753, 0.825)
##     No Information Rate : 0.773
##     P-Value [Acc > NIR] : 0.188
##
##                   Kappa : 0.278
##
##  Mcnemar's Test P-Value : 5.49e-08
##
##             Sensitivity : 0.2941
##             Specificity : 0.9360
##          Pos Pred Value : 0.5738
##          Neg Pred Value : 0.8190
##              Prevalence : 0.2267
##          Detection Rate : 0.0667
##    Detection Prevalence : 0.1162
##       Balanced Accuracy : 0.6150
##
##        'Positive' Class : Female
##
```

Naive Bayes

The best we can do in a Machine Learning problem is when we actually know

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

This gives us what we call *Bayes' Rule*. However, in practice we don't know Bayes' Rule, since estimating $p(x)$ is the main challenge.

Naive Bayes is an approach that tries to estimate $p(x)$ using Bayes theorem.

In this particular example we know that the normal distribution works rather well for the heights X for both classes $y = 1$ (female) and $y = 0$ (male). This implies that we can approximate the conditional distributions $f_{X|Y=1}$ and $f_{X|Y=0}$. These are the height distributions for females and males respectively and we can easily estimate all the necessary parameters from the data:

```
params <- train_set %>%
  group_by(sex) %>%
  summarize(avg = mean(height), sd = sd(height))
params
```

```
## # A tibble: 2 x 3
##   sex      avg    sd
##   <fct>  <dbl> <dbl>
## 1 Female  65.0  4.21
## 2 Male    69.3  3.47
```

Then using Bayes rule we can compute:

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

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

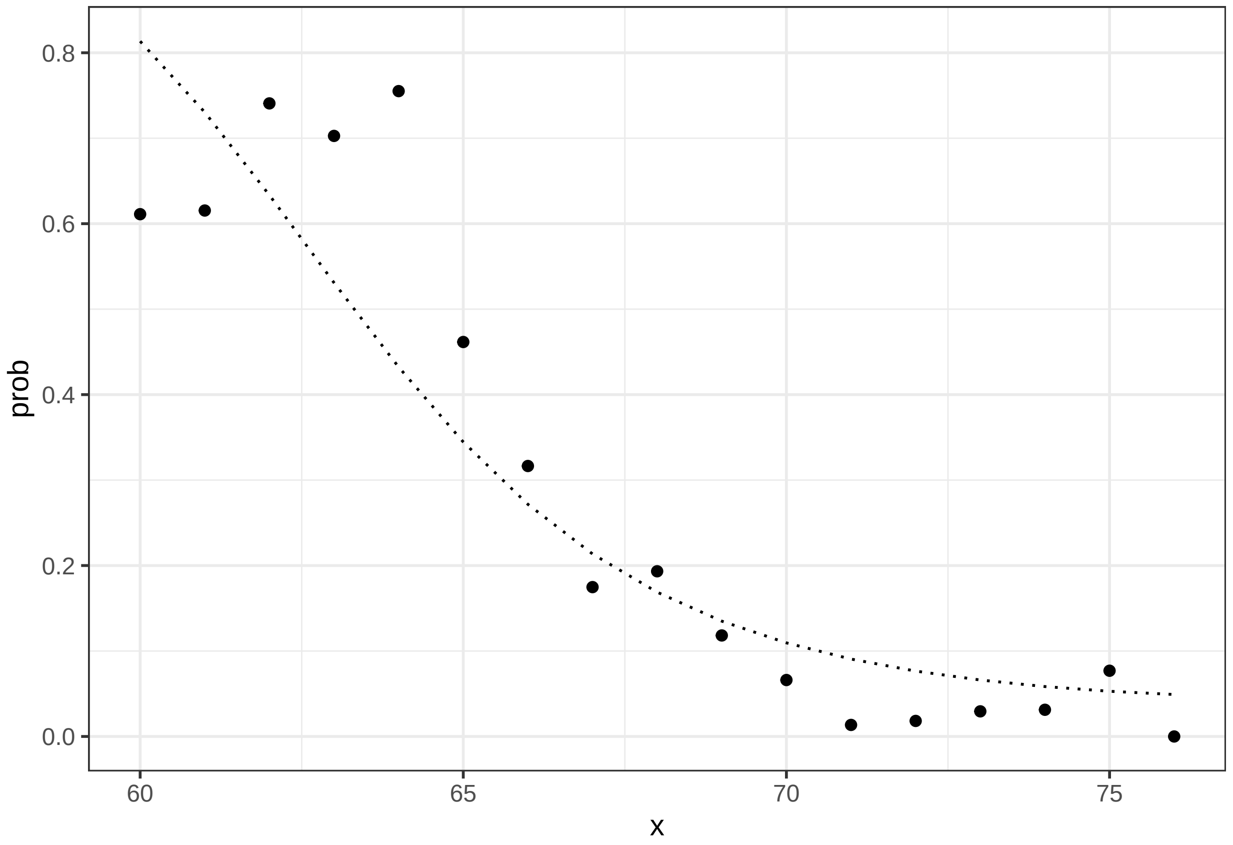
```
pi <- train_set %>%
  summarize(pi = mean(sex == "Female")) %>%
  .$pi
pi
```

```
## [1] 0.227
```

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

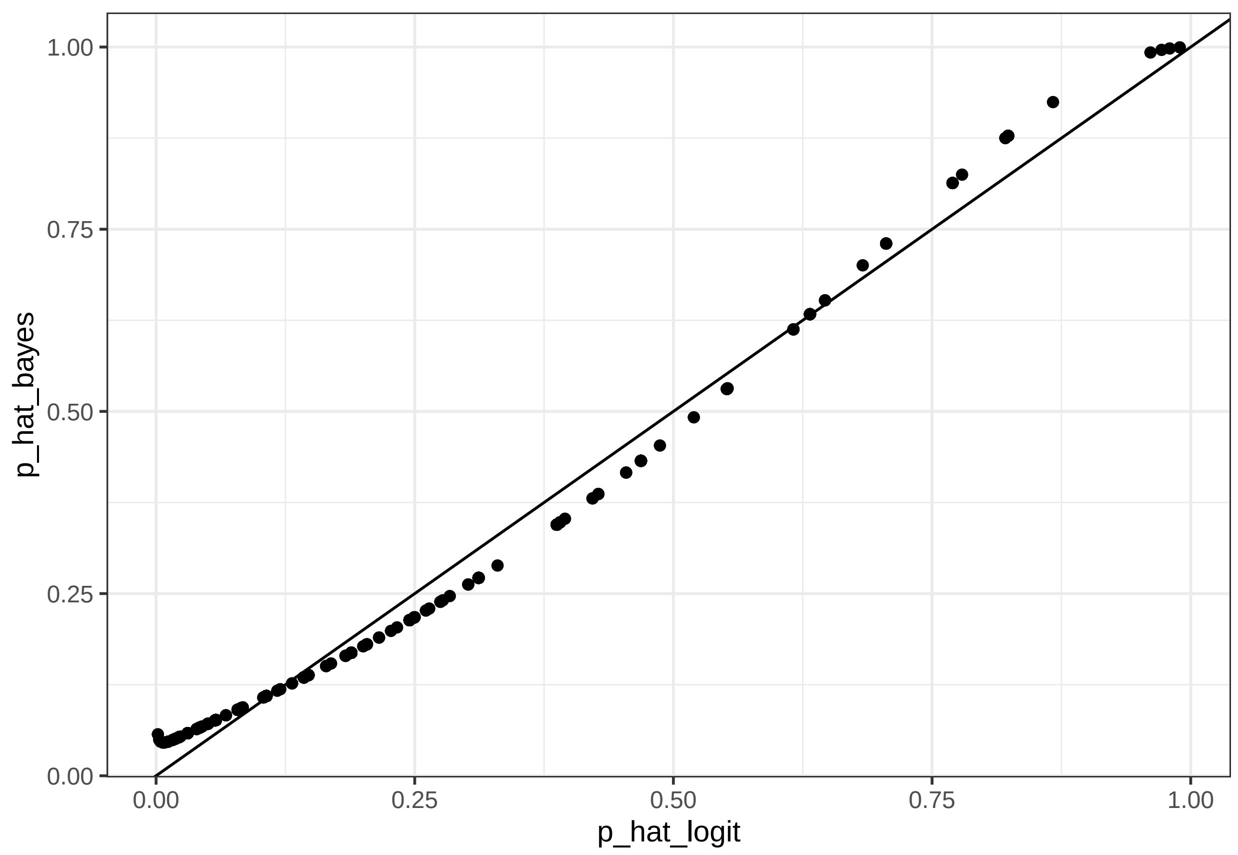
```
x <- test_set$height
f0 <- dnorm(x, params$avg[2], params$sd[2])
f1 <- dnorm(x, params$avg[1], params$sd[1])
p_hat_bayes <- f1*pi / (f1*pi + f0*(1 - pi))
```

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



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 this one. We can see that they are similar empirically:

```
qplot(p_hat_logit, p_hat_bayes) + geom_abline()
```



Let's look at the confusion matrix to see how well we are performing. Our results are very close to what we got for our logistic regression model.

```
y_hat_bayes <- ifelse(p_hat_bayes > 0.5, "Female", "Male")
confusionMatrix(data = as.factor(y_hat_bayes), reference = test_set$sex)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Female Male
##     Female     35   25
##     Male       84  381
##
##                Accuracy : 0.792
##                  95% CI : (0.755, 0.826)
##     No Information Rate : 0.773
##     P-Value [Acc > NIR] : 0.161
##
##                   Kappa : 0.282
##
##  Mcnemar's Test P-Value : 2.77e-08
##
##             Sensitivity : 0.2941
##             Specificity : 0.9384
##          Pos Pred Value : 0.5833
##          Neg Pred Value : 0.8194
##               Prevalence : 0.2267
##           Detection Rate : 0.0667
##     Detection Prevalence : 0.1143
##        Balanced Accuracy : 0.6163
##
##         'Positive' Class : Female
##
```

Day3

Controlling prevalence

One nice 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, our sample has a much lower prevalence, 0.227, 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] 0.294
```

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] 0.938
```

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, for example, π. So to balance specificity and sensitivity, instead of changing the cutoff in the decision rule we could simply change $\hat{\pi}$:

```
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")
```

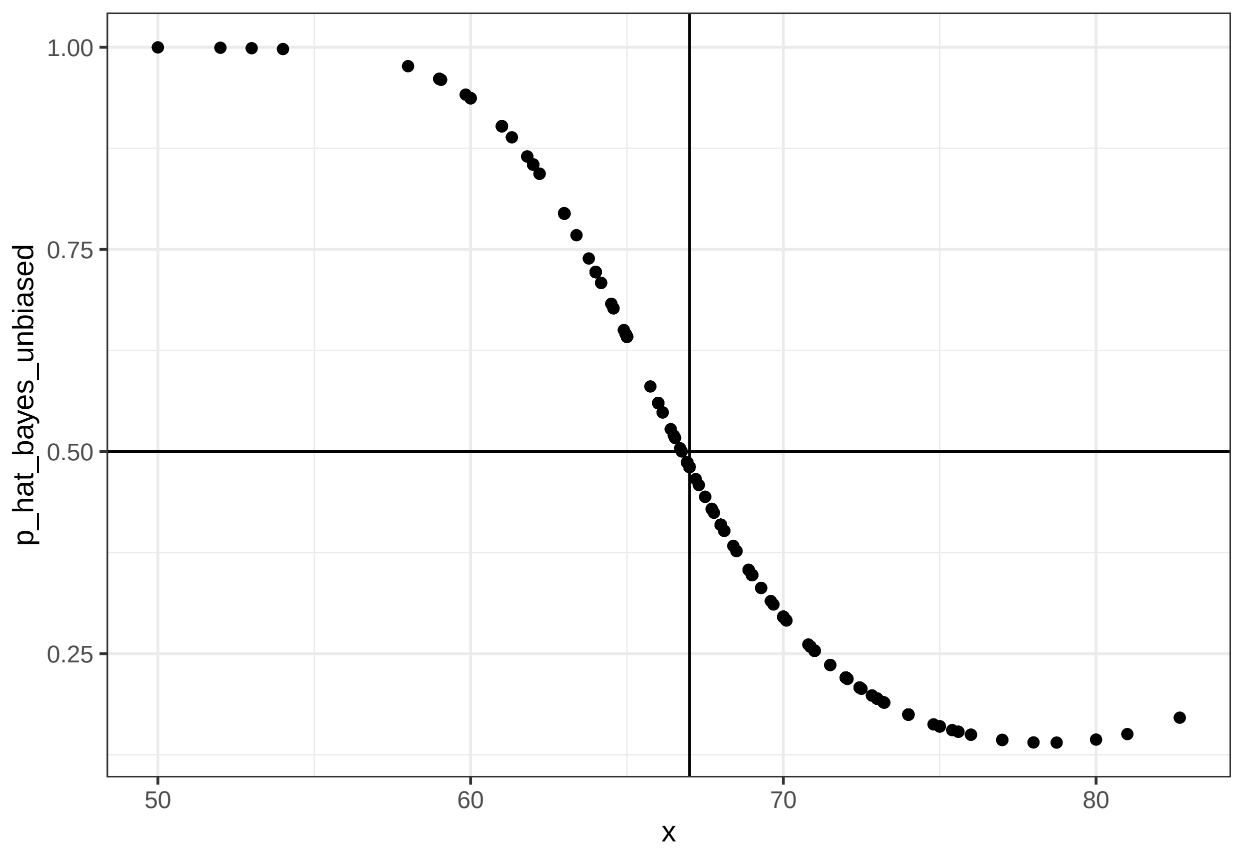
Note the difference in sensitivity with a better balance:

```r
confusionMatrix(data = as.factor(y_hat_bayes_unbiased), reference = test_set$sex)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Female Male
##     Female     84   75
##     Male       35  331
##
##                Accuracy : 0.79
##                  95% CI : (0.753, 0.825)
##     No Information Rate : 0.773
##     P-Value [Acc > NIR] : 0.1883
##
##                   Kappa : 0.466
##
##  Mcnemar's Test P-Value : 0.0002
##
##             Sensitivity : 0.706
##             Specificity : 0.815
##          Pos Pred Value : 0.528
##          Neg Pred Value : 0.904
##              Prevalence : 0.227
##          Detection Rate : 0.160
##    Detection Prevalence : 0.303
##       Balanced Accuracy : 0.761
##
##        'Positive' Class : Female
##
```

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

```r
qplot(x, p_hat_bayes_unbiased) + geom_hline(yintercept = 0.5) + geom_vline(xintercept = 67)
```



Now that we know how to perform Naive Bayes by hand and understand the math behind it, we can use the `naiveBayes` function from the `e1071` package to do these calculations automatically. Here, `naiveBayes` outputs the prediction class automatically with a default cutoff of 0.5 for the probabilities. Thus, we don't need to classify a value as Female or Male using a threshold by hand.

We see we get the exact same confusion matrix we got before we changed the prevalence.

```
nb_fit   <- naiveBayes(sex ~ height, data = train_set)
y_hat_nb <- predict(nb_fit, test_set)
confusionMatrix(data = as.factor(y_hat_nb), reference = test_set$sex)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Female Male
##     Female     35   25
##     Male       84  381
##
##                Accuracy : 0.792
##                  95% CI : (0.755, 0.826)
##     No Information Rate : 0.773
##     P-Value [Acc > NIR] : 0.161
##
##                   Kappa : 0.282
##
##  Mcnemar's Test P-Value : 2.77e-08
```

```
##
##             Sensitivity : 0.2941
##             Specificity : 0.9384
##          Pos Pred Value : 0.5833
##          Neg Pred Value : 0.8194
##              Prevalence : 0.2267
##          Detection Rate : 0.0667
##    Detection Prevalence : 0.1143
##       Balanced Accuracy : 0.6163
##
##        'Positive' Class : Female
##
```

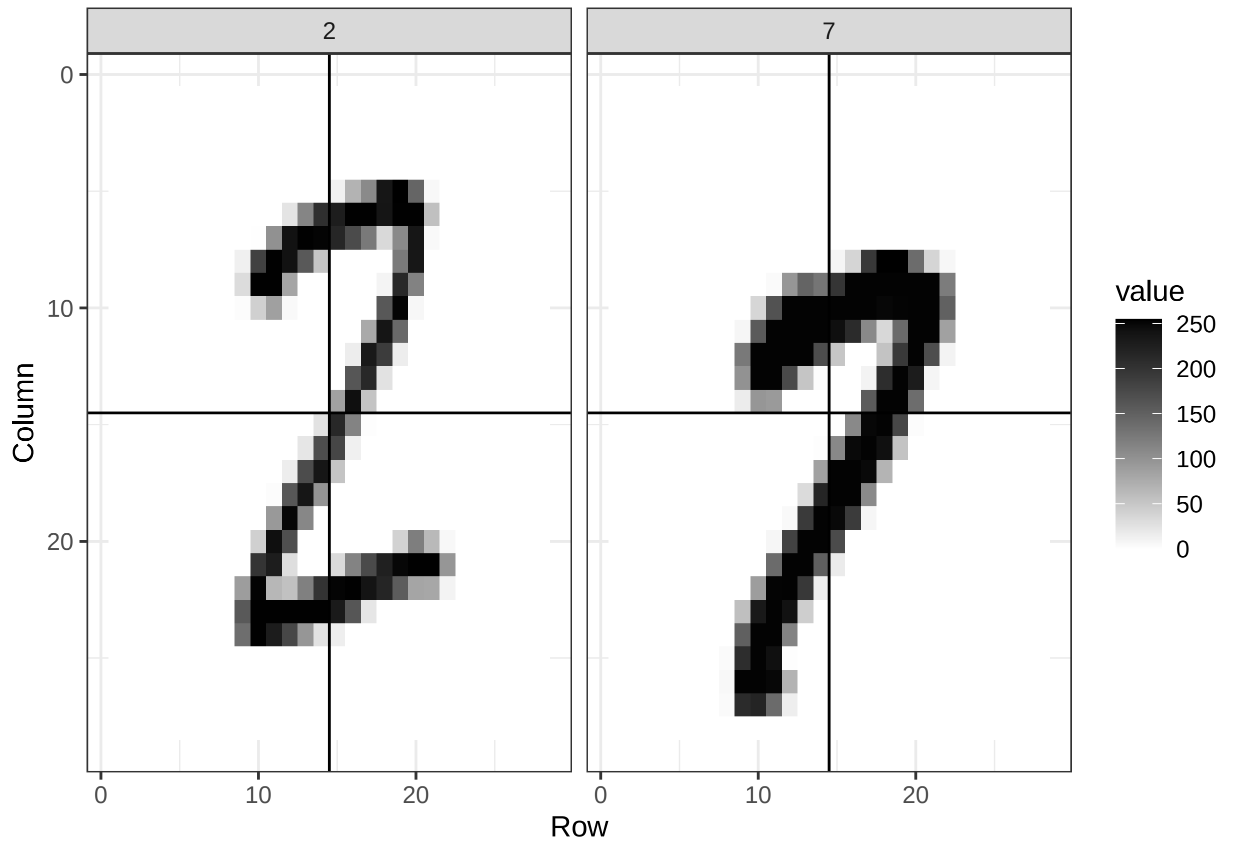
Two Predictors

In the two simple examples above we only had one predictor. We actually do not consider these machine learning challenges, which are characterized by including many predictors. Let's go back to the digits example in which we had 784 predictors. For illustrative purposes we will build an example with 2 features and only two classes, 2s and 7s. Then we will go back to the original 784 feature example.

First let's filter to include only 2s and 7s and change the labels from numbers to factors. This second step is important to assure that R does not treat the 2 and 7 as numbers.

```
digits_27 <- digits %>% filter(label %in% c(2,7)) %>%
  mutate(label =  as.character(label))
```

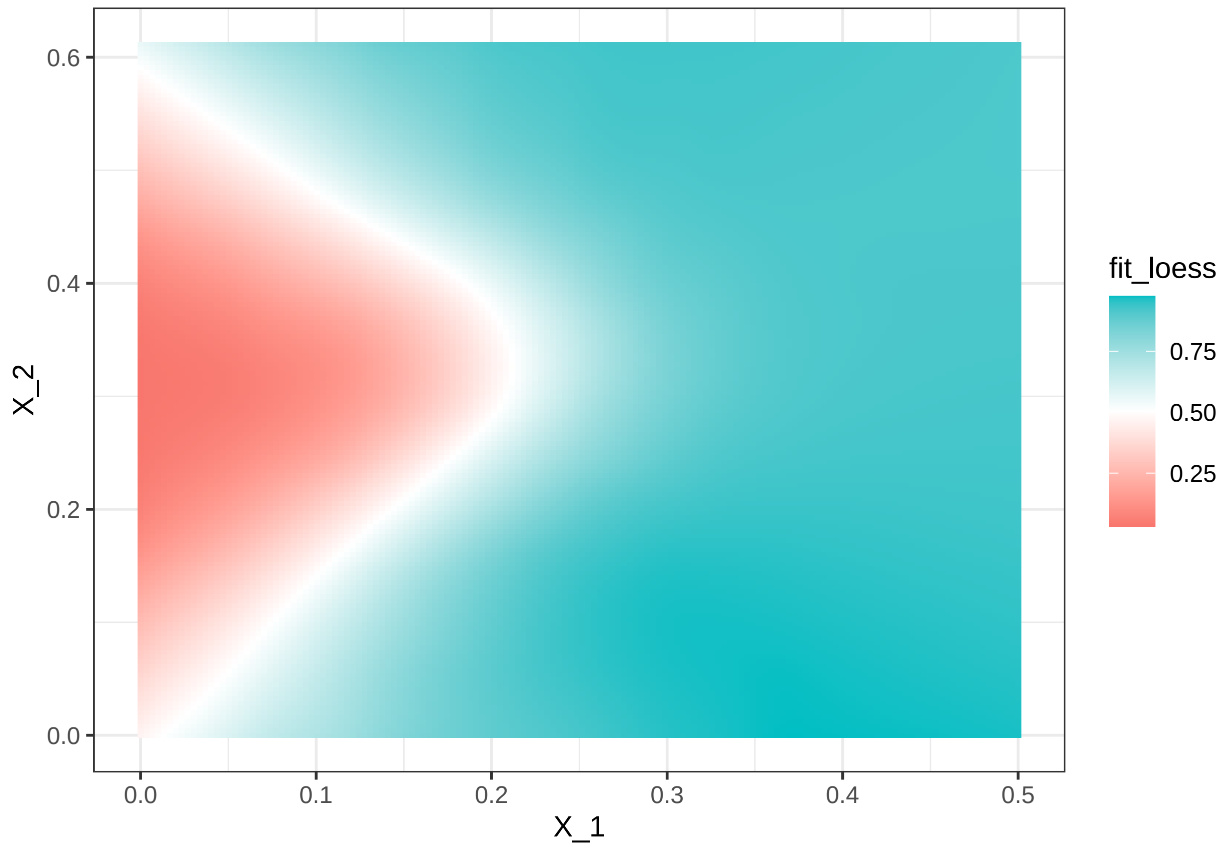
We note that to distinguish 2s from 7s it might be enough to look at the number of non-white pixels in the upper-left and lower-bottom quadrants:



So we will define two features X_1 and X_2 as the percent of non-white pixels in these two quadrants respectively. We add these two features to the `digits_27` table

```
row_column <- expand.grid(row=1:28, col=1:28)
ind1 <- which(row_column$col <= 14 & row_column$row <=14)
ind2 <- which(row_column$col > 14 & row_column$row > 14)
ind <- c(ind1,ind2)
X <- as.matrix(digits_27[,-1])
X <- X > 200
X1 <- rowSums(X[,ind1])/rowSums(X)
X2 <- rowSums(X[,ind2])/rowSums(X)
digits_27 <- digits_27 %>%
  mutate(y = ifelse(label == "7", 1, 0),
         X_1 = X1, X_2 = X2)
```

For illustrative purposes we consider this to be the population and use this data to define a function $p(X_1, X_2)$. Here is the conditional probability of being a 7 as a function of (X_1, X_2).



Next we will take a smaller random sample to mimic our training data as well as our test data.

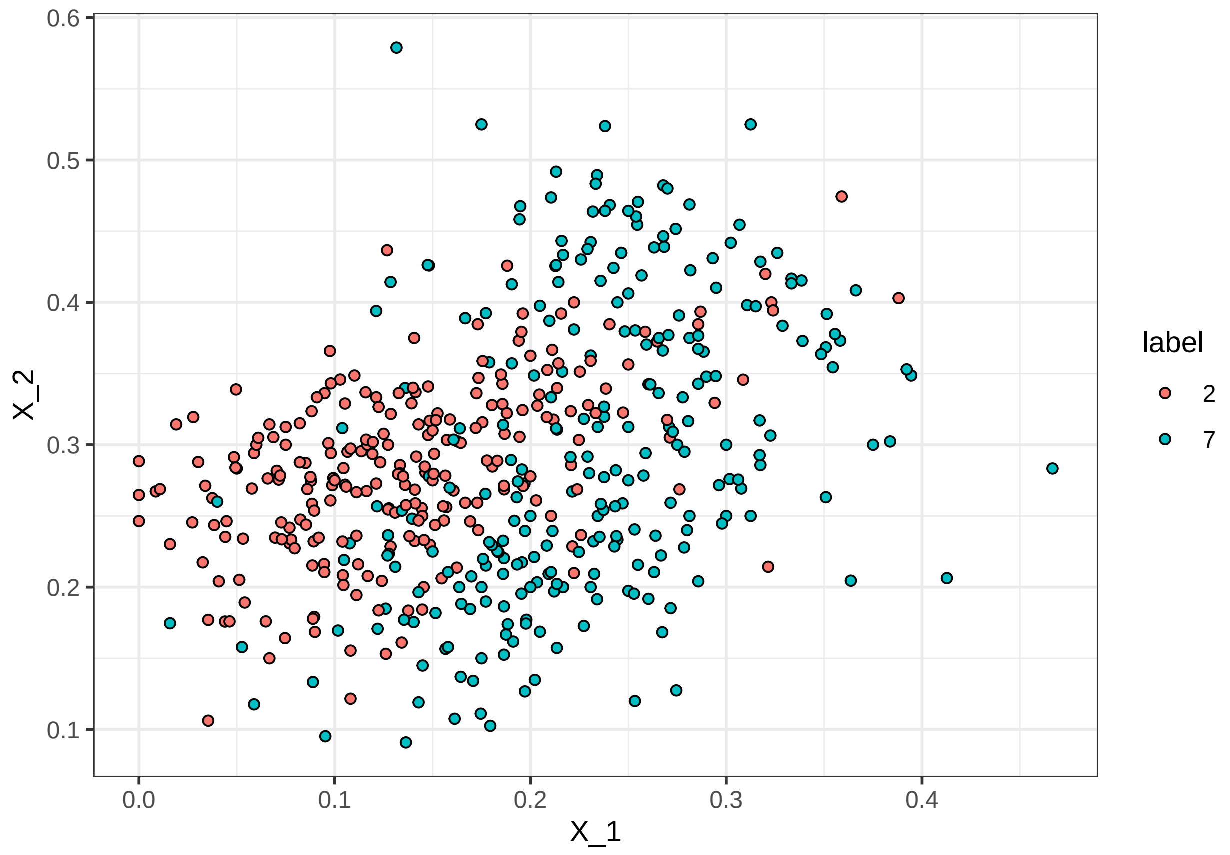
```
set.seed(1971)
dat <- sample_n(digits_27, 1000)
```

We start by creating a train and test set using the `caret` package:

```
library(caret)
index_train<- createDataPartition(y = dat$label, times =1, p=0.5, list = FALSE)
train_set <- slice(dat, index_train)
test_set <- slice(dat, -index_train)
```

We can visualize the training data now using color to denote the classes:

```
train_set %>%
  ggplot(aes(X_1, X_2, fill = label)) +
  geom_point(pch=21)
```



Let's try logistic regression. The model is simply:

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

and we fit it like this:

```
fit <-  glm(y ~ X_1 + X_2, data=train_set, family="binomial")
```

```
p_hat <- predict(fit, newdata = test_set)
y_hat <- ifelse(p_hat > 0.5, 1, 0)
confusionMatrix(data = as.factor(y_hat), reference = as.factor(test_set$y))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 205  63
##          1  38 194
##
##                Accuracy : 0.798
##                  95% CI : (0.76, 0.832)
##     No Information Rate : 0.514
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.597
```

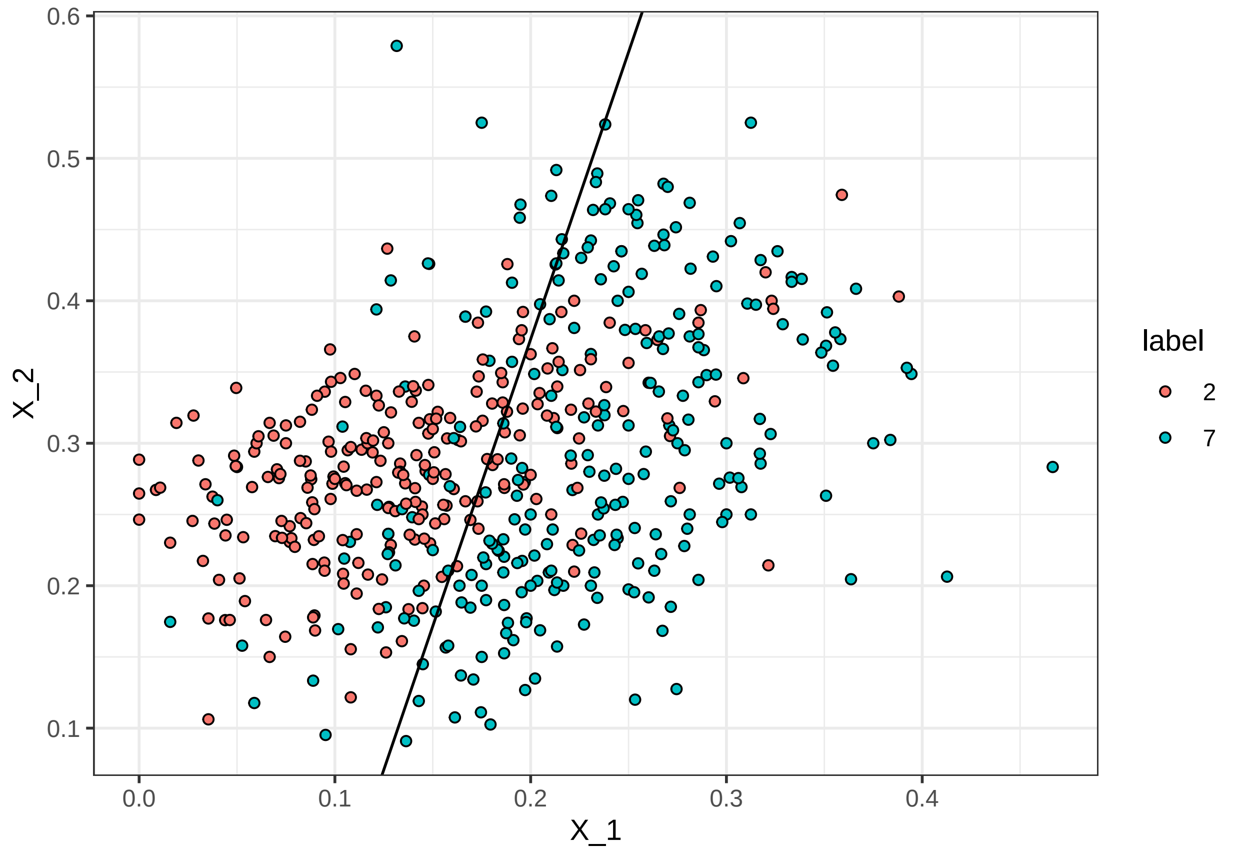
```
##
##  Mcnemar's Test P-Value : 0.0169
##
##             Sensitivity : 0.844
##             Specificity : 0.755
##          Pos Pred Value : 0.765
##          Neg Pred Value : 0.836
##              Prevalence : 0.486
##          Detection Rate : 0.410
##    Detection Prevalence : 0.536
##       Balanced Accuracy : 0.799
##
##        'Positive' Class : 0
##
```

Since we are using 0.5 as our cutoff, and the $\log\{0.5/(1-0.5)\} = 0$ we know that the decisiotn rule is to call a 7 if $\hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 > 0$ and 2 otherwise. This implies that the function

$$\hat{\beta}_0 + \hat{\beta}_1 x_1 + \hat{\beta}_2 x_2 = 0 \implies x_2 = -\hat{\beta}_0/\hat{\beta}_2 - \hat{\beta}_1 x_1/\hat{\beta}_2$$

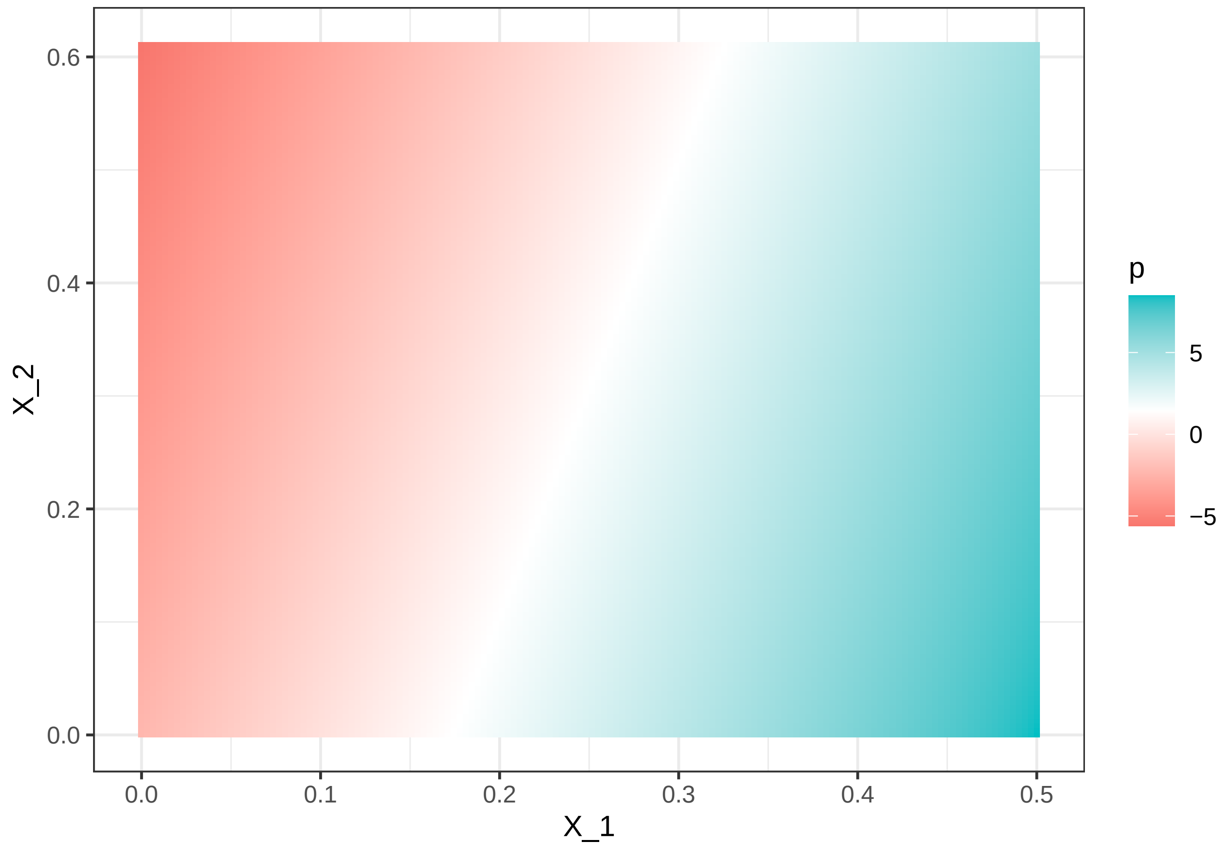
splits the x_1, x_2 plane in areas in which we call twos and areas in which we call sevens.

```r
train_set %>% ggplot(aes(X_1, X_2, fill = label)) +
  geom_point(pch=21) +
  geom_abline(intercept = -fit$coef[1]/fit$coef[3],
              slope = -fit$coef[2]/fit$coef[3])
```



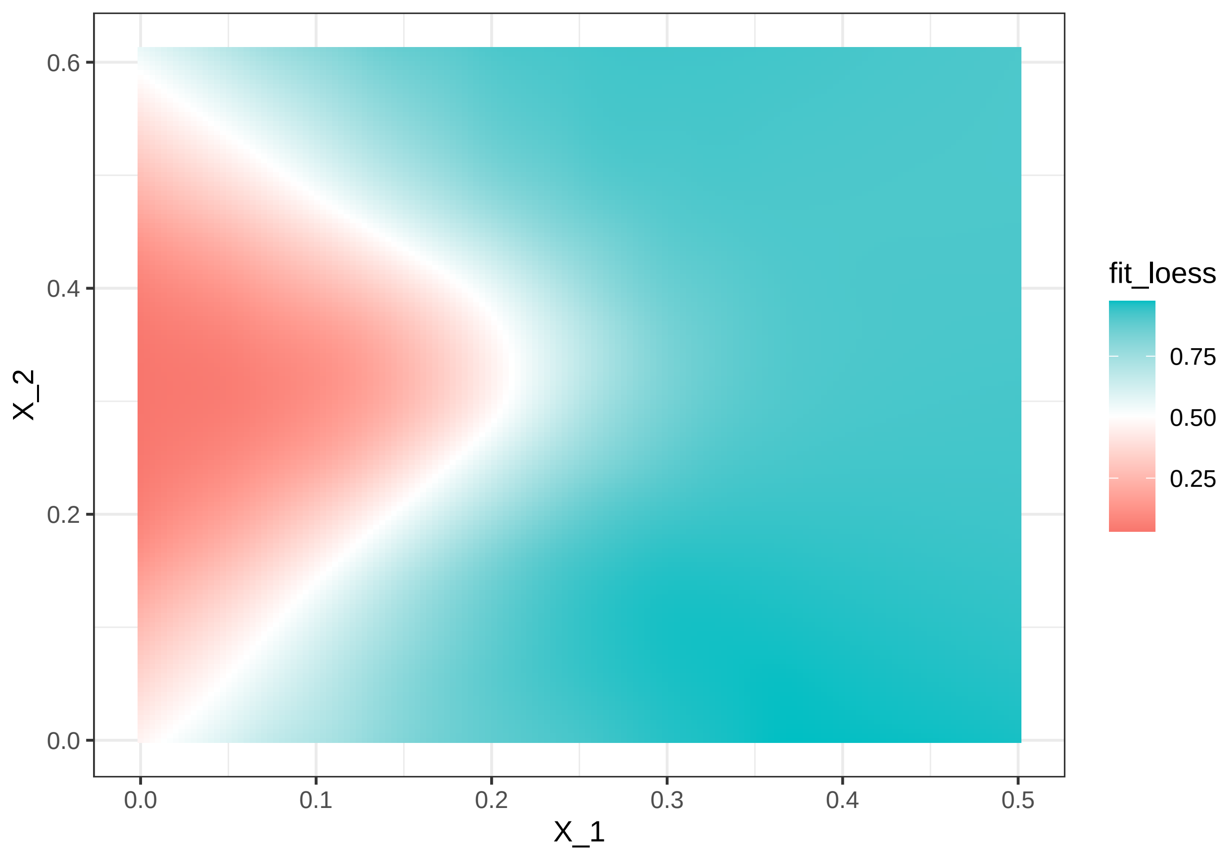
The estimate $\hat{p}(x_1, x_2)$ does not approximate the $p(x_1, x_2)$ very well:

```r
p_x %>% mutate(p = predict(fit, newdata = .)) %>%
  ggplot(aes(X_1, X_2, fill=p))  +
  scale_fill_gradientn(colors=c("#F8766D","white","#00BFC4"))+ geom_raster()
```



Given the shape of $p(x_1, x_2)$ it is impossible for a logistic regression model to provide a decent estimate because with logistic regression the estimate can only be a plane and the true conditional probability is not:

```r
p_x_plot
```



We will learn other machine learning algorithms that provide more flexibility.

Multiple predictors

What if we have two or more predictors? Here it is helpful to understand the concept of *distance*.

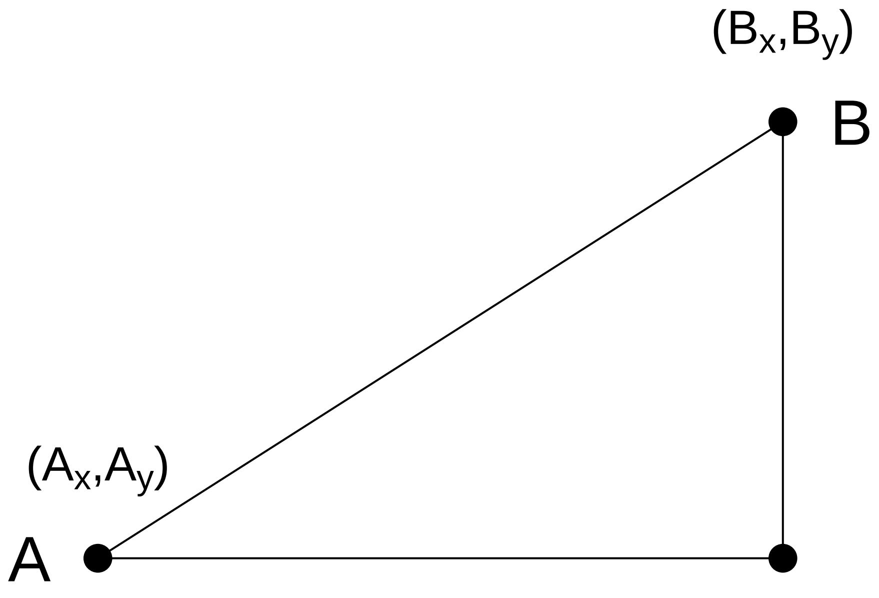
Distance

The concept of distance is quite intuitive. For example, when we cluster animals into subgroups, we are implicitly defining a distance that permits us to say what animals are "close" to each other.

Many of the analyses we perform with high-dimensional data relate directly or indirectly to distance. Many clustering and machine learning techniques rely on being able to define distance, using features or predictors.

Euclidean Distance

As a review, let's define the distance between two points, A and B, on a Cartesian plane.



The euclidean distance between A and B is simply:

$$\sqrt{(A_x - B_x)^2 + (A_y - B_y)^2}$$

Distance in High Dimensions

Earlier we introduced a training dataset with feature matrix measurements for 784 features for 500 digits.

```
sample_n(train_set,10) %>% select(label, pixel351:pixel360)
```

```
## # A tibble: 10 x 11
##    label pixel~1 pixel~2 pixel~3 pixel~4 pixel~5 pixel~6 pixel~7 pixel~8 pixel~9
##    <chr>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
##  1 7           0     128     254     189       0       0       0       0       0
##  2 7           0       0     161     252     253     179       0       0       0
##  3 2           0       0     164     253     253     253      41       0       0
##  4 2           0       0       0      76     244     253     186       0       0
##  5 2           0       0     134     255      63       0       0       0       0
##  6 2          58     191     252     252     145       6       0       0       0
##  7 7         120     252     160       0       0       0       0       0       0
##  8 7           0       0      14     202     253     253     179       6       0
##  9 2           0       0     114     253     253     138       0       0       0
## 10 2         253     154       0       0       0       0       0       0       0
## # ... with 1 more variable: pixel360 <dbl>, and abbreviated variable names
## #   1: pixel351, 2: pixel352, 3: pixel353, 4: pixel354, 5: pixel355,
## #   6: pixel356, 7: pixel357, 8: pixel358, 9: pixel359
```

We are interested in describing the distance between observations, in this case digits. Later, for the purposes of selecting features, we might also be interested in finding pixels that *behave similarly* across samples.

To define distance, we need to know what the points are since mathematical distance is computed between points. With high dimensional data, points are no longer on the Cartesian plane. Instead they are in higher dimensions. For example, observation i is defined by a point in 784 dimensional space: $(Y_{i,1}, \dots, Y_{i,784})^\top$. Feature j is defined by a point in 500 dimensions $(Y_{1,j}, \dots, Y_{500,j})^\top$

Once we define points, the Euclidean distance is defined in a very similar way as it is defined for two dimensions. For instance, the distance between two observations, say observations $i = 1$ and $i = 2$ is:

$$\mbox{dist}(1,2) = \sqrt{\sum_{j=1}^{784} (Y_{1,j} - Y_{2,j})^2}$$

and the distance between two features, say, 15 and 273 is:

$$\mbox{dist}(15,273) = \sqrt{\sum_{i=1}^{500} (Y_{i,15} - Y_{i,273})^2}$$

Example The first thing we will do is create a *matrix* with the predictors

```r
X <- select(train_set , pixel0:pixel783) %>% as.matrix()
```

Rows and columns of matrices can be accessed like this:

```r
third_row <- X[3,]
tenth_column <- X[,10]
```

So the first two observations are 7s and the 253rd is a 2. Let's see if their distances match this:

```r
X_1 <- X[1,]
X_2 <- X[2,]
X_253 <- X[253,]
sqrt(sum((X_1-X_2)^2))
```

```
## [1] 1935
```

```r
sqrt(sum((X_1-X_253)^2))
```

```
## [1] 1950
```

As expected, the 7s are closer to each other. If you know matrix algebra, note that a faster way to compute this is using matrix algebra:

```
sqrt( crossprod(X_1-X_2) )
```

```
##      [,1]
## [1,] 1935
```

```
sqrt( crossprod(X_1-X_253) )
```

```
##      [,1]
## [1,] 1950
```

Now to compute all the distances at once, we have the function `dist`.

```
d <- dist(X)
class(d)
```

```
## [1] "dist"
```

Note that this produces an object of class `dist` and, to access the entries using row and column indices, we need to coerce it into a matrix:

```
as.matrix(d)[1,2]
```

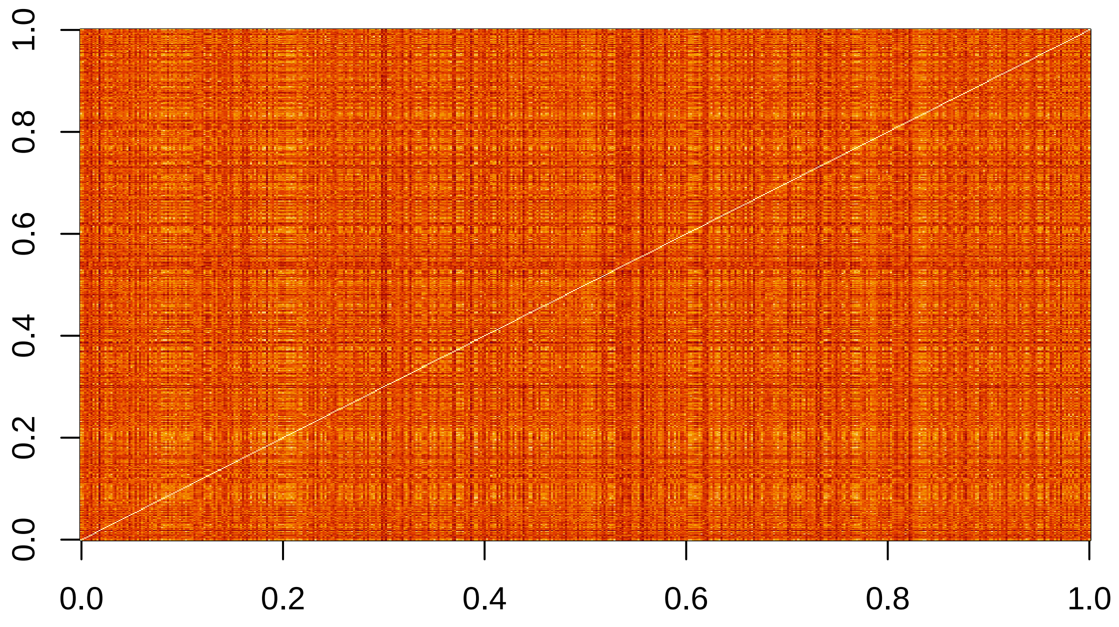
```
## [1] 1935
```

```
as.matrix(d)[1,253]
```

```
## [1] 1950
```

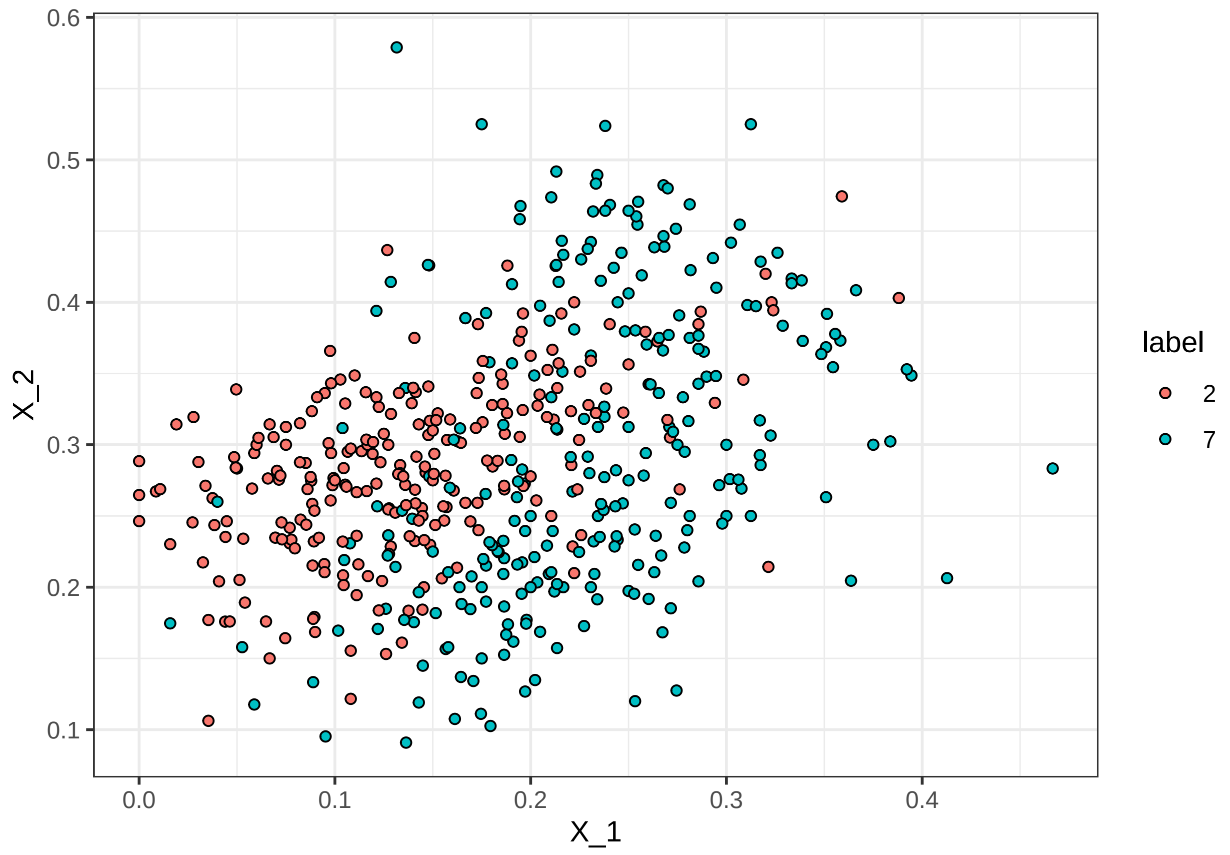
We can quickly see an image of these distances

```
image(as.matrix(d))
```



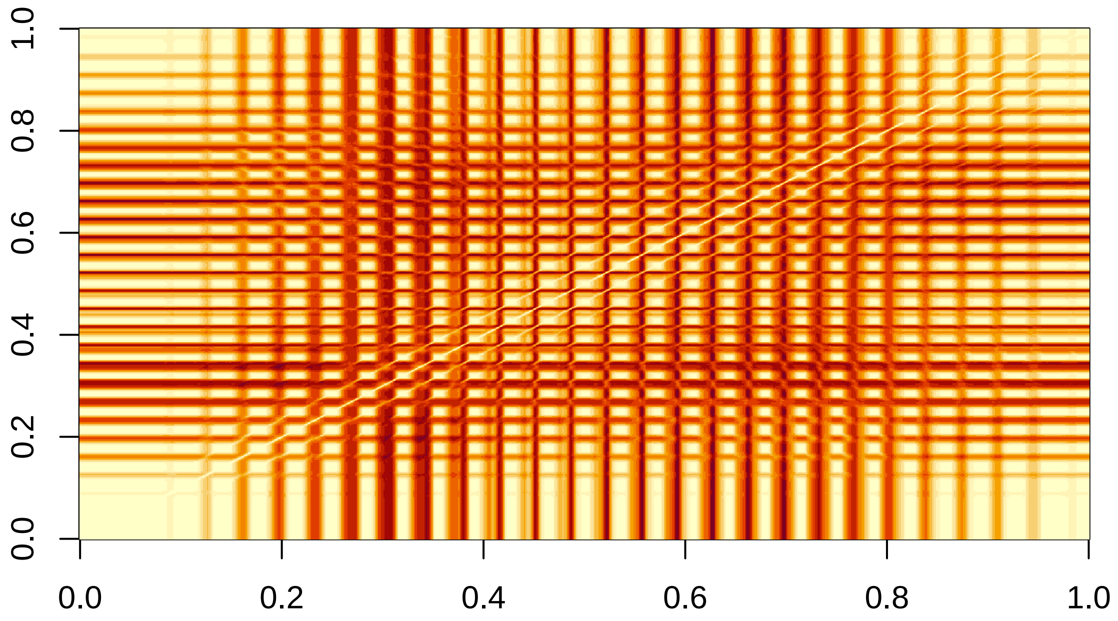
Note that for illustrative purposes we defined two predictors. Defining distances between observations based on these two covariates is much more intuitive since we can simply visualize the distance in a two dimensional plot

```
ggplot(train_set) +
  geom_point(aes(X_1, X_2, fill=label), pch=21)
```



Distance between predictors Perhaps a more interesting result comes from computing distance between predictors:

```
image(as.matrix(dist(t(X))))
```



k Nearest Neighbors

K-nearest neighbors (kNN) is easier to adapt to multiple dimensions. We first define the distance between all observations based on the features. Basically, for any point $\mathbf{x}$ for which we want an estimate of $p(\mathbf{x})$, we look for the k nearest points and then take an average of these points. This gives us an estimate of $p(x_1, x_2)$. We can now control flexibility through k.

Let's use our logistic regression as a baseline:

```
library(caret)
glm_fit <- glm(y~.,data = select(train_set, y, X_1, X_2) )
f_hat <- predict(glm_fit, newdata = test_set,
                 type = "response")
tab <- table(pred=round(f_hat), truth=test_set$y)
confusionMatrix(tab)$tab
```

```
##      truth
## pred   0   1
##    0 169  45
##    1  74 212
```

```
confusionMatrix(tab)$overall["Accuracy"]
```

```
## Accuracy
##    0.762
```

Now, lets compare to kNN. Let's start with the default $k = 5$

```
knn_fit <- knn3(y~.,data = select(train_set, y, X_1, X_2) )
f_hat <- predict(knn_fit, newdata = test_set)[,2]
tab <- table(pred=round(f_hat), truth=test_set$y)
confusionMatrix(tab)$tab
```

```
##       truth
## pred    0   1
##    0 179  51
##    1  64 206
```

```
confusionMatrix(tab)$overall["Accuracy"]
```

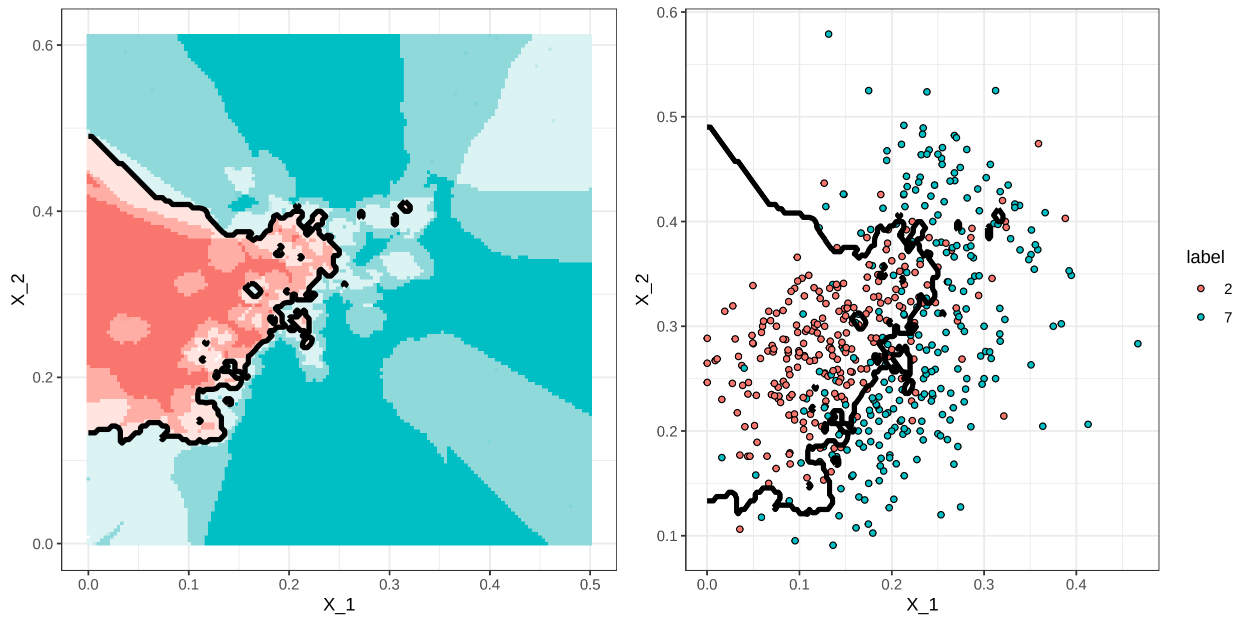
```
## Accuracy
##     0.77
```

This already improves accuracy over the logistic model. Let's see why this is:

```
## Warning: The ‘<scale>‘ argument of ‘guides()‘ cannot be ‘FALSE‘. Use "none" instead as
## of ggplot2 3.3.4.
```

```
## Warning: Using ‘size‘ aesthetic for lines was deprecated in ggplot2 3.4.0.
## i Please use ‘linewidth‘ instead.
```

```
## Warning: The following aesthetics were dropped during statistical transformation: fill
## i This can happen when ggplot fails to infer the correct grouping structure in
##   the data.
## i Did you forget to specify a ‘group‘ aesthetic or to convert a numerical
##   variable into a factor?
```



When $k = 5$, we see some islands of red in the blue area. This is due to what we call *over training*. Note that we have higher accuracy in the train set compared to the test set:

```r
f_hat <- predict(knn_fit, newdata = test_set)[,2]
tab <- table(pred=round(f_hat), truth=test_set$y)
confusionMatrix(tab)$overall["Accuracy"]
```

```
## Accuracy
##     0.77
```

```r
f_hat_train <- predict(knn_fit, newdata = train_set)[,2]
tab <- table(pred=round(f_hat_train), truth=train_set$y)
confusionMatrix(tab)$overall["Accuracy"]
```

```
## Accuracy
##    0.842
```

Over-training

Over-training is at its worse when we set $k = 1$. In this case we will obtain perfect accuracy in the training set because each point is used to predict itself. So perfect accuracy must happen by definition. However, the test set accuracy is actually worse than logistic regression.

```r
knn_fit_1 <- knn3(y~.,data = select(train_set, y, X_1, X_2), k=1)
f_hat <- predict(knn_fit_1, newdata = train_set)[,2]
tab <- table(pred=round(f_hat), truth=train_set$y)
confusionMatrix(tab)$overall["Accuracy"]
```

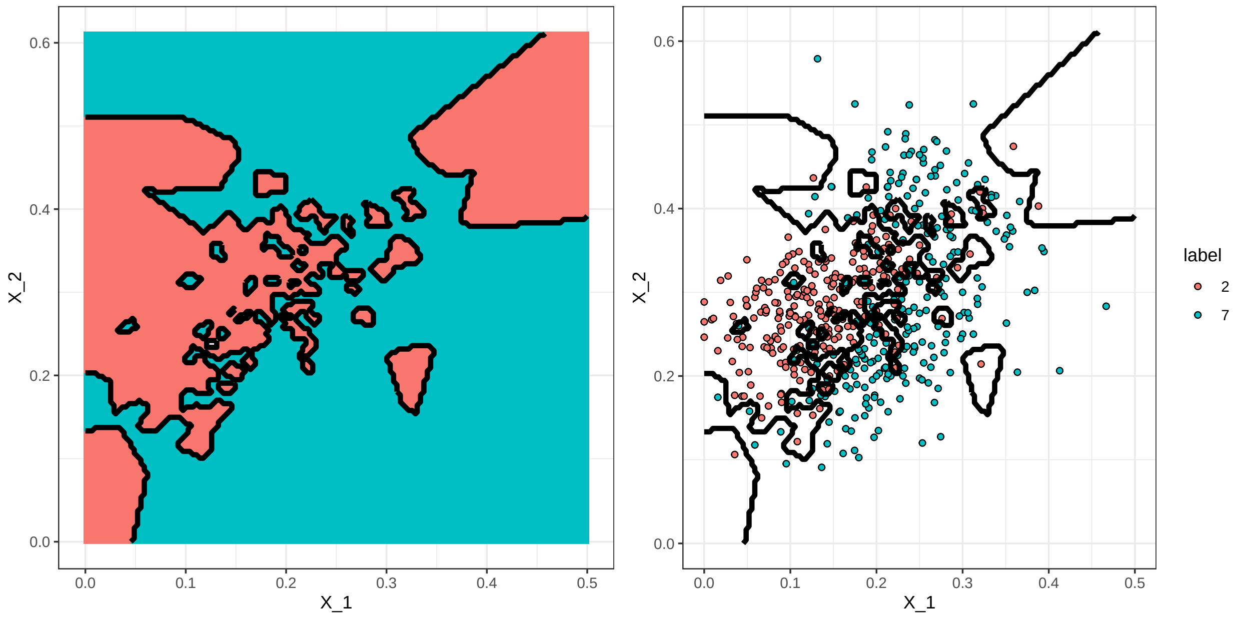
```
## Accuracy
##    0.998
```

```r
f_hat <- predict(knn_fit_1, newdata = test_set)[,2]
tab <- table(pred=round(f_hat), truth=test_set$y)
confusionMatrix(tab)$overall["Accuracy"]
```

```
## Accuracy
##    0.726
```

We can see the over-fitting problem in this figure:

```
## Warning: The following aesthetics were dropped during statistical transformation: fill
## i This can happen when ggplot fails to infer the correct grouping structure in
##   the data.
## i Did you forget to specify a ‘group‘ aesthetic or to convert a numerical
##   variable into a factor?
```



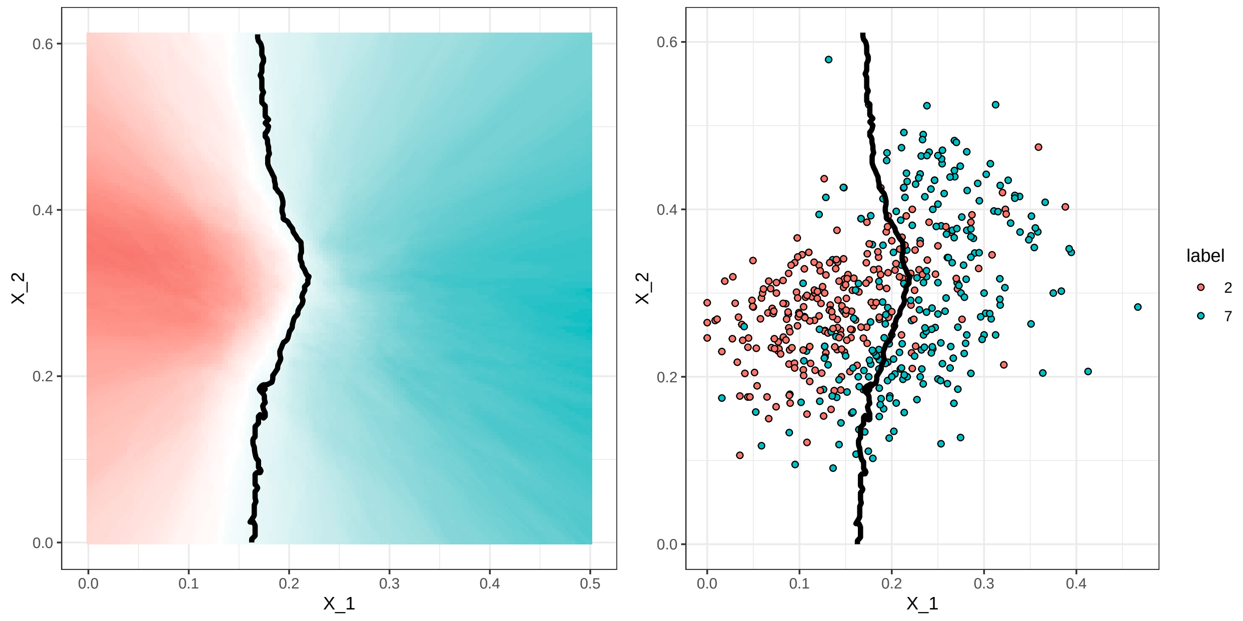
We can also go *over-smooth*. Look at what happens with 251 closest neighbors:

```
knn_fit_251 <- knn3(y~.,data = select(train_set, y, X_1, X_2), k=251)
f_hat <- predict(knn_fit_251, newdata = test_set)[,2]
tab <- table(pred=round(f_hat), truth=test_set$y)
confusionMatrix(tab)$overall["Accuracy"]
```

```
## Accuracy
##    0.804
```

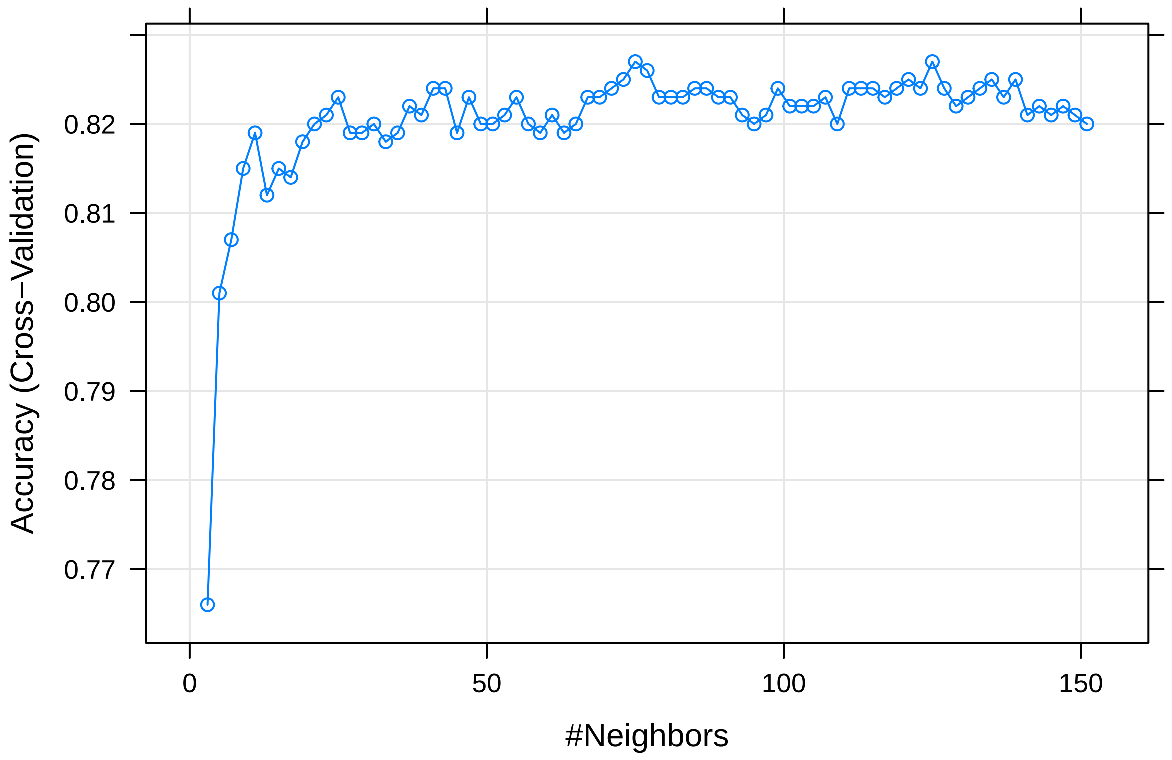
This turns out to be similar to logistic regression:

```
## Warning: The following aesthetics were dropped during statistical transformation: fill
## i This can happen when ggplot fails to infer the correct grouping structure in
##   the data.
## i Did you forget to specify a ‘group‘ aesthetic or to convert a numerical
##   variable into a factor?
```



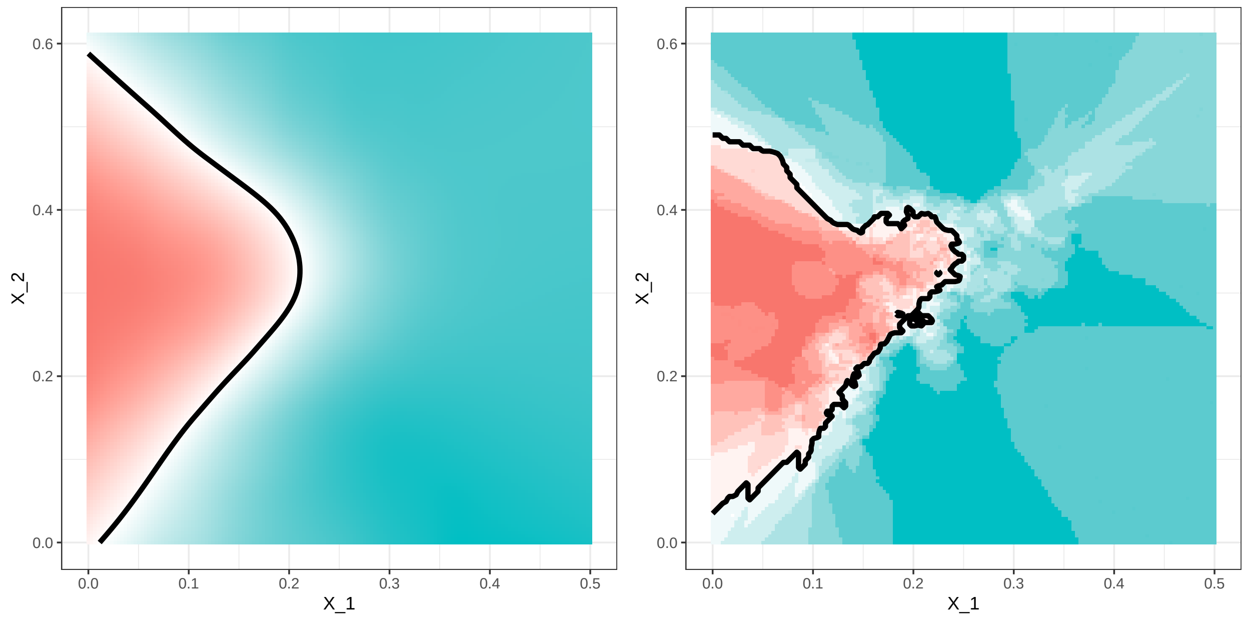
Let's plot the accuracy for different numbers of closest neighbors.

```
control <- trainControl(method='cv', number=2, p=.5)
dat2 <- mutate(dat, label=as.factor(label)) %>%
  select(label,X_1,X_2)
res <- train(label ~ .,
             data = dat2,
             method = "knn",
             trControl = control,
             tuneLength = 1, # How fine a mesh to go on grid
             tuneGrid=data.frame(k=seq(3,151,2)),
             metric="Accuracy")
plot(res)
```

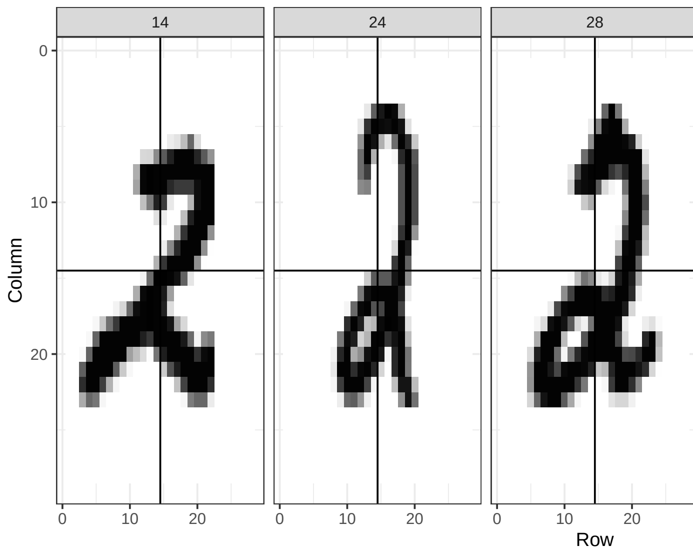


With $k = 11$ we obtain what appears to be a decent estimate of the true f.

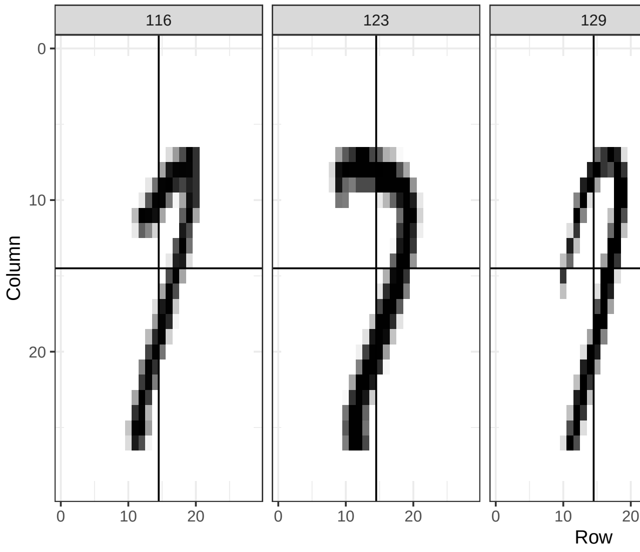
```
## Warning: The following aesthetics were dropped during statistical transformation: fill
## i This can happen when ggplot fails to infer the correct grouping structure in
##   the data.
## i Did you forget to specify a ‘group‘ aesthetic or to convert a numerical
##   variable into a factor?
## The following aesthetics were dropped during statistical transformation: fill
## i This can happen when ggplot fails to infer the correct grouping structure in
##   the data.
## i Did you forget to specify a ‘group‘ aesthetic or to convert a numerical
##   variable into a factor?
```



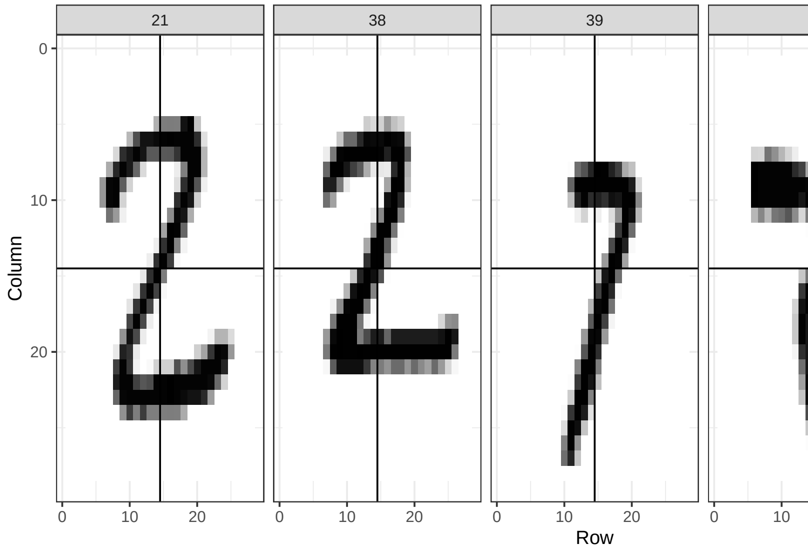
An important part of data science is visualizing results to determine why we are succeeding and why we are failing.



Here are some 2s that were correctly called with high probability:

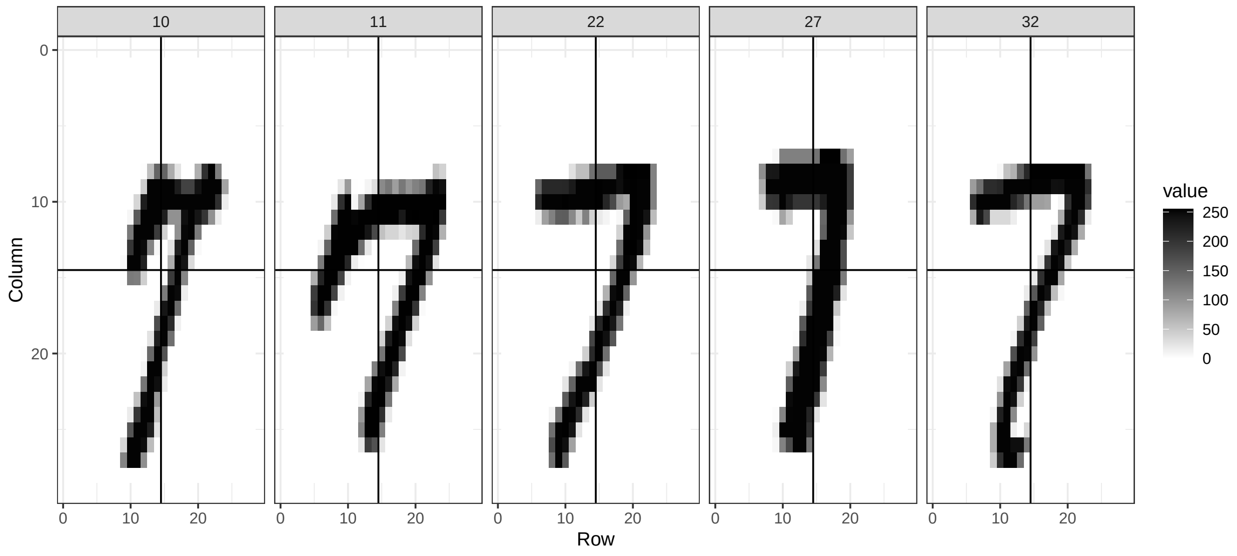


Here are some 7s that were incorrectly called 2s with high probability:

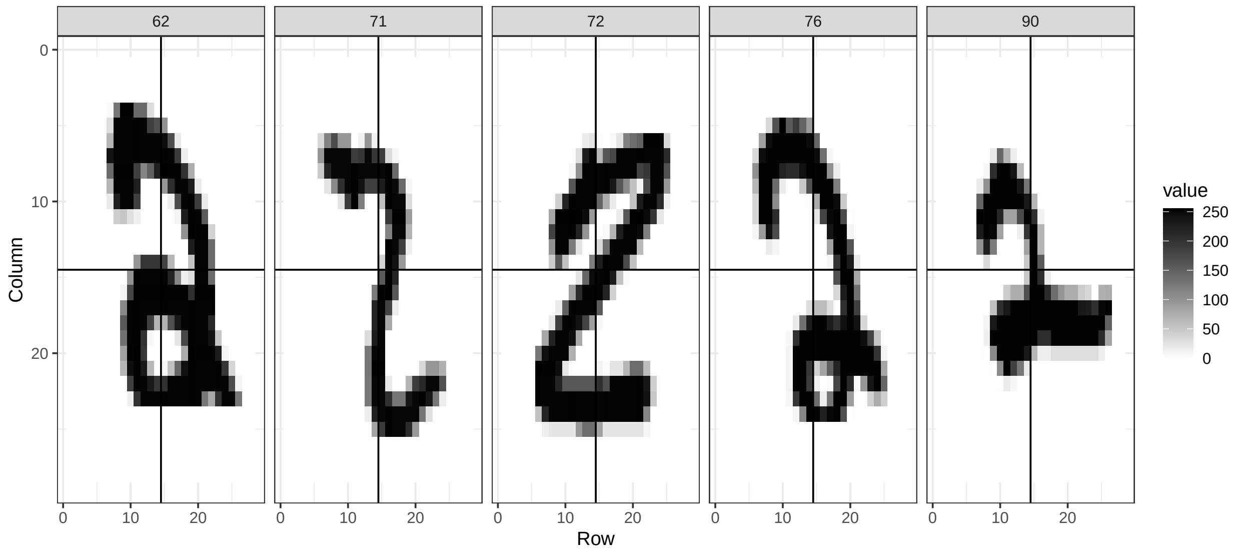


Here are some for which the predictor was about 50-50:

Here are some 7s that were correctly called with high probability:



Here are some 2s that were incorrectly called with high probability:


Column
Row
value