Challenge 1

Mitchell Ardolf, Evan Weibe, Adam Vazquez Rosales

10/13/2023

```
# Given functions

# Plots a digit as an image
plot_digit <- function(image) {
  a_digit <- matrix(image, nrow=28)
  image(a_digit[,28:1])
}

# This function returns a matrix with all the images corresponding to a digit (0-9)
# The matrix will have 784 columns and the number of rows corresponds to the
# number of images
get_image_digit_train <- function (digit) {
  idx <- mnist$train$labels==digit
  images <- mnist$train$images[idx,]
  return (images)
}


#This function gets a matrix of images from the testing data set
get_image_digit_test <- function (digit) {
  idx <- mnist$test$labels==digit
  images <- mnist$test$images[idx,]
  return (images)
}

#contrast digit takes a digit and contrasts it such that anything darker than 128 is now 255 and anthin
contrast_digit <- function(image){
  dark_idx = (image>128)
  image[dark_idx]=255    # Turn dark pixel indexes to 255
  image[!dark_idx]=0     # Turn non-dark pixel indexes to 0 (we use symbol !)
  return (image)
}
```

Feature Definition.

The first feature we plan to use is the width of the number. This is the distance betweeen the left border and the right border of the image. The borders are calculated by being the furthest out rows in our matrix where there is a sum of values greater than 0. We think that 4s will have a larger width than the 1s.

```
  # The function takes an image and looks at all columns and those columns
  # that contain pixels are counted.
get_width <- function(image){
  a_digit <- matrix(image, nrow = 28)
  adigit <- a_digit[, 28:1]
  i <- 1
  j <- 0

  while (i <= 28) {
    if (sum(adigit[i, ]) > 0) {
      j <- j + 1
    }

    i <- i + 1
  }
  return(j)

}

#get size runs the get get_width() function of a matrix of images and returns a vector of sizes
get_size <- function(images){

  n_images <- nrow(images)


  size_val <- vector(mode="integer", length=n_images)
  for (i in 1:n_images) {
    size_val[i] = get_width(images[i,])
  }
  return (size_val)
}
```

The second feature we are using is the amount of dark pixels within the upper half. This is calculated my counting the number of dark pixels in the first half of the vector (1 to 392). We think that fours will have a larger number of dark pixels in the top half since they typically have 2 vertical lines in the top half, while 1s typically only have the one line in the upper half.

```
# This function counts the dark pixels in a given image
count_dark_pixels <- function (image){
  dark_idx <- image>128
  num_pixels <- sum(dark_idx)
  return(num_pixels)
}

# get_num_pixels runs count_dark_pixels() where given a matrix of images it
# counts the number of dark pixels in each image and returns the values as a vector.
get_num_pixels <- function(images){
  n_images <- nrow(images)
  dark <- vector(mode="integer", length=n_images)
  # We use a for loop to calculate the top half dark pixels.
  for (i in 1:n_images) {
    dark[i] = count_dark_pixels(images[i, 1:392])
  }
```

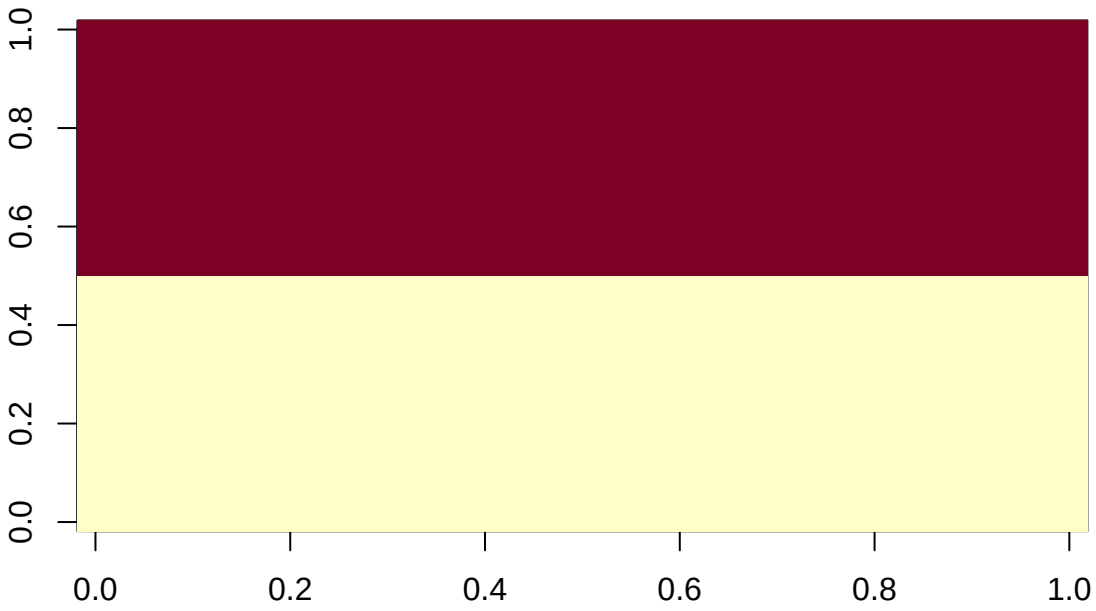
```
  return (dark)
}
```

Testing Features.

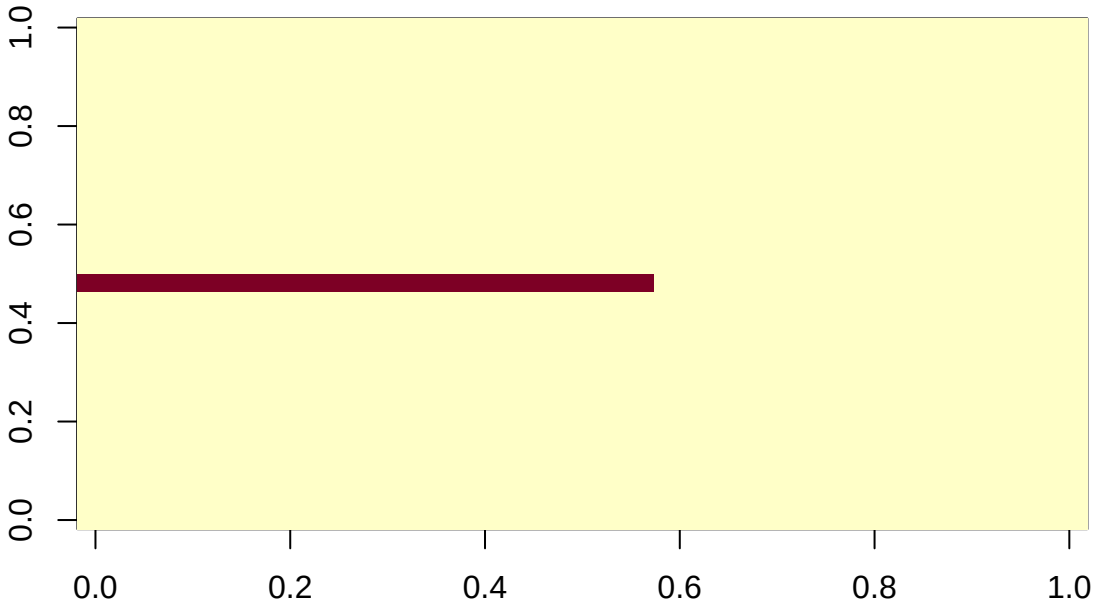
We wanted to specifically make sure our functions were working. To do this we created a two row matrix to represent a two images to test our functions. The first image was one with a grey scale value of 255 throughout the top half and greyscale value of 0 on the bottom half. This was to test our count_dark_pixels() function to make sure it could count all the dark pixels. Our function does count every pixel in the top half. We also tested this one with the get_width() function to make sure that said function could track track the width of the dark pixels forming the number (in this case 28). This result is what we expected.

For the second image we wanted to test the get_width() function to make sure that it could handle widths of under 28 and work as correctly intended. We also wanted to make sure our count_dark_pixels() function was tracking the correct pixels. That is why our second image the top half has no dark pixels while the bottom half has only a line 16 dark pixels wide. If the count_dark_pixels() function wasn't functioning correctly, these 16 pixels would be accounted for in "get_num_pixels(m)[2]".

```
# Test to make sure first 392 is the top half.

#Image with top have at 255 and bottom half at 0 for grey scale values
vectortest <- vector(mode = "integer", length = 784)
for (i in 1:392){
  vectortest[i] <- 255

}

#Image with top have at 0 and bottom half at 255 for grey scale values
vectortest2 <- vector(mode = "integer", length = 784)
for (i in 393:408){
  vectortest2[i] <- 255

}

# Matrix of both of our testing vectors to mimic the format of mmist$images.
m <- rbind(vectortest, vectortest2)

# Testing graphs.
# We decide to use a vector of complete darkness in the top half and dark in a
# partial area of the bottom half.
plot_digit(vectortest)
```




```
plot_digit(vectortest2)
```




```
# Testing width calculation.
get_width(vectortest)
```

```
## [1] 28
```

```
get_width(vectortest2)
```

```
## [1] 16
```

```
get_size(m)
```

```
## [1] 28 16
```

```
# Testing dark pixels top half.
# We include the 1:392 truncation since in count_dark_pixels() since truncation
# occurs in the get_num_pixels() function to limit our observations to the top half.
count_dark_pixels(vectortest[1:392])
```

```
## [1] 392
```

```
count_dark_pixels(vectortest2[1:392])
```

```
## [1] 0
```

```
get_num_pixels(m)
```

```
## [1] 392   0
```

```
get_num_pixels(m)[2]
```

```
## [1] 0
```

Dataset Creation.

Below is the code for our dataset creation and sampling. We calculate our features for all ones and all fours, not just our sample. This allows us to keep our calculated features in the same order as our ones and fours when we sample them later. We do calculate the contrasted image features for "Changing things up (I)", however, those columns are not used until that section. We additionally pull the row index by using for loops.

Initially the data is in matrix form. After identifying our features the data is stored into separate vectors each for a feature. Four data frames are formed from these vectors. We create a testing and training dataset using bind_rows() to combine our data frames. These bound tables are then sampled from after we set the seed to guarantee that our results are reproducible. Finally, CSVs are generated for later reference.

This exact format is later utilized when sampling sixes for our dataset in "Changing things up (II)"..

```
set.seed(123)

# This function gets the count of dark pixels for the top half of contrasted images.
# We are writing this now to have contrasted image values within the same table, even
# though the "Changing things up (I)" still has awhile to wait.
count_dark_pixels_contrast <- function(images){
  # We get all images corresponding to a digit
  n_images <- nrow(images)

  dark <- vector(mode="integer", length=n_images)
  for (i in 1:n_images) { # the for loop iterates through the images
    dark[i] = count_dark_pixels(contrast_digit(images[i, 1:392]))
  }
  return (dark)
}




# This function gets the width of contrasted images.
# We are writing this now to have contrasted image values within the same table, even
# though the "Changing things up (I)" still has awhile to wait.
get_size_contrast <- function(images){

  n_images <- nrow(images)

  size_val <- vector(mode="integer", length=n_images)
  for (i in 1:n_images) {
```

```
    size_val[i] = get_width(contrast_digit(images[i,])) #width of the image
  }
  return (size_val)
}

#Get all ones and fours for training.
image1_train <- get_image_digit_train(1)
image4_train <- get_image_digit_train(4)

# Get row numbers (indexes) for 1s and 4s in training.
train_count_1_bool <- mnist$train$labels==1 # Get boolean of 1s in train.
length1_train <- sum(train_count_1_bool) # Get number of 1s for the loop.
idx1_train <- vector(mode = "integer", length = length1_train)
n=1
for (i in 1:60000){

  if (train_count_1_bool[i] == TRUE){
    if (idx1_train[n] != 0){
      n = n+1
    }
    idx1_train[n] <- i

  }

}

train_count_4_bool <- mnist$train$labels==4 # Get boolean of 4s in train.
length4_train <- sum(train_count_4_bool) # Get number of 4s for the loop.
idx4_train <- vector(mode = "integer", length = length4_train)
n=1
for (i in 1:60000){

  if (train_count_4_bool[i] == TRUE){
    if (idx4_train[n] != 0){
      n = n+1
    }
    idx4_train[n] <- i

  }

}


#Get all ones and fours for testing.
image1_test <- get_image_digit_test(1)
image4_test <- get_image_digit_test(4)

# Get row numbers (indexes) for 1s and 4s in training.
test_count_1_bool <- mnist$test$labels==1 # Get boolean of 1s in test.
length1_test <- sum(test_count_1_bool) # Get number of 1s for the loop.
idx1_test <- vector(mode = "integer", length = length1_test)
n=1
for (i in 1:10000){
```

```
  if (test_count_1_bool[i] == TRUE){
    if (idx1_test[n] != 0){
      n = n+1
    }
    idx1_test[n] <- i

  }

}

test_count_4_bool <- mnist$test$labels==4 # Get boolean of 4s in test.
length4_test <- sum(test_count_4_bool) # Get number of 4s for the loop.
idx4_test <- vector(mode = "integer", length = length4_test)
n=1
for (i in 1:10000){

  if (test_count_4_bool[i] == TRUE){
    if (idx4_test[n] != 0){
      n = n+1
    }
    idx4_test[n] <- i

  }

}


# Calculate features for training get the number of pixels for ones.

image_1_dark_train <- get_num_pixels(image1_train) # Dark pixels top half
# Dark pixels top half on the contrasted digit.
image_1_dark_cont_train <- count_dark_pixels_contrast(image1_train)

#Calculate features for training get width of the image for ones.
image_1_size_train <- get_size(image1_train)# Width for ones.
# Width for contrasted ones.
image_1_size_cont_train <- get_size_contrast(image1_train)

#Calculate features for testing get the number of pixels for ones.
image_1_dark_test <- get_num_pixels(image1_test)# Dark pixels top half.
# Dark pixels top half on the contrasted digit.
image_1_dark_cont_test <- count_dark_pixels_contrast(image1_test)

#Calculate features for testing get size of the image for ones.
image_1_size_test <- get_size(image1_test)# Width for ones.
# Width for contrasted ones.
image_1_size_cont_test <- get_size_contrast(image1_test)


#Combine the tables to get our training and testing tables.
combined_table_1_train <- data.frame(Size = image_1_size_train,
                                     Size_Contrast = image_1_size_cont_train,
                                     Dark = image_1_dark_train,
```

```
                                    Dark_Contrast = image_1_dark_cont_train,
                                    Number = 1,
                                    idx = idx1_train)

combined_table_1_test <- data.frame(Size = image_1_size_test,
                                    Size_Contrast = image_1_size_cont_test,
                                    Dark = image_1_dark_test,
                                    Dark_Contrast = image_1_dark_cont_test,
                                    Number = 1,
                                    idx = idx1_test)


#Calculate features for training get the number of pixels for fours.
image_4_dark_train <- get_num_pixels(image4_train)# Dark pixels top half.
# Dark pixels top half on the contrasted digit.
image_4_dark_cont_train <- count_dark_pixels_contrast(image4_train)

#Calculate features for training get size of the image for fours.
image_4_size_train <- get_size(image4_train)# Width for fours.
# Width for contrasted fours.
image_4_size_cont_train <- get_size_contrast(image4_train)

#Calculate features for testing get the number of pixels for fours.
image_4_dark_test <- get_num_pixels(image4_test)# Dark pixels top half.
# Dark pixels top half on the contrasted digit.
image_4_dark_cont_test <- count_dark_pixels_contrast(image4_test)

#Calculate features for testing get size of the image for fours.
image_4_size_test <- get_size(image4_test)# Width for fours.
# Width for contrasted fours.
image_4_size_cont_test <- get_size_contrast(image4_test)

#Combine the tables to get our training and testing tables.
combined_table_4_train <- data.frame(Size = image_4_size_train,
                                     Size_Contrast = image_4_size_cont_train,
                                     Dark = image_4_dark_train,
                                     Dark_Contrast = image_4_dark_cont_train,
                                     Number = 4,
                                     idx = idx4_train)

combined_table_4_test <- data.frame(Size = image_4_size_test,
                                    Size_Contrast = image_4_size_cont_test,
                                    Dark = image_4_dark_test,
                                    Dark_Contrast = image_4_dark_cont_test,
                                    Number = 4,
                                    idx =idx4_test)

# Combine the 1s and 4s.
combined_train <- bind_rows(combined_table_1_train, combined_table_4_train)
combined_test <- bind_rows(combined_table_1_test, combined_table_4_test)

# Sample testing and training.
set.seed(123)
```

```r
sampled_data_train <- combined_train %>% sample_n(800, replace = FALSE)
sampled_data_test <- combined_test %>% sample_n(200, replace = FALSE)

# Create File Paths for CSVs.
file_path_test <- "~/Mscs 341 F23/Project/Mitch, Evan, Adam/Data/sampled_data_test.csv"
file_path_train <- "~/Mscs 341 F23/Project/Mitch, Evan, Adam/Data/sampled_data_train.csv"

# Write the files.
write.csv(sampled_data_train, file = file_path_train)
write.csv(sampled_data_test, file = file_path_test)
```

Model Creation.

```r
# Data for the models.

# These renames accommodate for the model building, which was initially written
# in different files, which pulled the CSVs, then renamed the testing data to
# raw_test and the training data to raw_train.

# Testing data set.
raw_test <- sampled_data_test

# Training data set.
raw_train <- sampled_data_train
```

KKNN.

```r
# Data manipulation.
# We need the output as a factor for classification.
test_tbl <- raw_test %>%
  mutate(Number = as.factor(Number))

train_tbl <- raw_train %>%
  mutate(Number = as.factor(Number))
```

Before generating our final KNN model, we optimize our k parameter using a for loop and error calculation function. This function outputs the Misclassification given a value of k. We find that 5 is our optimal value of k. We then use the tidymodels format to build our optimized KNN model.

First, we highlight the plausible ties in our dataset. Within the process of finding the optimal k value we encountered issues regarding multiple optimal k values and consistency of accuracy across runs. Initially we used a knn3 model to predict the accuracy and thus find the k value. This produced multiple different K values every run and was inconsistent. If there are ties, knn3 will select randomly to keep the amount at the specified k. To handle this we changed our approach we ended up using using tidy models in a kknn model structure to reduce this problem. We do still have multiple optimal k values in the kknn model, however this is solved with some tidying. After that, we build our optimal model.

```r
train_tbl %>%
  group_by(Size, Dark) %>%
  summarise(n = n()) %>%
  arrange(desc(n)) %>%
  slice(1:5)
```

```
## `summarise()` has grouped output by 'Size'. You can override using the
## `.groups` argument.
```

```
## # A tibble: 85 x 3
## # Groups:   Size [17]
##     Size  Dark     n
##    <dbl> <int> <int>
##  1     4    20     6
##  2     4    19     5
##  3     4    24     5
##  4     4    16     3
##  5     4    18     3
##  6     5    20     3
##  7     5    26     3
##  8     5    16     2
##  9     5    18     2
## 10     5    25     2
## # i 75 more rows
```

```r
Number_Of_Plausible_Ties <- train_tbl %>%
  group_by(Size, Dark) %>%
  summarise(n = n(), .groups = "drop") %>%
  filter(n > 1)%>%
  nrow()

Number_Of_Plausible_Ties
```

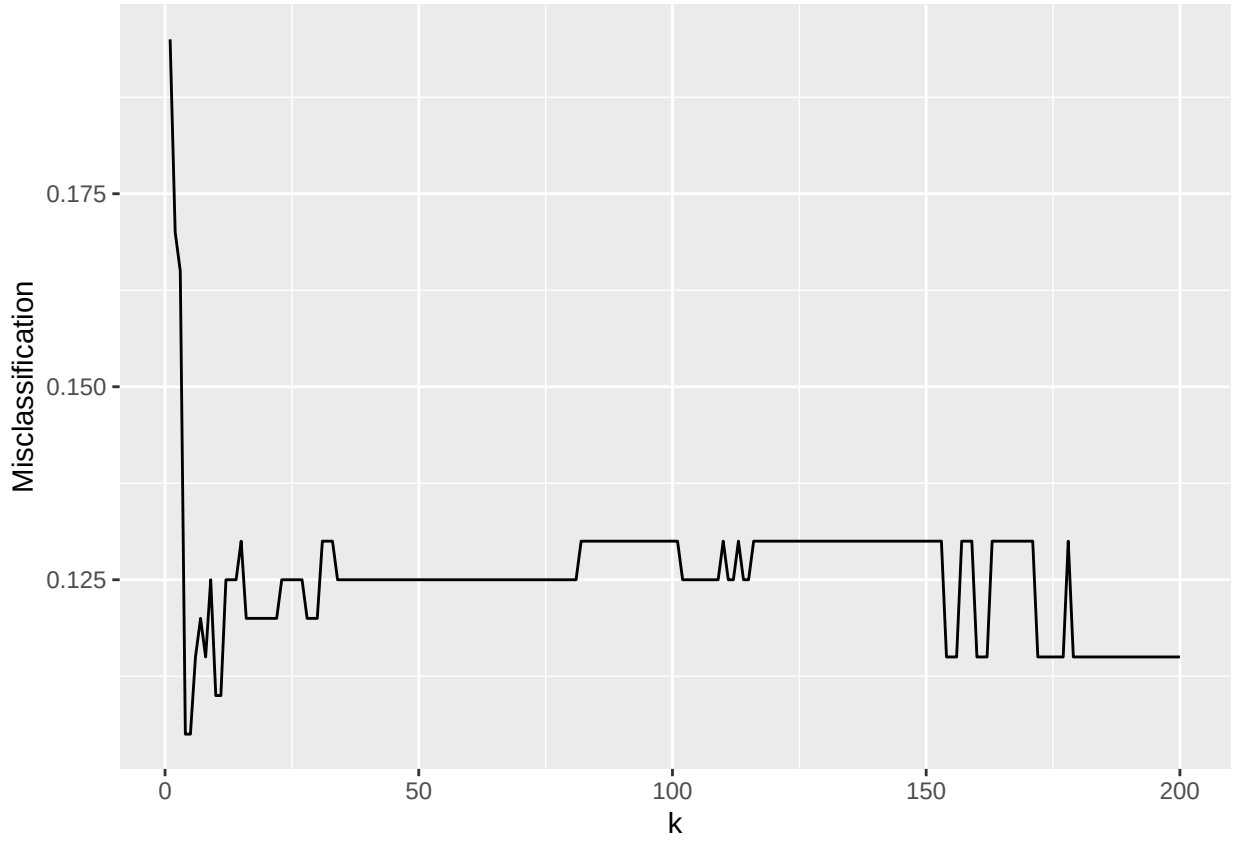
```
## [1] 191
```

```r
set.seed(128)
# Model Building.

# Calculate error rate with given k.
calc_error <- function(kNear, train_tbl, test_tbl) {

  NumberRecipe <-
  recipe(Number ~ Size + Dark, data=train_tbl)

  Knn_model <- nearest_neighbor(neighbors = kNear) %>%
  set_engine("kknn") %>%
  set_mode("classification")

  Knn_wflow <- workflow() %>%
  add_recipe(NumberRecipe) %>% # Mix the recipe and model.
  add_model(Knn_model)
```

```
  Knn_fit_digit <- fit(Knn_wflow, train_tbl)

  augmented_test <-augment(Knn_fit_digit, test_tbl)%>%
    accuracy(Number, .pred_class)

  return((1-augmented_test$.estimate))
}

# Test many values of k to optimize the model.
error_test <- vector(length = nrow(test_tbl))
for(i in 1:nrow(test_tbl)){
  error_test[i] <- calc_error(i, train_tbl, test_tbl)
}
```

```
##
## Attaching package: 'kknn'

## The following object is masked from 'package:caret':
##
##     contr.dummy
```

```
error_tbl <- tibble (k=1:nrow(test_tbl),
                     Misclassification = error_test)

# Plot the error as k varies.
error_tbl %>%
  ggplot(aes(x=k, y=Misclassification)) +
  geom_line()
```




```
# Find which k minimizes the misclassification. rate (maximize the accuracy).
min_k_tbl <- error_tbl %>%
  slice_min(Misclassification) %>%
  arrange(desc(k)) %>%
  slice(1)

min_k_tbl
```

```
## # A tibble: 1 x 2
##       k Misclassification
##   <int>             <dbl>
## 1     5             0.105
```

```
# Set our recipe.
NumberRecipe <-
  recipe(Number ~ Size + Dark, data=train_tbl)

# Set our model type.
Knn_model <- nearest_neighbor(neighbors = min_k_tbl$k) %>%
  set_engine("kknn") %>%
  set_mode("classification")

# Set our work flow.
Knn_wflow <- workflow() %>%
  add_recipe(NumberRecipe) %>% # Mix the recipe and model.
```

```
  add_model(Knn_model)

# Fit the model.
Knn_fit_digit <- fit(Knn_wflow, train_tbl)

# Table to evaluate our model.
augmented_test <-augment(Knn_fit_digit, test_tbl)

# Accuracy.
augmented_test%>%
  accuracy(Number, .pred_class)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.895
```

Logistic regression.

We use tidymodels to build our logistic regression model. There are no parameters to optimize.

```
# Data manipulation.
# We need the output as a factor for classification.
test_tbl_logit <- raw_test %>%
  mutate(Number = as.factor(Number))

train_tbl_logit <- raw_train %>%
  mutate(Number = as.factor(Number))
```

```
# Model building.

# Set our recipe.
NumberRecipe_logit <-
  recipe(Number ~ Size + Dark, data=train_tbl_logit)

# Set our model type.
logit_model <- logistic_reg() %>%
  set_engine("glm") %>% # Decide engine.
  set_mode("classification") # We are wanting to use classification here.

# Work flow.
logit_wflow <- workflow() %>%
  add_recipe(NumberRecipe_logit) %>% # Mix recipe and model.
  add_model(logit_model)

# Fit the model on our training data set.
logit_fit_digit <- fit(logit_wflow, train_tbl_logit)

# Table to evaluate our model on our testing data set.
augmented_test_logit <- augment(logit_fit_digit, test_tbl)

# Accuracy
```

```
augmented_test_logit%>%
  accuracy(Number, .pred_class)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.895
```

Evaluate the Chosen Model.

Since the accuracy of our logic regression model and kknn model exactly the same we chose to work with he logistic regression model due to the opportunity to get more aquatinted with multinomial logistic regression.

```
# Accuracy, Logistic Regression.
acc_log<-augmented_test_logit%>%
  accuracy(Number, .pred_class)

# Accuracy, KNN.
acc_knn<-augmented_test%>%
  accuracy(Number, .pred_class)

# Calculate Misclassification Rate.
mcr_log <- 1 - acc_log$.estimate
mcr_knn <- 1 - acc_knn$.estimate

mcr_df = data.frame(knn = mcr_knn, logistic.regression = mcr_log)

# Misclassification Rates for both models.
mcr_df
```

```
##     knn logistic.regression
## 1 0.105               0.105
```

Below is the confusion matrix for the Logistic Regression Model. Ones that are predicted as a four are a more common misclassification in our model than fours misclassified as ones.

```
augmented_test_logit%>%
  conf_mat(Number, .pred_class)
```

```
##           Truth
## Prediction   1   4
##          1 104   5
##          4  16  75
```

Visualization.

Probabilities and the Decision Boundary

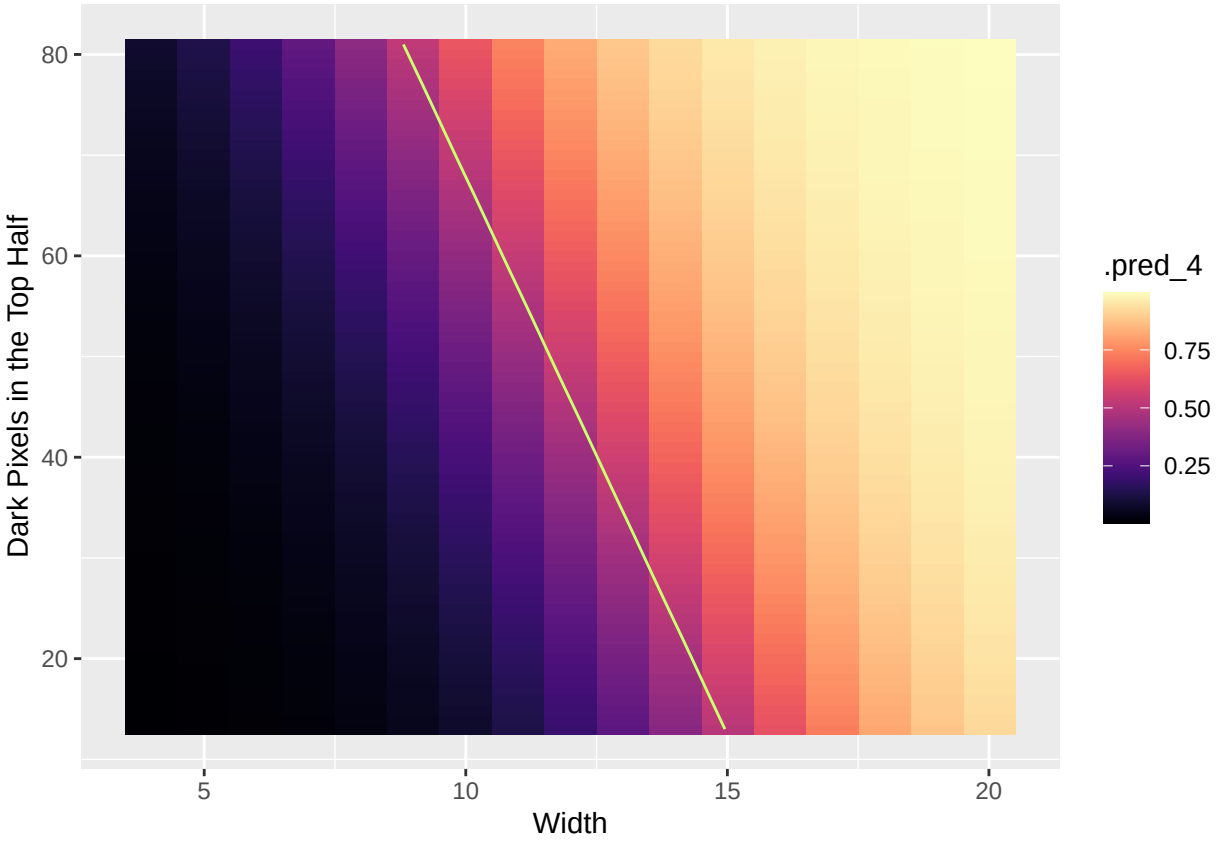
```r
# Decision boundary

#Find range of size.
range(augmented_test_logit$Size)
```

```
## [1]  4 20
```

```r
# gridvec1 as range of size (unit of 1 increase).
gridvec1 <- c(4:20)

#Find range of dark pixels top half.
range(augmented_test_logit$Dark)
```

```
## [1] 13 81
```

```r
# gridvec2 as range of size (unit of 1 increase).
gridvec2 <- c(13:81)

# Grid of theoretical values is created by every possible combination
# of our 2 ranges.
grid_tbl <- expand_grid(Size=gridvec1, Dark=gridvec2)

# Get the graph of our probabilities and the decision boundary.
augment(logit_fit_digit, grid_tbl)%>% # Percentages
ggplot(mapping = aes(x = Size, y = Dark, z = .pred_4, fill = .pred_4)) +
  geom_raster() + # Assigns color to each square of the grid.
  stat_contour(breaks=c(0.5), color="darkolivegreen1") + #Bayes/Decision boundary
  scale_fill_viridis(option="magma") +
  labs(x= "Width",
       y = "Dark Pixels in the Top Half")
```

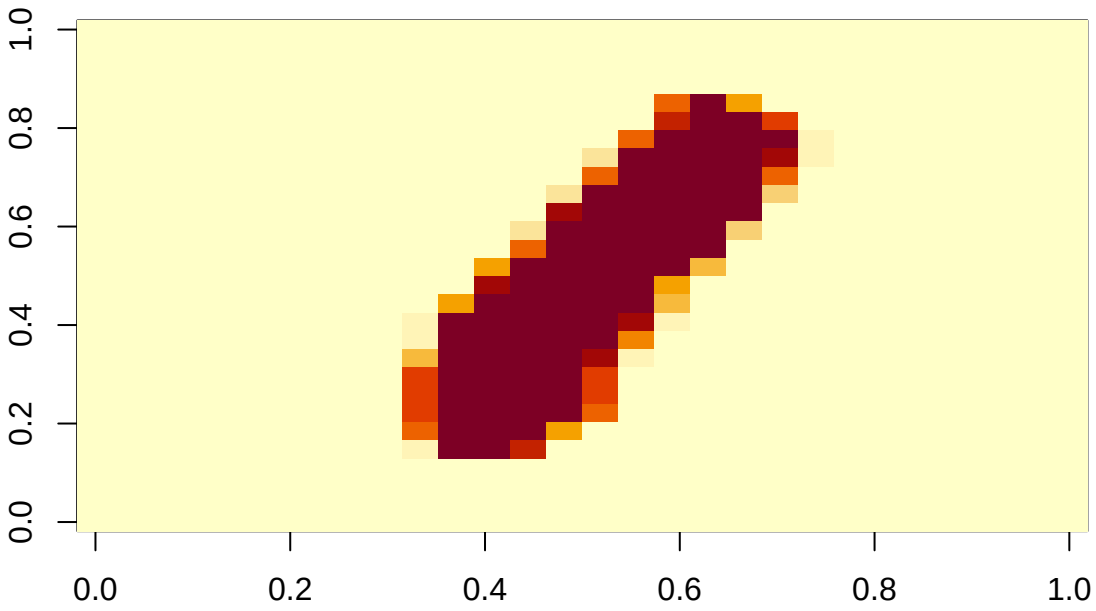


The decision boundary shows that digits with higher width and more dark pixels in the top half are classified as fours. The reverse is true for ones. When the digit has a low width and low amount of dark pixels in the top half, our logistic regression model classifies it as one.

Misclassified Digits

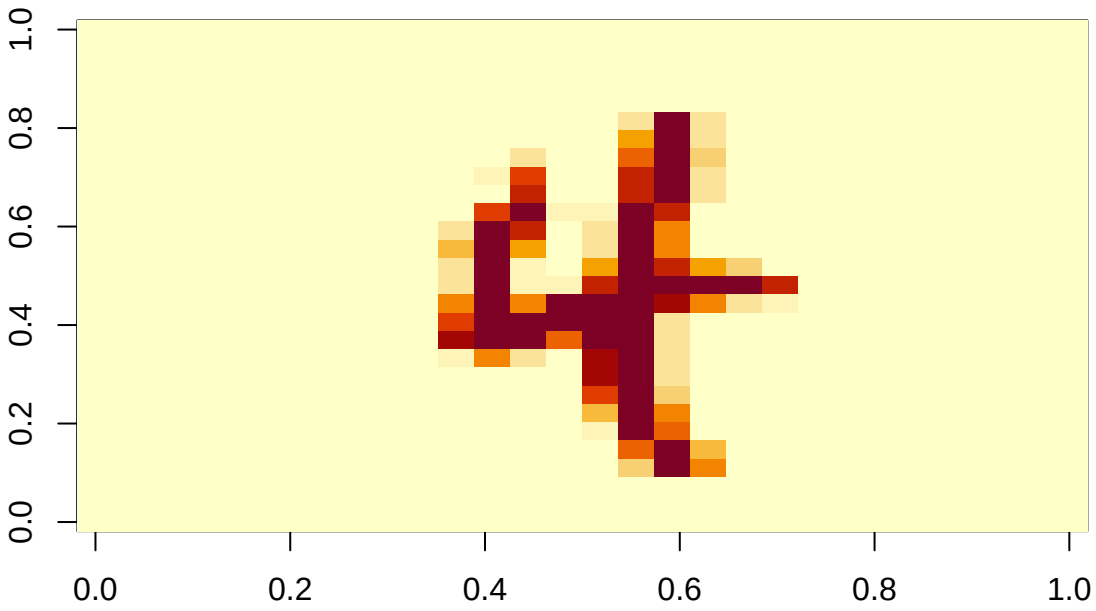
```
Misclassified_digits <- augmented_test_logit %>%
  filter(Number != .pred_class) %>%
  group_by(Number) %>%
  slice_min(Size)
```

```
plot_digit(mnist$test$images[Misclassified_digits$idx[1], ])
```



For this image of a one, it has many dark pixels in the top half, and it is also very wide. The width is due to the slant. The combination of these two elements lead the model to classify this image as a 4.

```
plot_digit(mnist$test$images[Misclassified_digits$idx[2], ])
```



This four contains very few dark pixels in the top half. Therefore, it would be classified as a 1 since 1s typically have less dark pixels in the top half. While it is somewhat wide, the width is not significant enough for it to be classified as a 4.

Changing things up (I)

We have already calculated our contrasted calculated features in the Dataset Creation section. Therefore, we only need to make the appropriate changes to our recipe.

```
# Set our recipe.
# Note: Size_Contrast and Dark_Contrast are just our contrast variables that are calculated on new rows
NumberRecipe_contrast <-
  recipe(Number ~ Size_Contrast + Dark_Contrast, data=train_tbl)

# set our model type.
logit_model_contrast <- logistic_reg() %>%
  set_engine("glm") %>% # decide engine
  set_mode("classification") # we are so wanting to do classification here

# work flow.
logit_wflow_contrast <- workflow() %>%
  add_recipe(NumberRecipe_contrast) %>% # mix recipe and model
  add_model(logit_model_contrast)
```

```
# fit the model.
logit_fit_contrast <- fit(logit_wflow_contrast, train_tbl)
```

We now need to test our model.

```
# table to evaluate our table
augmented_test_contrast <-augment(logit_fit_contrast, test_tbl)

#confusion matrix
augment(logit_fit_contrast, test_tbl)%>%
  conf_mat(Number, .pred_class)
```

```
##           Truth
## Prediction   1   4
##          1 106   5
##          4  14  75
```

```
# Accuracy
ctu1_acc<-augmented_test_contrast%>%
  accuracy(Number, .pred_class)

# Misclassification Rate
ctu1_mcr <- 1 - ctu1_acc$.estimate
ctu1_mcr
```

```
## [1] 0.095
```

```
# Comparing to the non-contrasted model.
mcr_df$logistic.regression
```

```
## [1] 0.105
```

```
# Distributional differences between contrasted width and non-contrasted width.
train_tbl %>%
  group_by(Number)%>%
  summarise(mean_width = mean(Size),
            mean_width_contrasted = mean(Size_Contrast),
            mean_width_diff = mean(Size) - mean(Size_Contrast))
```

```
## # A tibble: 2 x 4
##   Number mean_width mean_width_contrasted mean_width_diff
##   <fct>       <dbl>                 <dbl>           <dbl>
## 1 1            9.07                  7.89            1.18
## 2 4           16.1                  15.1             1.07
```

```
# Distributional differences between contrasted width and non-contrasted width.
train_tbl %>%
  group_by(Number)%>%
  summarise(mean_width = mean(Size),
            mean_width_contrasted = mean(Size_Contrast))
```

```
## # A tibble: 2 x 3
##   Number mean_width mean_width_contrasted
##   <fct>       <dbl>                 <dbl>
## 1 1            9.07                  7.89
## 2 4           16.1                  15.1
```

```
# Distributional differences between contrasted top half and non-contrasted top half.
train_tbl %>%
  group_by(Number)%>%
  summarise(mean_top_half = mean(Dark),
            mean_top_half_contrasted = mean(Dark_Contrast))
```

```
## # A tibble: 2 x 3
##   Number mean_top_half mean_top_half_contrasted
##   <fct>          <dbl>                    <dbl>
## 1 1               28.1                     28.1
## 2 4               40.7                     40.7
```

```
# Is there any difference between Dark pixels in the top half between contrasted
# digits and non-contrasted digits.
cont_dark_diff <- train_tbl %>%
  filter(Dark != Dark_Contrast)

row_number(cont_dark_diff)
```

```
## integer(0)
```

The contrasted image model provides a greater accuracy and lower misclassification rate than the non-contrasted model. This is most likely due to the differences in width. There is a greater difference between ones and fours in the contrasted width than in the non-contrasted width. Additionally there is no difference in our "Dark Pixels in the Top Half" calculation between the contrasted and non-contrasted images.

Changing things up (II)

Pulling in our sixes and feature calculation.

Below we are copying the previous method for sampling ones and fours, but this time it is sampling sixes. We do calculate the features on contrasted sixes as well, but this is not necessary for the model we will end up building.

```
image6_train <- get_image_digit_train(6)

train_count_6_bool <- mnist$train$labels==6 # get boolean of 1s in train
length6_train <- sum(train_count_6_bool) # get number of 1s for the loop
idx6_train <- vector(mode = "integer", length = length6_train)
n=1
for (i in 1:60000){

  if (train_count_6_bool[i] == TRUE){
    if (idx6_train[n] != 0){
```

```
      n = n+1
    }
    idx6_train[n] <- i

  }

}

image6_test <- get_image_digit_test(6)

test_count_6_bool <- mnist$test$labels==6 # get boolean of 1s in test
length6_test <- sum(test_count_6_bool) # get number of 1s for the loop
idx6_test <- vector(mode = "integer", length = length6_test)
n=1
for (i in 1:10000){

  if (test_count_6_bool[i] == TRUE){
    if (idx6_test[n] != 0){
      n = n+1
    }
    idx6_test[n] <- i

  }

}

#Calculate features for training get the number of pixels for sixes.
image_6_dark_train <- get_num_pixels(image6_train)# Dark pixels top half.
# Dark pixels top half on the contrasted digit.
image_6_dark_cont_train <- count_dark_pixels_contrast(image6_train)

#Calculate features for training get size of the image for sixes.
image_6_size_train <- get_size(image6_train)# Width for sixes.
# Width for contrasted sixes.
image_6_size_cont_train <- get_size_contrast(image6_train)

#Calculate features for testing get the number of pixels for sixes.
image_6_dark_test <- get_num_pixels(image6_test)# Dark pixels top half.
# Dark pixels top half on the contrasted digit.
image_6_dark_cont_test <- count_dark_pixels_contrast(image6_test)

#calculate features for testing get size of the image for sixes.
image_6_size_test <- get_size(image6_test)# Width for sixes.
# Width for contrasted sixes.
image_6_size_cont_test <- get_size_contrast(image6_test)

#Combine the tables to get our training and testing tables
combined_table_6_train <- data.frame(Size = image_6_size_train,
                                     Size_Contrast = image_6_size_cont_train,
                                     Dark = image_6_dark_train,
                                     Dark_Contrast = image_6_dark_cont_train,
                                     Number = 6,
                                     idx = idx6_train)
```

```
combined_table_6_test <- data.frame(Size = image_6_size_test,
                                    Size_Contrast = image_6_size_cont_test,
                                    Dark = image_6_dark_test,
                                    Dark_Contrast = image_6_dark_cont_test,
                                    Number = 6,
                                    idx =idx6_test)

# Sample testing and training.
set.seed(123)
sampled_data_train6 <- combined_table_6_train %>% sample_n(400, replace = FALSE)
sampled_data_test6 <- combined_table_6_test %>% sample_n(100, replace = FALSE)

# Add to datasets with 1s and 4s.
total_sample_with6_train <- bind_rows(sampled_data_train, sampled_data_train6)
total_sample_with6_test <- bind_rows(sampled_data_test, sampled_data_test6)
```

Build our model

Since we are adding a third possible outcome, we will need to use multinomial logistic regression instead of regular logistic regression. We will use our non-contrasted calculated features as our predictors in this model.

```
# Get Number (our output) as a factor.
total_sample_with6_train <- total_sample_with6_train %>%
  mutate(Number = as.factor(Number))%>%
mutate(Number=fct_relevel(Number, c("1","4","6")))

total_sample_with6_test <- total_sample_with6_test %>%
  mutate(Number = as.factor(Number))%>%
mutate(Number=fct_relevel(Number, c("1","4","6")))

# Model building.

# Set our recipe.

digit_recipe_multi <- recipe(Number ~ Size + Dark, data=total_sample_with6_train)

# Set our model type.

logit_model_multi <- multinom_reg(
  mode = "classification",
  engine = "nnet")

# Work flow.

digit_wflow_multi <- workflow() %>%
  add_recipe(digit_recipe_multi) %>%
  add_model(logit_model_multi)

# Fit the model on our training data set.

digit_fit_multi <- fit(digit_wflow_multi, total_sample_with6_train)
```

```
# Table to evaluate our model on our testing data set.

augment_test_multi <- augment(digit_fit_multi, total_sample_with6_test)
```

Confusion Matrix and Accuracy

```
# Accuracy of model with sixes.
acc_multi<-augment_test_multi%>%
  accuracy(Number, .pred_class)
acc_multi
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.633
```

```
# Accuracy of the logistic regression model without sixes.
acc_log
```

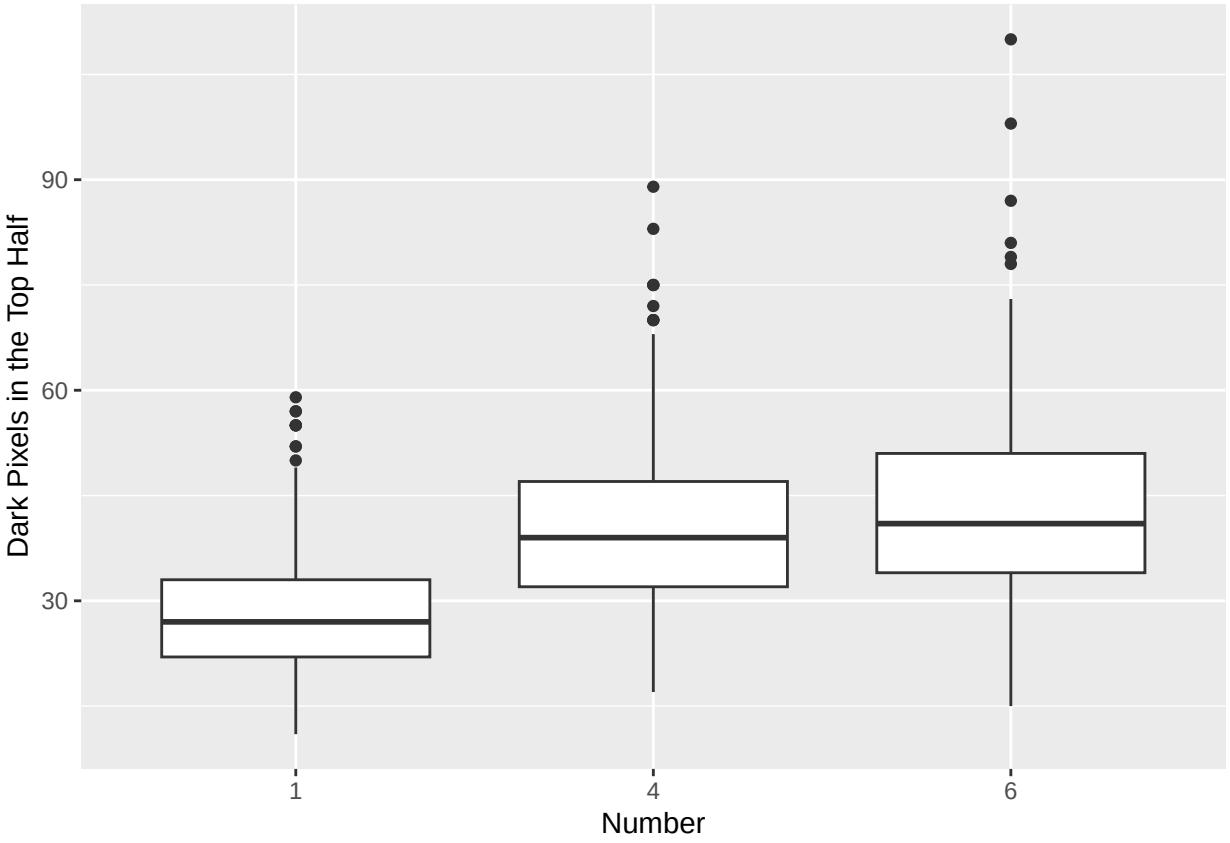
```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.895
```

```
# Confusion Matrix of our logistic regression model with sixes.
augment_test_multi %>%
  conf_mat(Number, .pred_class)
```

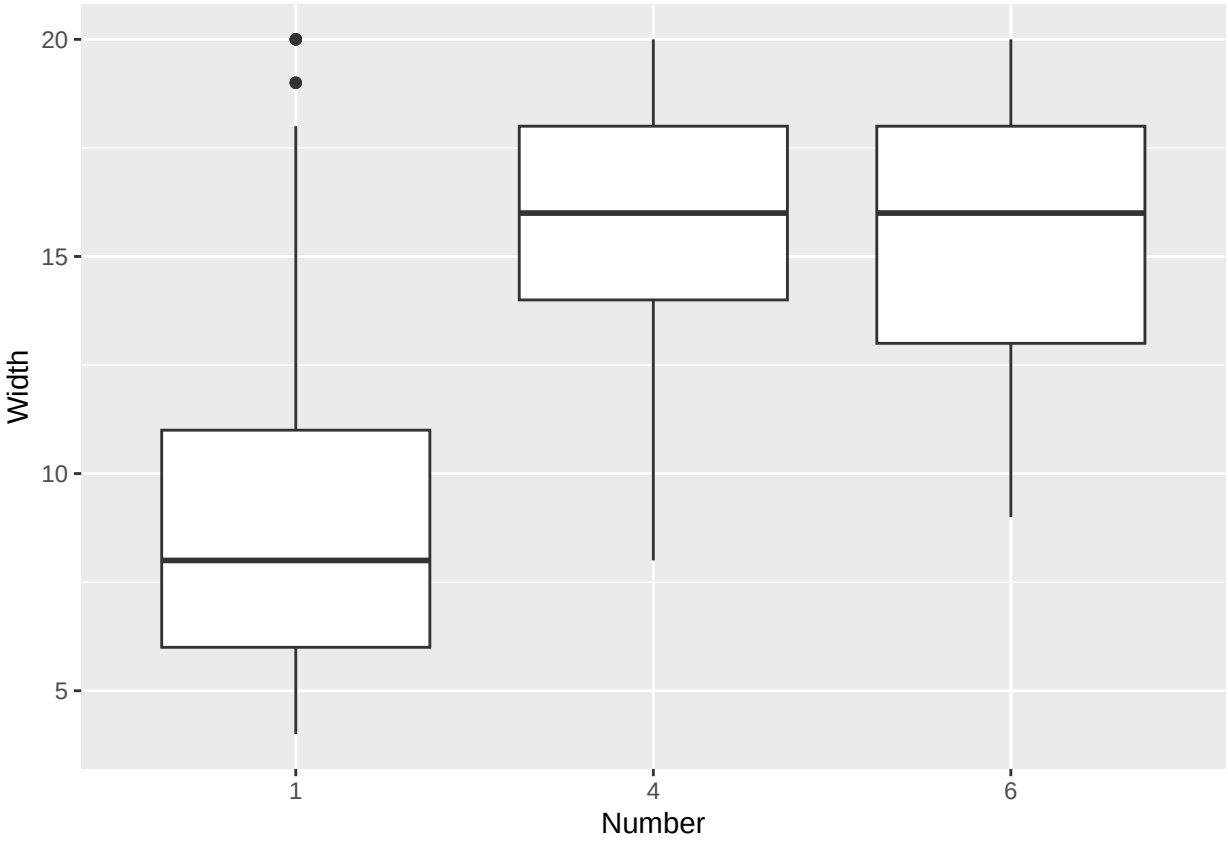
```
##           Truth
## Prediction   1   4   6
##          1 103   4  20
##          4  12  48  41
##          6   5  28  39
```

When adding sixes to our data set, our chosen model's accuracy declines. The accuracy declines from 89.5% to 63.3%. This can be attributed to the model not being able to tell the differences between fours and sixes. The confusion matrix shows this pattern. The boxplots below show why this is the case. Fours and sixes have similar distributional qualities in both of our calculated features.

```
#Distribution of calculated columns.
total_sample_with6_train %>%
  ggplot(mapping = aes(x = Number, group = Number, y = Dark))+
  geom_boxplot()+
  labs(y = "Dark Pixels in the Top Half")
```




```
total_sample_with6_train %>%
  ggplot(mapping = aes(x = Number, group = Number, y = Size))+
  geom_boxplot()+
  labs(y = "Width")
```



Decision Boundary

```
augment(digit_fit_multi, grid_tbl)%>% # generate percentages for the background,
  # grid_tbl was defined for the other decision boundary before.
ggplot(mapping = aes(x = Size, y = Dark, z = .pred_4 , fill = .pred_4)) +
  geom_raster() + # Assigns color to each square of the grid.
  scale_fill_viridis( option="inferno") + # Define color scale we want to use.
  stat_contour(mapping = aes(x = Size, y = Dark, z = .pred_4),
               breaks=c(0.5),
               color="royalblue1") + #Bayes/Decision boundary.
  stat_contour(mapping = aes(x = Size, y = Dark, z = .pred_1),
               breaks=c(0.5),
               color="royalblue1") +  #Bayes/Decision boundary.
  # The annotate function will provide text labels on the graph.
  annotate(geom="text",
           x=17,
           y=20,
           label="Pred 4",
           color="royalblue1") +
  annotate(geom="text",
           x=14,
           y=60,
           label="Pred 6",
           color="royalblue1")+
```

```
annotate(geom="text",
         x= 6,
         y=50,
         label="Pred 1",
         color="royalblue1")+
# Axis labels and title.
labs(x= "Width",
     y = "Dark Pixels in the Top Half",
     title = "Decision Boundary for 1s, 4s, and 6s")
```