Challenge 1

Luke Bleers, Caleb Griffiths, Henry Smith

3/16/2023

Initializing The Dataset

We are looking to develop a model that effectively classifies digitized images of handwritten numbers. We will start by looking at classifying 1s and 4s. Using two predictors, the number of inked pixels in the row with the highest number of inked pixels and the amount of pixels with medium ink values. Based on common drawings of 1s and 4s, we figure that 4s will have more inked pixels in the highest inked row and will have more medium ink, given the gradient in ink value around the edges of each pen stroke. After comparing classification models for accuracy, the most successful model will be retrained and tested on a new dataset containing 1s, 4s, and 6s.

Below is our method for building the dataset containing the 1s and 4s digits. We randomly sampled 1,000 digits and split them into a training and a testing dataset with an 8:2 allocation ratio, respectively.

```
idx <- mnist$train$labels %in% c(1,4)
sum(idx) #12584 1s and 4s.
```

```
## [1] 12584
```

```
set.seed(437) #Luke Bleers in Gematria

#dataset of digit pixel values (only 1,4).
digits.pixels.df <- mnist$train$images[idx,]
digits.labels.df <- mnist$train$labels[idx]

#sample 1000
indeces <- 1:12584
sample <- sample(indeces, 1000)
digits.pixels.sample <- digits.pixels.df[sample,]
digits.labels.sample <- digits.labels.df[sample]

#split sample
split = sort(sample(nrow(digits.pixels.sample), nrow(digits.pixels.sample)*.8))
digits.pixels.sample.train <- digits.pixels.sample[split,]
digits.pixels.sample.test <- digits.pixels.sample[-split,]
digits.labels.sample.train <- digits.labels.sample[split]
digits.labels.sample.test <- digits.labels.sample[-split]

#make list of digit matrices
digits.pixels.matrix.train <- list(matrix(digits.pixels.sample.train[1,],
                                          nrow=28)[,28:1])
digits.pixels.matrix.test <- list(matrix(digits.pixels.sample.test[1,],
                                         nrow=28)[,28:1])
```

```r
#add all digits to list
for(i in 1:nrow(digits.pixels.sample.train)){
  mat.train <- matrix(digits.pixels.sample.train[i,], nrow=28)[,28:1]
  digits.pixels.matrix.train[[i]] <- mat.train
}

for(i in 1:nrow(digits.pixels.sample.test)){
  mat.test <- matrix(digits.pixels.sample.test[i,], nrow=28)[,28:1]
  digits.pixels.matrix.test[[i]] <- mat.test
}
```

Defining Functions

With our dataset built, we developed the functions that will calculate our features. The two features we decided to implement are the highest inked row from an image and the count of the "medium" pixels in an image. Since we are trying to differentiate between a 1 and a 4, we looked at the visual differences in each number. We saw that 4's usually have a horizontal line in the middle of the number, and for most of the orientations of a 1, 1's do not have a horizontal line. This was our reason for creating the highest ink'd row function. Theoretically, since 4's have a horizontal line, they will have a larger highest ink'd row number than 1's will have. Our function rowMax first takes a dataset of matrix's, creates a vector called maxVals which is the length of the imputed dataset and initializes rowCount to be 0. Then, rowMax takes each matrix in the dataset, transforms it so the cells match the way they would show up when the matrix is plotted with the function image(), and runs through each row of the matrix by summing each row's cell values. If the sum of a row's cells are greater than rowCount, that sum becomes the new rowCount. This way when the highest rowCount is found, it will never be overtaken by another row's summed cells. Finally, each highest rowCount is added to maxVals corresponding to its respective matrix's location in the inputed dataset and maxVals is returned.

Highest Inked Row

```r
#input list of matrices
#initialize vector
#for each index: initiate counts, mutate matrix, generate stat, store in vector
#return vector

rowMax <- function(ds){
  maxVals <- vector("double", length(ds))
  for(i in 1:length(ds)){
    rowCount = 0
    digit <- ds[[i]]>0
    digit <- t(digit)
    for(x in 1:nrow(digit)){
      if(sum(digit[x,]) >= rowCount){
        rowCount = sum(digit[x,])
      }
    }
    maxVals[i] <- rowCount
  }
  return(maxVals)
}
```

Next the medium ink function is defined. We also noticed that since 4's take up more space and ink in general, they are more likely to have a longer perimeter than the 1's. For this reason, we thought it was logical to calculate the perimeter of a digit by counting the number of "medium ink'd" digits in an image. Before calculating the medium ink'd value, we would need to find the optimal boundaries for counting a cell as medium. Our function medInk takes a dataset, as well as lower and upper boundaries for what constitutes a cell as medium ink. First, our function initializes a vector medVals which is the length of the imputed dataset, then for each matrix in the dataset, the loop goes through each row in the matrix, and counts the number of medium ink'd cells based on the lower and upper boundary inputs. Then, the function sums the rows of counts of medium ink's cells to get a total number of medium ink's cells in the matrix. Finally, each matrix's total count of medium ink'd cells is added to medVals corresponding to its matrix location in the imputed dataset and medVals is returned.

Medium Ink (Borders)

```
#input list of matrices
#initialize vector
#for each index: calculate medium pixels, store in vector
#return vector

medInk <- function (ds, low, high){
  medVals <- vector("double", length(ds))
  for(i in 1:length(ds)){
    digit <- ds[[i]]
    med_pix <- digit %in% c(low:high)
    num_pixels <- sum(med_pix)
    medVals[i] <- num_pixels
  }
  return(medVals)
}
```

Since there is no agreed-upon spectrum on which the "medium-valued pixels" lay, we considered defining this range arbitrarily. However, we can compare the difference in mean amount of ink between the two digit classes to optimize this feature; we looked to compute the range that maximized this difference.

A Hiccup

```
#optimize high low

opt.grid <- expand_grid(low = seq(0,130, 10), high=seq(130, 250, 10))
param.tbl <- tibble(low = opt.grid$low, high = opt.grid$high,
                    mean1 = rep(0, nrow(opt.grid)), mean4 = rep(0, nrow(opt.grid)),
                    diff = mean4 - mean1)

for(i in 1:nrow(opt.grid)){
  tbl <- tibble(label = as.factor(digits.labels.sample.train),
                ink = medInk(digits.pixels.matrix.train,
                             opt.grid[i,1][[1]],
                             opt.grid[i,2][[1]]))

  m <- tbl %>%
    group_by(label) %>%
    summarize(mean = mean(ink))
```

```
  param.tbl[i,3] <- m[2][[1,1]]
  param.tbl[i,4] <- m[2][[2,1]]
}

param.tbl %>%
  mutate(diff = mean4 - mean1) %>%
  slice_max(diff)
```

```
## # A tibble: 1 x 5
##     low  high mean1 mean4  diff
##   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1    10   250  49.0  85.7  36.7
```

The output above tells us that an in-value range of 10-250 will maximize this difference in sample means. Considering the fact that the ink ranges from values of 0 to 255, this optimized metric would effectively be comparing the total amount of ink in each drawing, rather than the ink along the borders of each stroke. However, it is fair to say that there is likely a positive relationship between the number of lightly shaded pixels along the border of the drawing and the total amount of ink. Therefore, we will consider this predictor to be more of an indicator of total ink, rather than trying to capture the bordering ink.

With both features defined and optimized, we can begin building our classification models. We have to calculate the features for both our testing and training data sets. Then we create a recipe as we're going to use the standard tidyverse set up for organizing the models. We always predict the label value based on the features we've calculated, regardless of what models we use.

Building Models

```
digits.data.train <- tibble(index = seq(1:length(digits.labels.sample.train)),
                            label = as.factor(digits.labels.sample.train),
                            maxRow=rowMax(digits.pixels.matrix.train),
                            mediumInk=medInk(digits.pixels.matrix.train, 10, 250))

digits.data.test <- tibble(index = seq(1:length(digits.labels.sample.test)),
                           label = as.factor(digits.labels.sample.test),
                           maxRow=rowMax(digits.pixels.matrix.test),
                           mediumInk=medInk(digits.pixels.matrix.test, 10, 250))

digits.recipe <-  digits.data.train %>%
  recipe(label~maxRow + mediumInk)
```

Now that we've got the recipe set, we'll develop the model, workflow, and fit for each of our selected models. We've decided to explore the KNN, Logistic, and LDA models. We chose LDA and logistic regressions because they better handle categorical variables and are more suited by classification. Logistic is naturally bound between 0 and 1 which means that no transformation is required and the results can be more easily interpreted. LDA also helps to account for issues that might arrise within the Logistic model such as issues handling the 6s later on. KNN helps to establish a rudimentary baseline from which to compare our other models.

Logistic Model

```
logit.model <- logistic_reg() %>%
  set_engine("glm") %>%
  set_mode("classification")

logit.wflow <- workflow() %>%
  add_recipe(digits.recipe) %>%
  add_model(logit.model)

logit.fit <- fit(logit.wflow, digits.data.train)

logit.digits.data <- logit.fit %>%
  augment(digits.data.test)
```

Linear Discriminant Analysis (LDA) Model

```
lda.model <- discrim_linear() %>%
  set_mode("classification") %>%
  set_engine("MASS")

lda.wflow <- workflow() %>%
  add_recipe(digits.recipe) %>%
  add_model(lda.model)

lda.fit <- fit(lda.wflow, digits.data.train)

lda.digits.data <- lda.fit %>%
  augment(digits.data.test)
```

KNN Model

```
#Optimization
calc_error <- function(kNear, train_tbl, test_tbl) {
  knn_model <- knn3(label~maxRow+mediumInk, data=train_tbl, k=kNear)
  pred <- predict(knn_model, test_tbl, type="class")
  mean(pred!=test_tbl$label)
}

error_test <- vector(mode="double",length = 20)
for(i in 1:20){
  error_test[i] <- calc_error(i, digits.data.train, digits.data.test)
}
tibble(k=1:20,error = error_test) %>%
  slice_min(error, with_ties = 0)
```

```
## # A tibble: 1 x 2
##       k error
##   <int> <dbl>
## 1     3 0.005
```

```
#Model Building
knn_model <- knn3(label~maxRow+mediumInk, data=digits.data.train, k=3)
pred <- predict(knn_model, digits.data.test)
pred_class <- predict(knn_model, digits.data.test, type = "class")

knn.digits.data <- digits.data.test %>%
  mutate(.pred_class=pred_class,
         .pred_1 = pred[,1],
         .pred_4 = pred[,2])
```

Model Discussion

```
logit.digits.data %>% conf_mat(label, .pred_class)
```

```
##            Truth
## Prediction   1   4
##          1 103   3
##          4   1  93
```

```
lda.digits.data %>% conf_mat(label, .pred_class)
```

```
##            Truth
## Prediction   1   4
##          1 103   3
##          4   1  93
```

```
knn.digits.data %>% conf_mat(label, .pred_class)
```

```
##            Truth
## Prediction   1   4
##          1 103   0
##          4   1  96
```

```
metrics <- metric_set(accuracy, sens, spec)
metrics(logit.digits.data, truth = label, estimate = .pred_class)
```

```
## # A tibble: 3 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.98
## 2 sens     binary         0.990
## 3 spec     binary         0.969
```

```
metrics(lda.digits.data, truth = label, estimate = .pred_class)
```

```
## # A tibble: 3 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.98
## 2 sens     binary         0.990
## 3 spec     binary         0.969
```

```
metrics(knn.digits.data, truth = label, estimate = .pred_class)
```

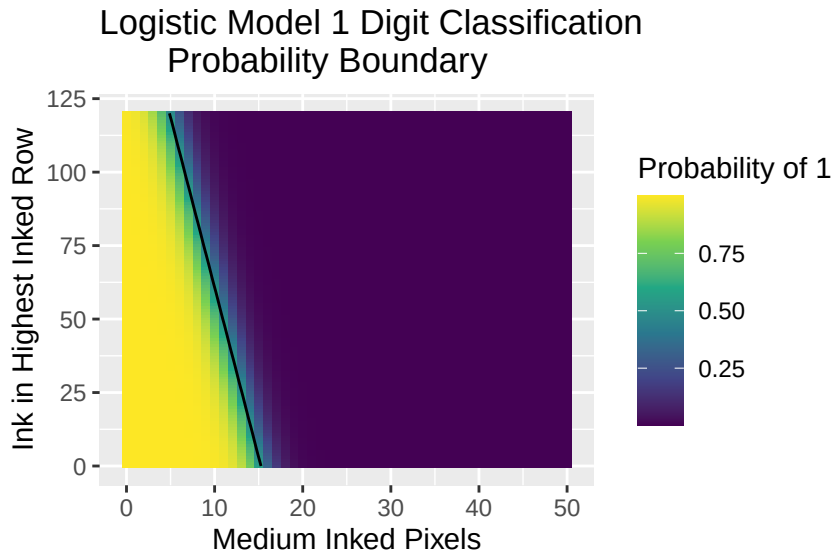
```
## # A tibble: 3 x 3
##   .metric  .estimator .estimate
```

```
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.995
## 2 sens     binary         0.990
## 3 spec     binary         1
```

Most of the time we could compare 2 models and pretty quickly determine if one was a better tool for our question, although in this situation we can see that both of our models are performing exactly the same. With identical confusion matrices, accuracy, sensitivity, and specificity, it's fair to say that LDA and Logistic perform just as well as one another. We can see that they both have an accuracy of 98% because only 4 numbers were incorrectly classified. Looking at the confusion matrix, ¾ of these were 4's that our model classified as 1's. We can see that KNN is performing 1.5% better with only 1 number misclassified.

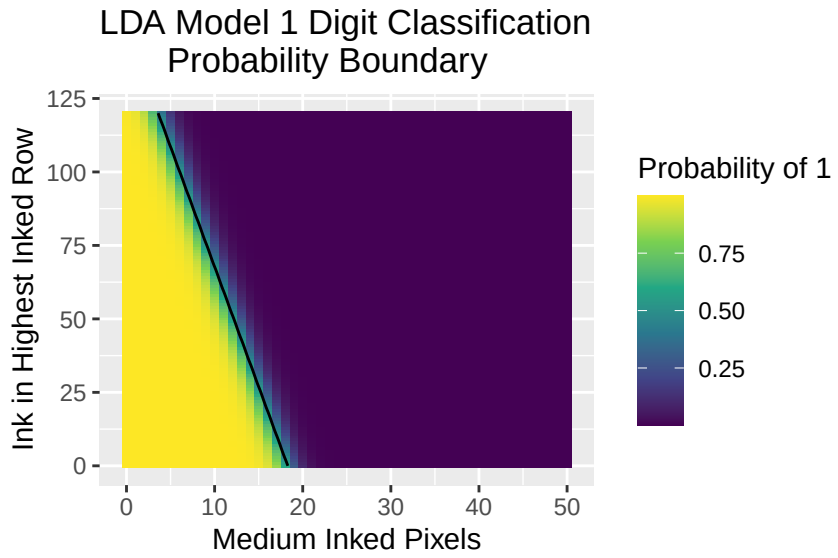
Visualization

```
grid_vec <- seq(0,120,1)
grid_2 <- seq(0,50,1)
grid_tbl <- expand_grid(maxRow=grid_2, mediumInk=grid_vec)

#Logit
augment(logit.fit, grid_tbl) %>%
ggplot(aes(maxRow, mediumInk, z=.pred_1, fill = .pred_1)) +
  geom_raster() +
  scale_fill_viridis_c() +
  stat_contour(breaks = c(0.5), color = "black")  +
  labs(title = "Logistic Model 1 Digit Classification
       Probability Boundary",
       fill = "Probability of 1") +
  xlab("Medium Inked Pixels") + ylab("Ink in Highest Inked Row")
```

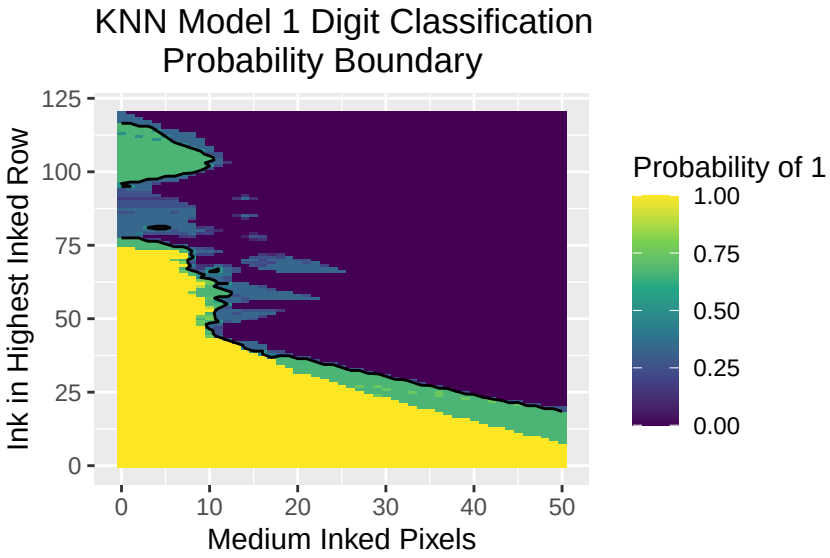



```
#LDA
augment(lda.fit, grid_tbl) %>%
ggplot(aes(maxRow, mediumInk, z=.pred_1, fill = .pred_1)) +
  geom_raster() +
  scale_fill_viridis_c() +
  stat_contour(breaks = c(0.5), color = "black")  +
```

```
  labs(title = "LDA Model 1 Digit Classification
       Probability Boundary",
       fill = "Probability of 1") +
  xlab("Medium Inked Pixels") + ylab("Ink in Highest Inked Row")
```




```
#KNN
pred_grid <- predict(knn_model, grid_tbl)
pred_class_grid <- predict(knn_model, grid_tbl, type = "class")

grid_fit <- grid_tbl %>%
  mutate(.pred_class=pred_class_grid,
         .pred_1 = pred_grid[,1],
         .pred_4 = pred_grid[,2])

grid_fit %>%
ggplot(aes(maxRow, mediumInk, z=.pred_1, fill= .pred_1)) +
  geom_raster() +
  scale_fill_viridis_c() +
  stat_contour(breaks=c(0.5), color="black")  +
  labs(title = "KNN Model 1 Digit Classification
       Probability Boundary",
       fill = "Probability of 1") +
  xlab("Medium Inked Pixels") + ylab("Ink in Highest Inked Row")
```



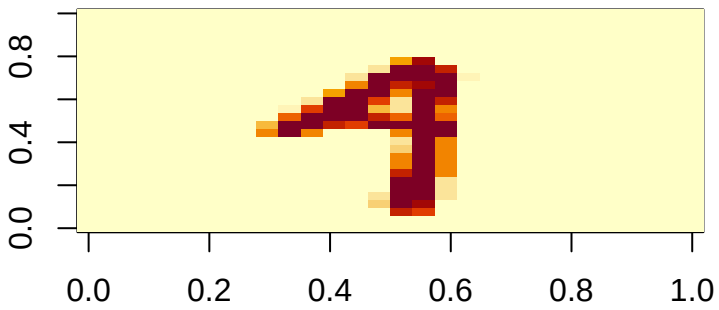
The plotted probability on this graph (lda) is the probability that a given image is a 1. This way, lower probabilities mean that there is a greater probability of that image being a 4. Our decision boundary intersects the maxRow axis at about 19, and intersects the mediumInk axis at about 150. Any maxRow and mediumInk values that are both below this line are most likely to be 1's. Any maxRow and mediumInk values that are both above this line are most likely to be 4's. MaxRow and mediumInk values that are close to or on the line may be misclassified.

Misclassified Digits

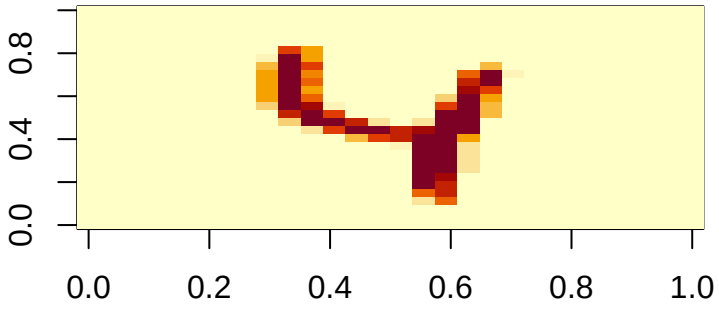
```
lda.digits.data %>%
  filter(label != .pred_class)
```

```
## # A tibble: 4 x 7
##   index label maxRow mediumInk .pred_class .pred_1 .pred_4
##   <int> <fct>  <dbl>     <dbl> <fct>         <dbl>   <dbl>
## 1    74 4          9        72 1             0.647   0.353
## 2    79 4          9        66 1             0.824   0.176
## 3   134 4          9        70 1             0.715   0.285
## 4   193 1         10        68 4             0.490   0.510
```

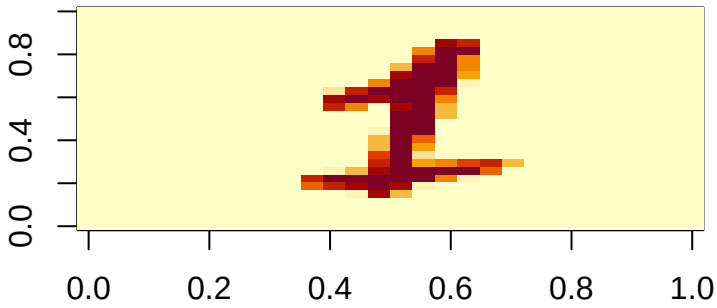
```
image(digits.pixels.matrix.test[[79]]) #4 classed as 1
```




```
image(digits.pixels.matrix.test[[134]]) #4 classed as 1
```




```
image(digits.pixels.matrix.test[[193]]) #1 classed as 4
```



As we can see when we pull the misclassified digits from the lda dataset, our misclassified digits are the ones that are close to our decision boundary line. Additionally, it should be noted that most of our misclassified digits are actually 4's and we predicted them to be 1's. Looking at the specific digits that got messed up, we might be able to make some guesses as to why. We look at our 1 and we are able to see that it has an unusual horizontal line at the bottom of the 1 which wasn't found among the bulk of the dataset and we use the max row feature to identify 4's so it makes sense why it was confused. The 4's that got confused as 1's generally have a softer horizontal line with it going at an angle in such a way that one single row doesn't get too populated with ink for our rowMax feature. Overall we might be able to adjust this to take in a range of rows and then calculate the rowMax although it's hard to say if this would reduce the misclassification or just create different issues.

This is where we've added 6's with a similar process from the beginning. We create testing and training groups and apply the feature calculations to both.

Adding Sixes

```
idxs <- mnist$train$labels %in% c(1,4,6)
sum(idxs)
```

```
## [1] 18502
```

```
set.seed(437)

digits.pixels.df <- mnist$train$images[idxs,]
digits.labels.df <- mnist$train$labels[idxs]

indeces <- 1:18502
sample <- sample(indeces, 1000)
digits.pixels.sample <- digits.pixels.df[sample,]
digits.labels.sample <- digits.labels.df[sample]

split = sort(sample(nrow(digits.pixels.sample), nrow(digits.pixels.sample)*.8))
digits.pixels.sample.train <- digits.pixels.sample[split,]
digits.pixels.sample.test <- digits.pixels.sample[-split,]
```

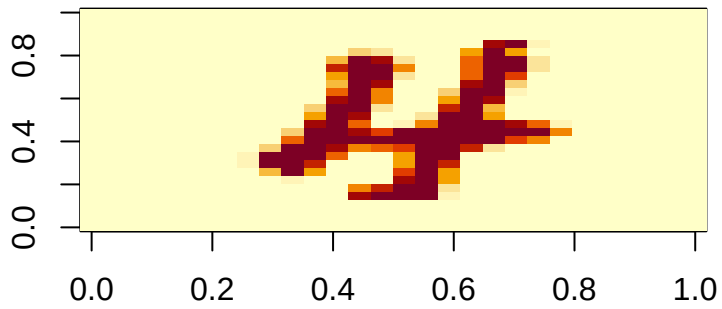
```
digits.labels.sample.train <- digits.labels.sample[split]
digits.labels.sample.test <- digits.labels.sample[-split]

digits.pixels.matrix.train <- list(matrix(digits.pixels.sample.train[1,],
                                          nrow=28)[,28:1])
digits.pixels.matrix.test <- list(matrix(digits.pixels.sample.test[1,],
                                         nrow=28)[,28:1])

for(i in 1:nrow(digits.pixels.sample.train)){
  mat.train <- matrix(digits.pixels.sample.train[i,], nrow=28)[,28:1]
  digits.pixels.matrix.train[[i]] <- mat.train
}

for(i in 1:nrow(digits.pixels.sample.test)){
  mat.test <- matrix(digits.pixels.sample.test[i,], nrow=28)[,28:1]
  digits.pixels.matrix.test[[i]] <- mat.test
}

digits.data.train.six <- tibble(index = seq(1:length(digits.labels.sample.train)),
                                label = as.factor(digits.labels.sample.train),
                                maxRow=rowMax(digits.pixels.matrix.train),
                                mediumInk=medInk(digits.pixels.matrix.train, 10, 250))

digits.data.test.six <- tibble(index = seq(1:length(digits.labels.sample.test)),
                               label = as.factor(digits.labels.sample.test),
                               maxRow=rowMax(digits.pixels.matrix.test),
                               mediumInk=medInk(digits.pixels.matrix.test, 10, 250))
```

```
lda_recipe <- recipe(label ~ mediumInk+maxRow, data=digits.data.train.six)

lda_wflow <- workflow() %>%
  add_recipe(lda_recipe) %>%
  add_model(lda.model)

digit_lda_fit <- fit(lda_wflow, digits.data.train.six)

digits.test.six.pred <- digit_lda_fit %>%
  augment(digits.data.test.six)

digits.test.six.pred %>%
  filter(label != .pred_class)
```

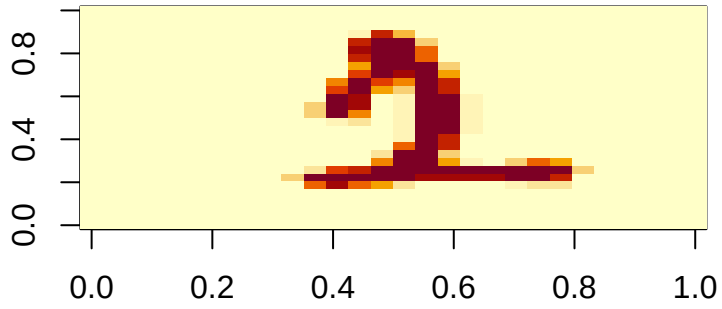
```
## # A tibble: 55 x 8
##    index label maxRow mediumInk .pred_class   .pred_1 .pred_4 .pred_6
##    <int> <fct>  <dbl>     <dbl> <fct>           <dbl>   <dbl>   <dbl>
##  1     5 4         13        92 6           0.00131    0.353   0.646
##  2     6 6         12        78 4           0.0179     0.501   0.482
##  3     9 4          8        58 1           0.905      0.0522  0.0423
##  4    14 4         15       111 6           0.0000176  0.216   0.784
##  5    16 4          9        70 1           0.473      0.224   0.303
##  6    21 1         14        80 4           0.00191    0.618   0.380
##  7    23 4         11        78 6           0.0469     0.417   0.536
##  8    25 6         13        64 4           0.0265     0.759   0.214
##  9    32 4         14        89 6           0.000707   0.471   0.528
```

```
## 10    33 6         16        90 4           0.0000839  0.599   0.401
## # ... with 45 more rows
## # i Use 'print(n = ...)' to see more rows
```

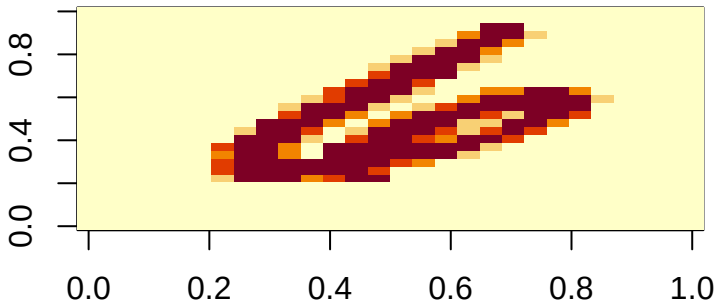
```
image(digits.pixels.matrix.test[[5]]) #4 classed as 6
```




```
image(digits.pixels.matrix.test[[21]]) #1 classed as 4
```




```
image(digits.pixels.matrix.test[[25]]) #6 classed as 4
```



We can notice these values which are predicted as another number than they actually are based on our features. We look at these errors and notice that they look very strange. Barely recognizable as the numbers they are labeled as, I wouldn't expect a human to read these without tilting their head. With strange extra segments in conjunction with features not optimized for differentiating 6's, I see where our model would begin to struggle.

Sixes Discussion

```
digits.test.six.pred %>%
  conf_mat(label, .pred_class)
```

```
##           Truth
## Prediction  1  4  6
##          1 81  3  5
##          4  2 32 18
##          6  1 26 32
```

```
metrics(digits.test.six.pred, truth = label, estimate = .pred_class)
```

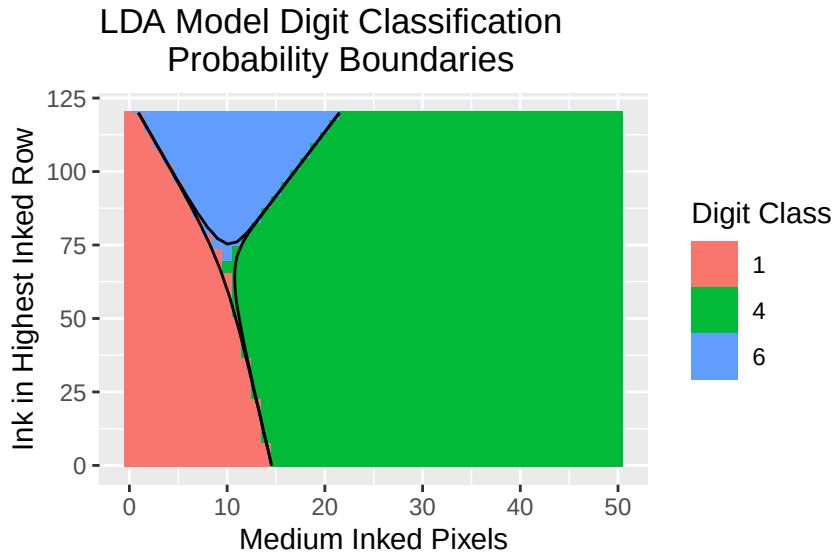
```
## # A tibble: 3 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.725
## 2 sens     macro          0.690
## 3 spec     macro          0.867
```

Our misclassification rate increases here because there are more opportunities to be wrong. Our features are focussed on splitting 1's from 4's. If we created a model with different/more features we might be able to better pull apart 1's, 4's, and 6's but as our model exists right now it's not super accurate. Our accuracy drops to 73% which is a 25% decrease but is still significantly better than a strictly random model which would hover right around 33%. Additionally, the digit that gets misclassified the most is the 4. 4's were misclassified for 6's 26 times and similarly, 6's were misclassified as 4's 18 times. We believe this is because our features are similar for 4's and 6's which would not create enough distinction between the two digits, and ultimately make for lower accuracy.

```
augment(digit_lda_fit, grid_tbl) %>%
ggplot(aes(maxRow, mediumInk)) +
  geom_raster(aes(z=.pred_1, fill= .pred_class)) +
  stat_contour(aes(z=.pred_1, fill= .pred_1), breaks=c(0.5), color="black") +
  stat_contour(aes(z=.pred_4, fill= .pred_4), breaks=c(0.5), color="black") +
  stat_contour(aes(z=.pred_6, fill= .pred_6), breaks=c(0.5), color="black")  +
  labs(title = "LDA Model Digit Classification
       Probability Boundaries",
       fill = "Digit Class") +
  xlab("Medium Inked Pixels") + ylab("Ink in Highest Inked Row")
```

```
## Warning: Ignoring unknown aesthetics: z
```

```
## Warning: Ignoring unknown aesthetics: fill
## Ignoring unknown aesthetics: fill
## Ignoring unknown aesthetics: fill
```



Considering KNN

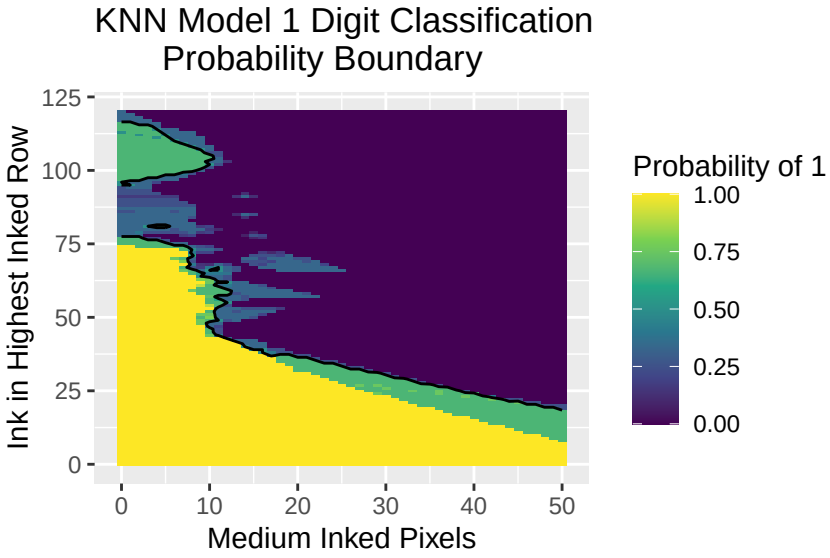
Here we have the KNN Model, although some of the visualizations are difficult to interpret. It remains the most accurate model but we can't fully explain how it gets to this point. We've completed some visualizations which might interest you.

Model Metrics & Boundary

```
metrics(knn.digits.data, truth = label, estimate = .pred_class)
```

```
## # A tibble: 3 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.995
## 2 sens     binary         0.990
## 3 spec     binary         1
```

```
pred_grid <- predict(knn_model, grid_tbl)
pred_class_grid <- predict(knn_model, grid_tbl, type = "class")

grid_fit <- grid_tbl %>%
  mutate(.pred_class=pred_class_grid,
         .pred_1 = pred_grid[,1],
         .pred_4 = pred_grid[,2])

grid_fit %>%
ggplot(aes(maxRow, mediumInk, z=.pred_1, fill= .pred_1)) +
  geom_raster() +
  scale_fill_viridis_c() +
  stat_contour(breaks=c(0.5), color="black")  +
  labs(title = "KNN Model 1 Digit Classification
       Probability Boundary",
       fill = "Probability of 1") +
  xlab("Medium Inked Pixels") + ylab("Ink in Highest Inked Row")
```



Sixes Application

```
knn_model <- knn3(label~maxRow+mediumInk, data=digits.data.train.six, k=3)
pred <- predict(knn_model, digits.data.test.six)
pred_class <- predict(knn_model, digits.data.test.six, type = "class")

knn.digits.data.six <- digits.data.test.six %>%
  mutate(.pred_class=pred_class,
         .pred_1 = pred[,1],
         .pred_4 = pred[,2],
         .pred_6 = pred[,3])

knn.digits.data.six %>% conf_mat(label, estimate = .pred_class)
```

```
##           Truth
```

```
## Prediction  1  4  6
##          1 80  1  1
##          4  1 38 22
##          6  3 22 32
```

```
knn.digits.data.six %>% metrics(label, estimate=.pred_class)
```

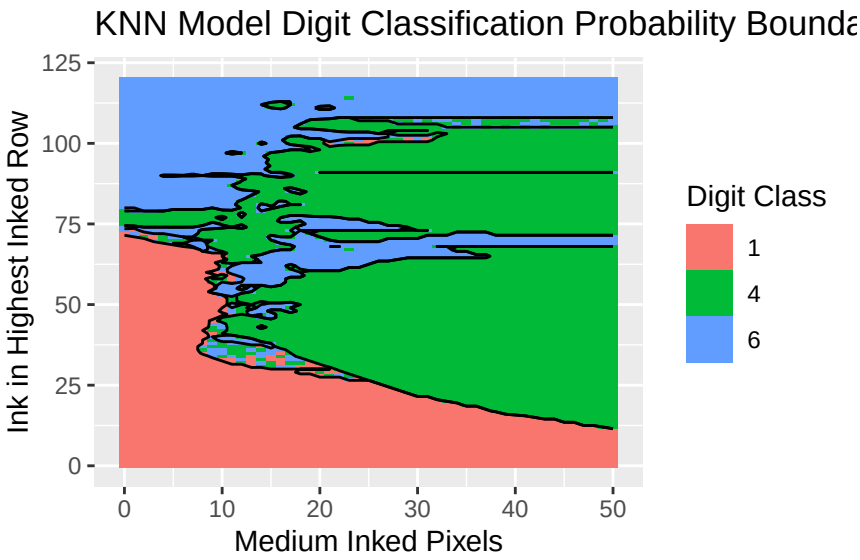
```
## # A tibble: 3 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.75
## 2 sens     macro          0.719
## 3 spec     macro          0.882
```

Probability Boundaries Plot

```
pred_grid <- predict(knn_model, grid_tbl)
pred_class_grid <- predict(knn_model, grid_tbl, type = "class")

grid_fit <- grid_tbl %>%
  mutate(.pred_class=pred_class_grid,
         .pred_1 = pred_grid[,1],
         .pred_4 = pred_grid[,2],
         .pred_6 = pred_grid[,3])

grid_fit %>%
ggplot(aes(maxRow, mediumInk)) +
  geom_raster(aes(z=.pred_1, fill= .pred_class)) +
  stat_contour(aes(z=.pred_1, fill= .pred_1), breaks=c(0.5), color="black") +
  stat_contour(aes(z=.pred_4, fill= .pred_4), breaks=c(0.5), color="black") +
  stat_contour(aes(z=.pred_6, fill= .pred_6), breaks=c(0.5), color="black") +
  labs(title = "KNN Model Digit Classification Probability Boundaries",
       fill = "Digit Class") +
  xlab("Medium Inked Pixels") + ylab("Ink in Highest Inked Row")
```

```
## Warning: Ignoring unknown aesthetics: z
```

```
## Warning: Ignoring unknown aesthetics: fill
## Ignoring unknown aesthetics: fill
## Ignoring unknown aesthetics: fill
```


KNN Model Digit Classification Probability Bounda
Ink in Highest Inked Row
125
100
75
50
25
0
0
10
20
30
40
50
Medium Inked Pixels
Digit Class
1
4
6