Introduction to Regression Trees

Andrew Noecker

4/20/2021

Introduction

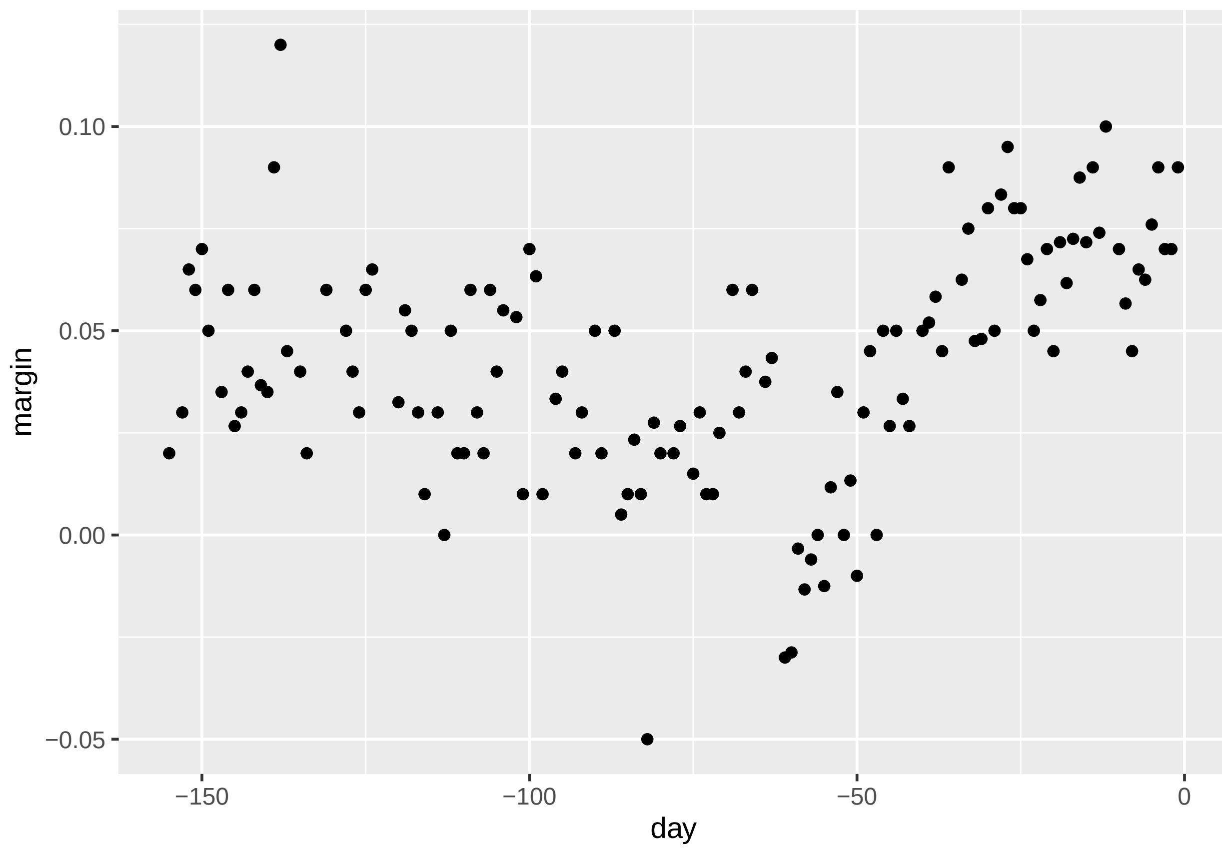
Today we will be using data from the presidential polls for the 2008 election (Obama vs McCain). Let's start by loading the dataset

```
library(dslabs)
data("polls_2008")
polls.2008.tbl <- tibble(polls_2008)
polls.2008.tbl
```

```
## # A tibble: 131 x 2
##      day margin
##    <dbl>  <dbl>
##  1  -155 0.0200
##  2  -153 0.0300
##  3  -152 0.065
##  4  -151 0.06
##  5  -150 0.07
##  6  -149 0.05
##  7  -147 0.035
##  8  -146 0.06
##  9  -145 0.0267
## 10  -144 0.0300
## # ... with 121 more rows
```

Notice that we only have two variables, the first one is `day` which measures the day until election day (day 0 is election night) and `margin` which is the average difference margin between Obama and McCain for that day. We can plot our data by doing

```
ggplot(polls.2008.tbl, aes(day, margin))+
  geom_point()
```



Using regression trees

We are interested in finding the **trend** of the margin using the day as our input variable. In particular we will be assuming that the trend for a period of days will be constant, so using a regression tree seems like the natural choice. So without further do, let's implement our usual steps using `tidymodels()`

- We define our testing/training dataset:

```
set.seed(123)
poll.split <- initial_split(polls.2008.tbl)
poll.train.tbl <- training(poll.split)
poll.test.tbl <- testing(poll.split)
```

- We define our regression tree model. Initially we will settle for `tree_depth` parameter of 2 and since the margin is a continous variable we will be using the `"regression"` mode.

```
poll.model <-
  decision_tree(tree_depth=2) %>%
  set_mode("regression") %>%
  set_engine("rpart")

poll.recipe <- recipe(margin ~ day, data=poll.train.tbl)

poll.wflow <- workflow() %>%
    add_recipe(poll.recipe) %>%
    add_model(poll.model)
```

- We train our model using our training data

```
poll.fit  <- fit(poll.wflow, poll.train.tbl)
```

- And we evaluate our model performance using our testing data

```
poll.final.tbl <- augment(poll.fit, poll.test.tbl)
rmse(poll.final.tbl, margin, .pred)
```

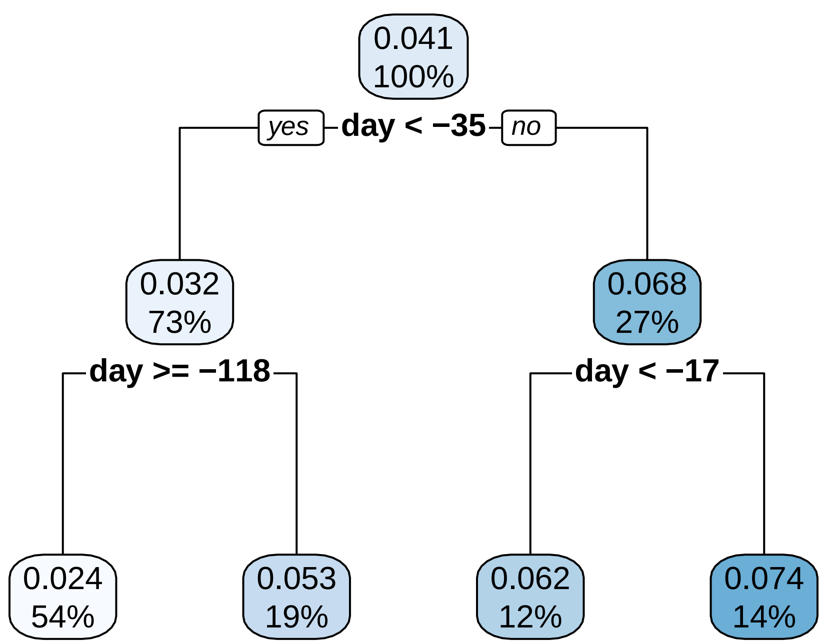
```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard      0.0274
```

```
rsq(poll.final.tbl, margin, .pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rsq     standard       0.220
```

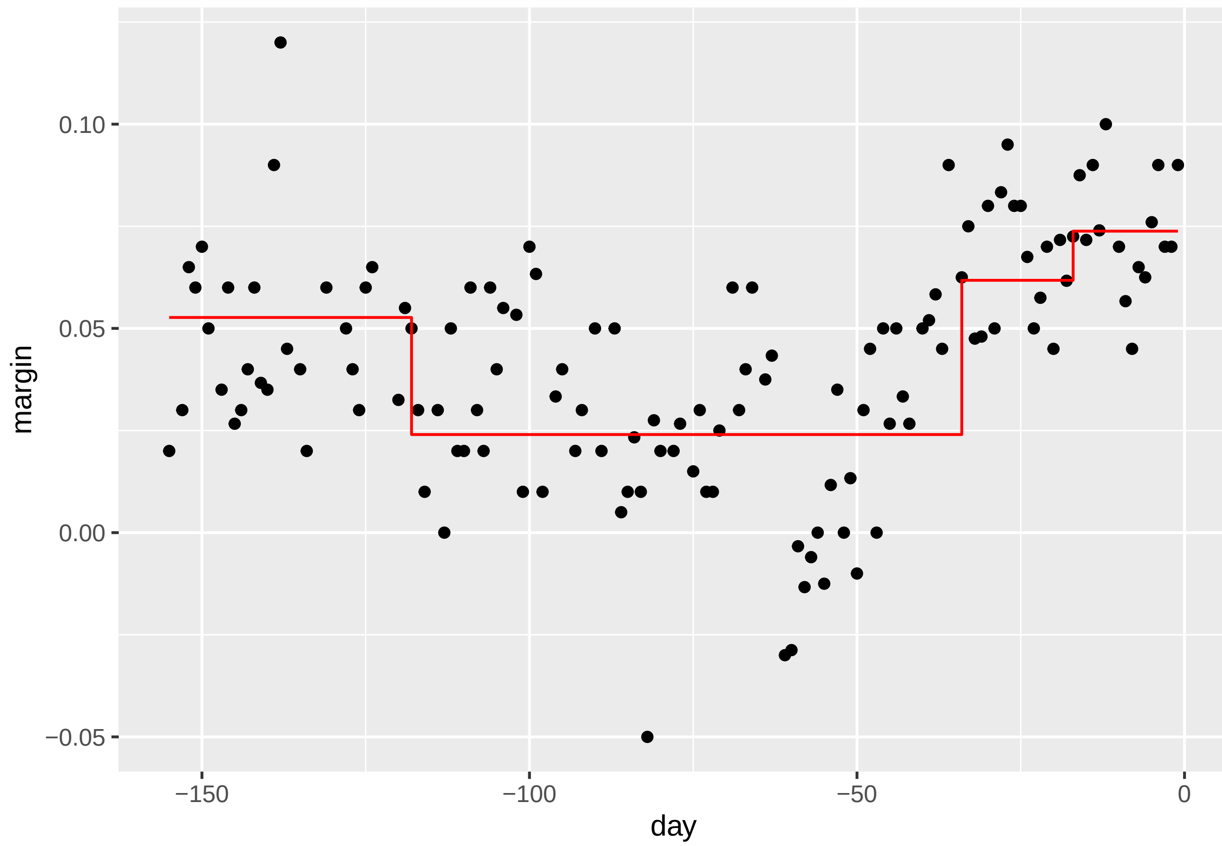
We can visualize our regression tree as a tree

```
poll.fit %>%
  extract_fit_engine() %>%
  rpart.plot()
```



Or better yet we can see the trend obtained by the regression tree on our original dataset

```
augment(poll.fit, polls.2008.tbl) %>%
  ggplot()+
  geom_point(aes(day,margin))+
  geom_step(aes(day,.pred), col="red")
```



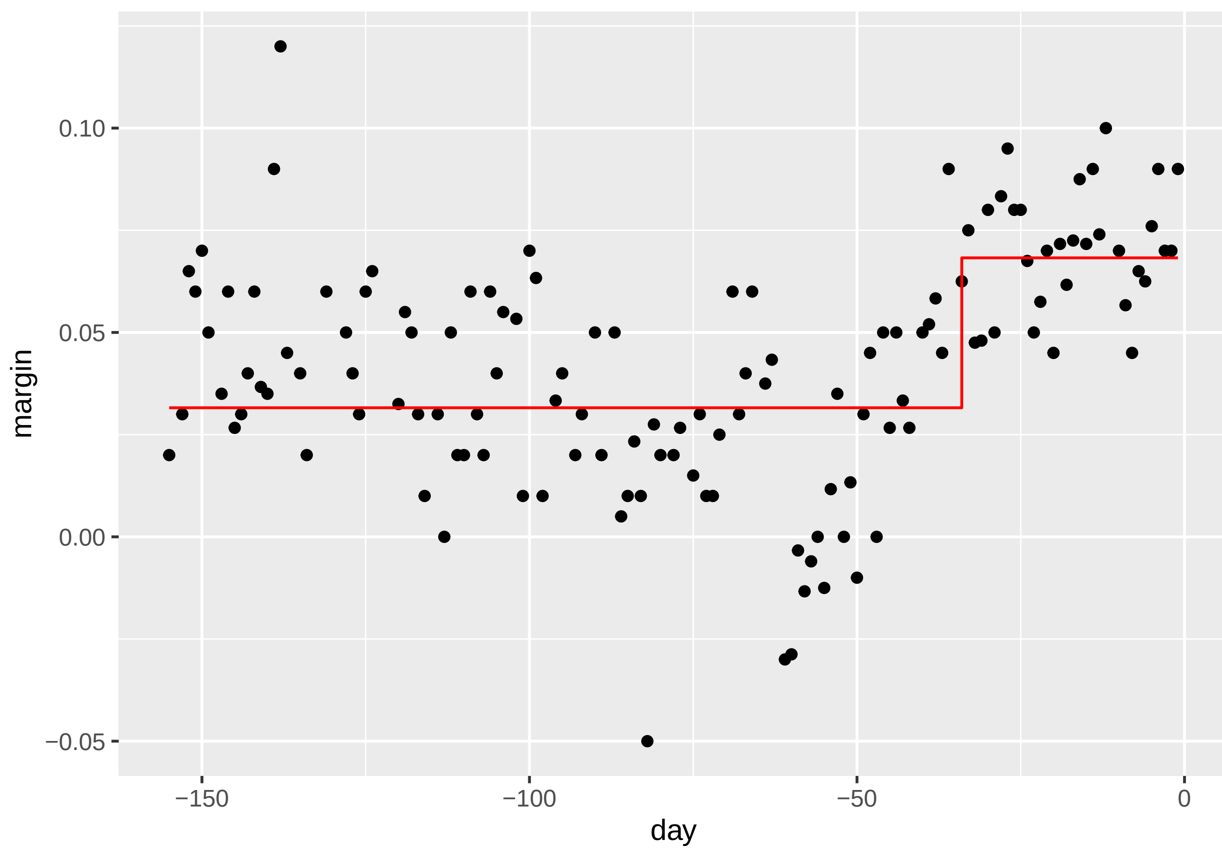
Understanding the parameters of regression trees

In the following exercises we will be exploring the process of the construction of the regression tree and how to optimize the selection of the parameters for our tree model.

1. Fill out the blanks of the function `calc_mse_tree` that receives two parameters, `tree_depth` and `cost_complexity`, creates a regression tree with such parameters and calculates the *mse* on the training data (yes that's correct, the *training* dataset). Test your function using `tree_depth`=1,2, while keeping `cost_complexity`=0.1

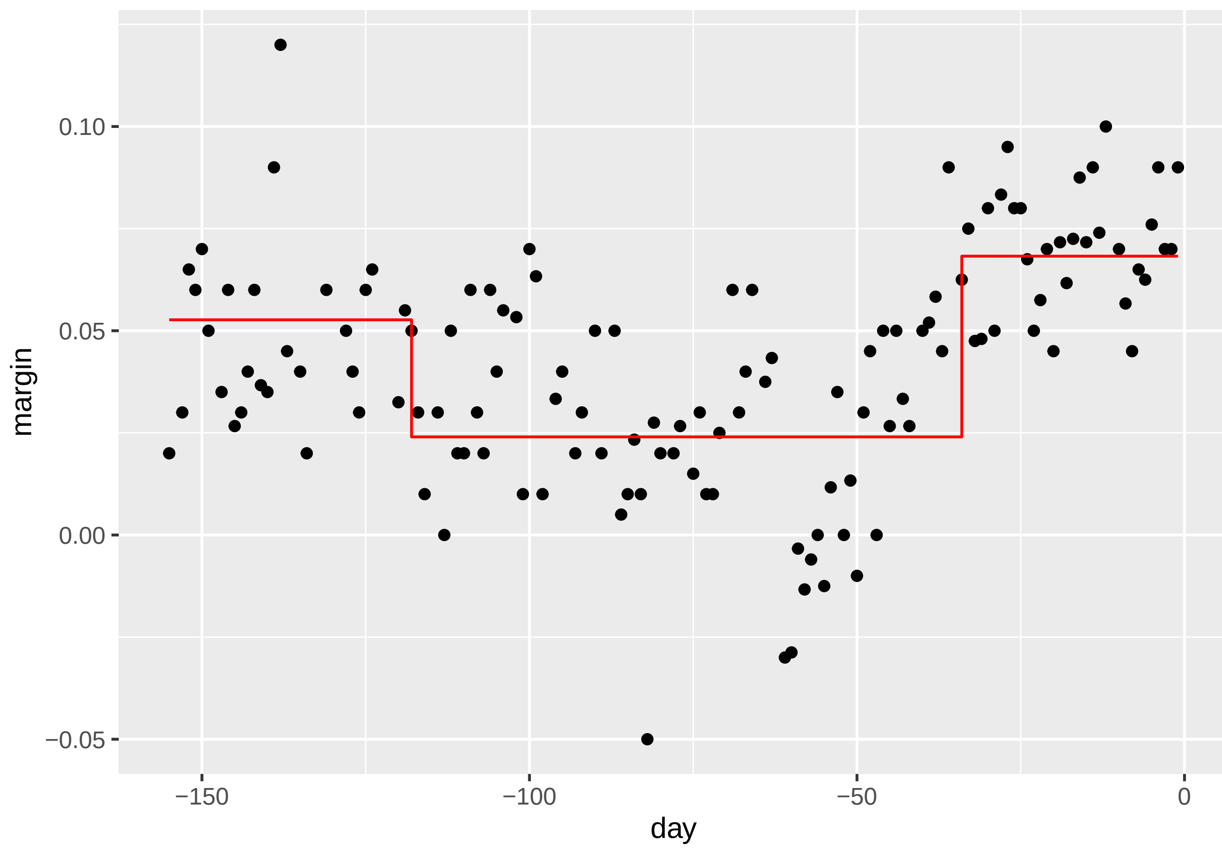
```
calc_mse <- function(tree_depth, cost_complexity) {
   # Train your model
    poll.model <-
      decision_tree(tree_depth=tree_depth, cost_complexity = cost_complexity) %>%
      set_mode("regression") %>%
      set_engine("rpart")

    poll.recipe <- recipe(margin ~ day, data=poll.train.tbl)

    poll.wflow <- workflow() %>%
```

```
      add_recipe(poll.recipe) %>%
      add_model(poll.model)

    poll.fit  <- fit(poll.wflow, poll.train.tbl)

   # Visualize your model
    print(augment(poll.fit, polls.2008.tbl) %>%
    ggplot()+
    geom_point(aes(day,margin))+
    geom_step(aes(day,.pred), col="red"))

  # Calculate and output the rmse
    poll.final.tbl <- augment(poll.fit, poll.train.tbl)
    print((rmse(poll.final.tbl, margin, .pred)$.estimate)^2)
    #print(rsq(poll.final.tbl, margin, .pred))
}

calc_mse(1,0.1)
```




```
## [1] 0.0005434706
```

```
calc_mse(2, 0.1)
```

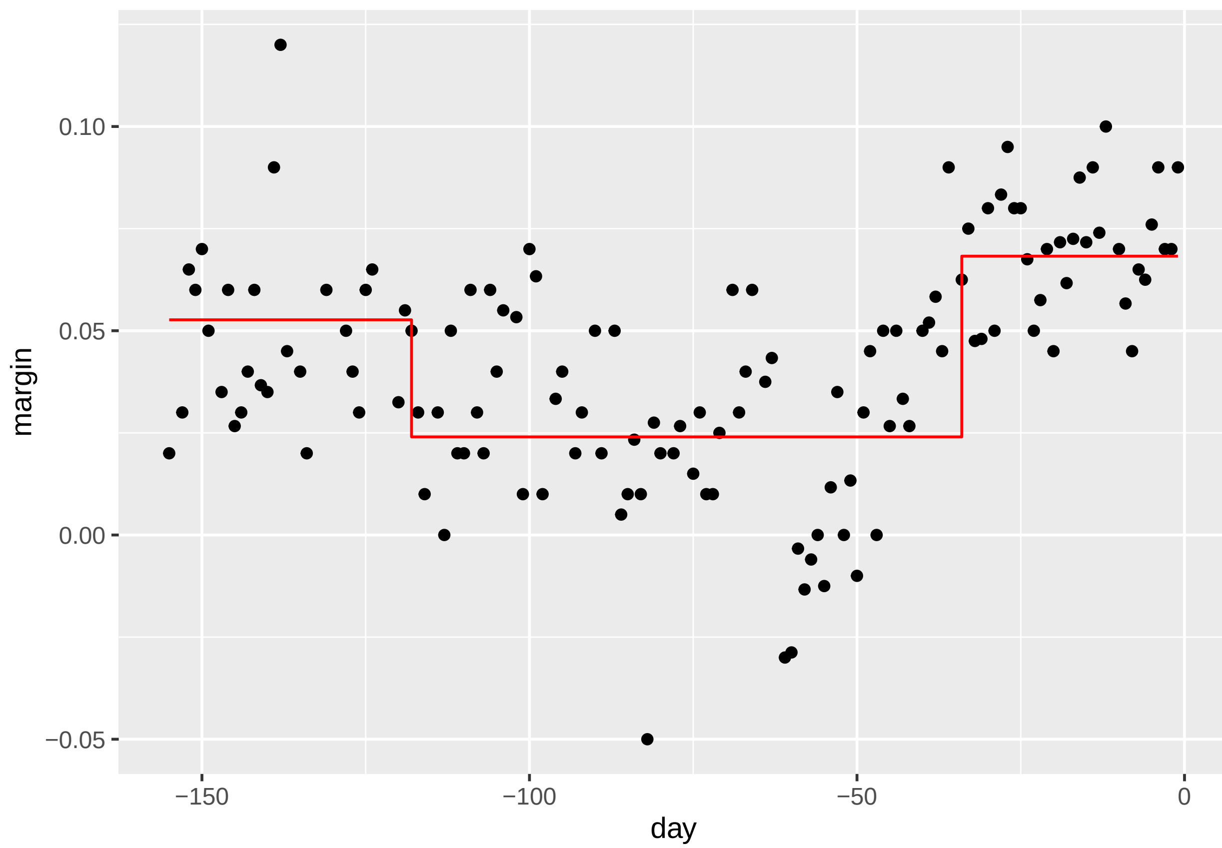



```
## [1] 0.0004262214
```

2. In principle, every time that we add a level to our tree we can decrease our RSS. If we continue this approach indefinitely we could end up with a tree where every leaf is a single point which is a clear case of overfitting. The `complexity_parameter` (`cp`) controls the number of recursive splits your model takes. Roughly, it does this by measuring the difference in fit (measured by the MSE) by adding a new level and stopping if this value is less than the `cp` value. Armed with this knowledge explain why `calc_mse(3,0.1)` produces the same results as `calc_mse(2,0.1)`. Experiment changing the `cp` parameter so that you get a regression tree with three levels when you set the `tree_depth`=3. Change your parameters so that you get a regression tree with six levels.

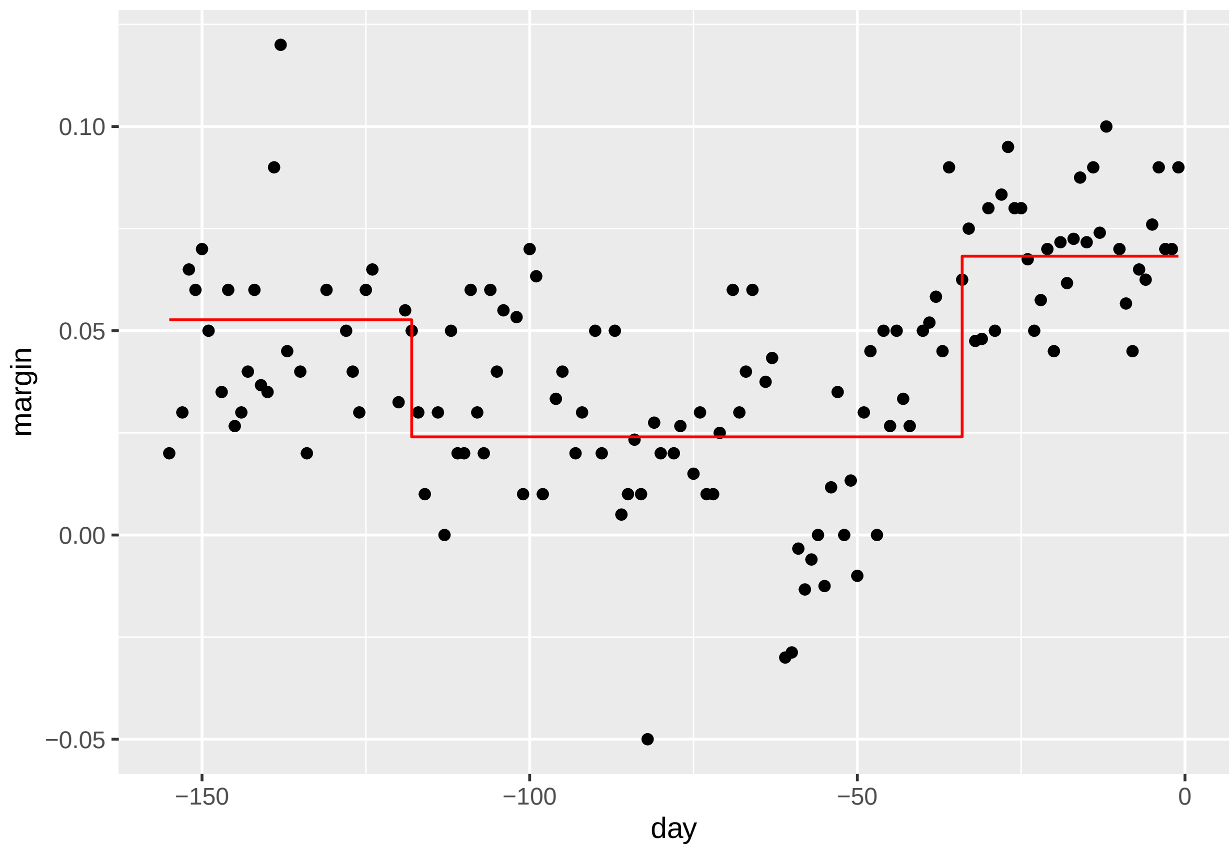
The cp parameter is just a value that tells us when we should stop - if we don't gain enough in decrease mse, then we won't add to the tree. This is similar to the penalty value in ridge and lasso. Thus, we see that tree depth of 3 gives the same results as tree depth of 2 because adding that extra level with a cost complexity of 0.1 does not decrease mse enough to justify the added complexity of the tree.

```
calc_mse(3, 0.1)
```



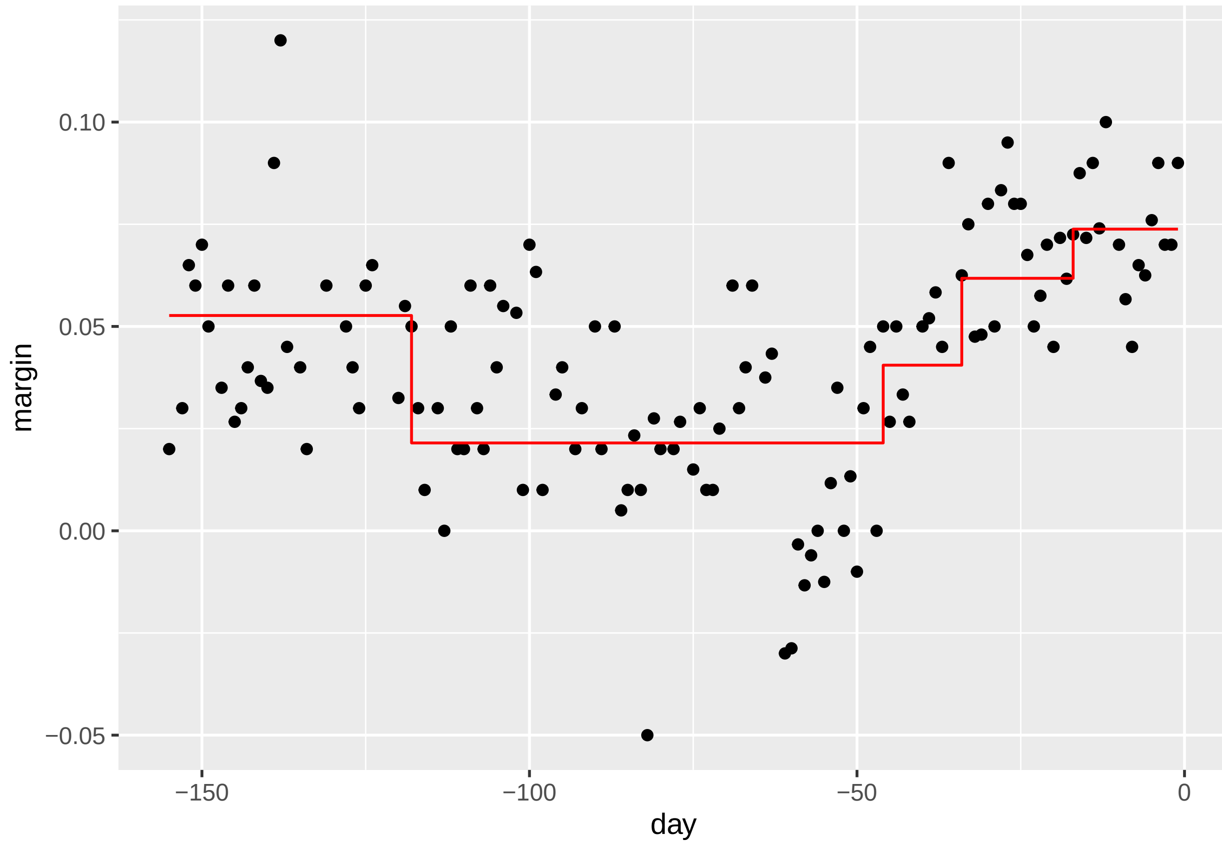

```
## [1] 0.0004262214
```

```
calc_mse(2, 0.1)
```



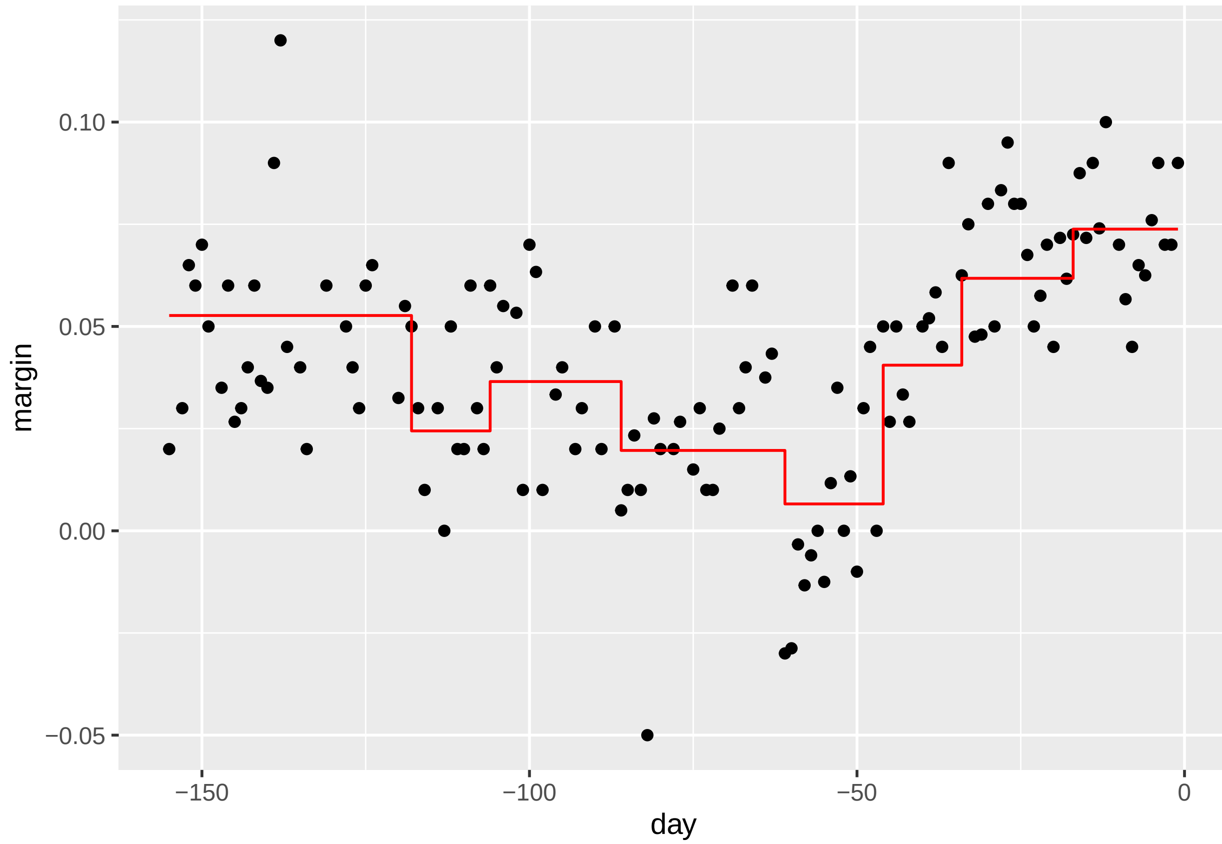

```
## [1] 0.0004262214
```

```r
calc_mse(3, 0.001) #We get 5 levels now!
```




```
## [1] 0.0003942404
```

```
calc_mse(6, 0.001) #Now we get ~8 levels
```

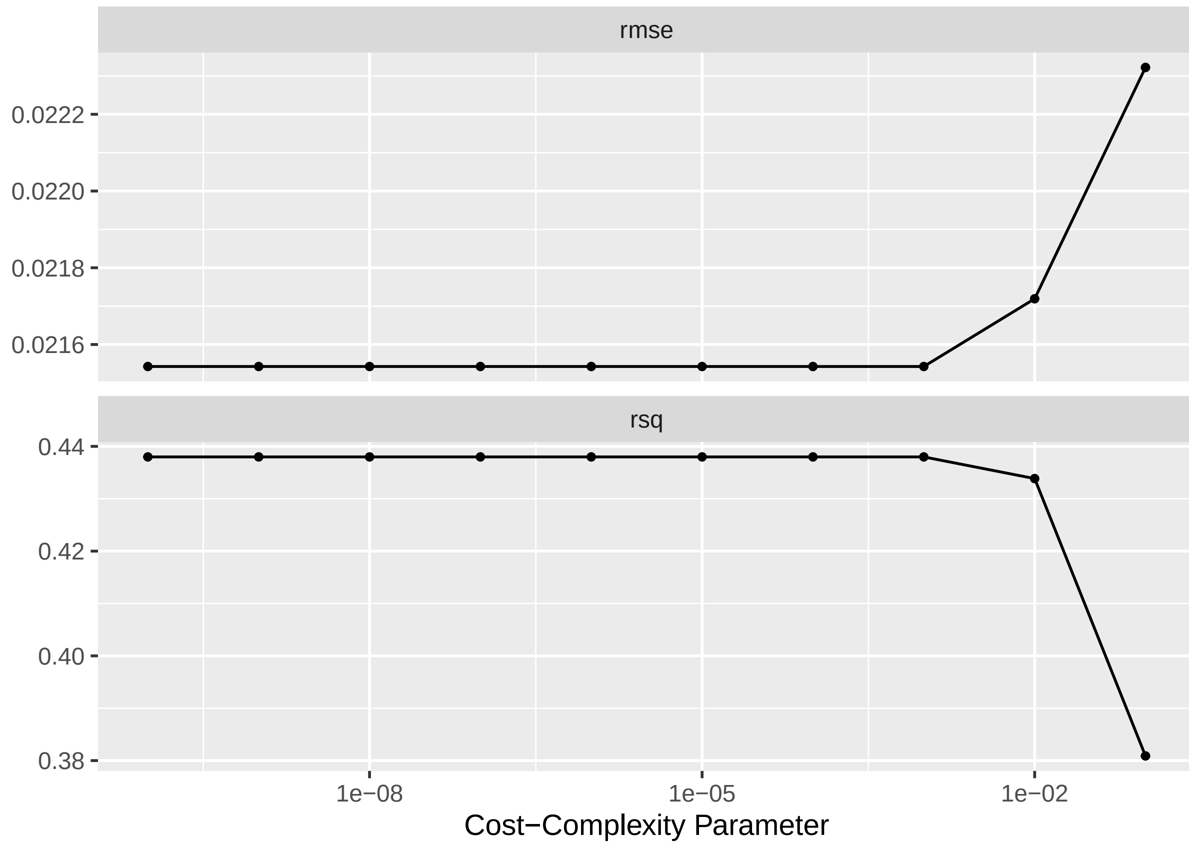



```
## [1] 0.000342623
```

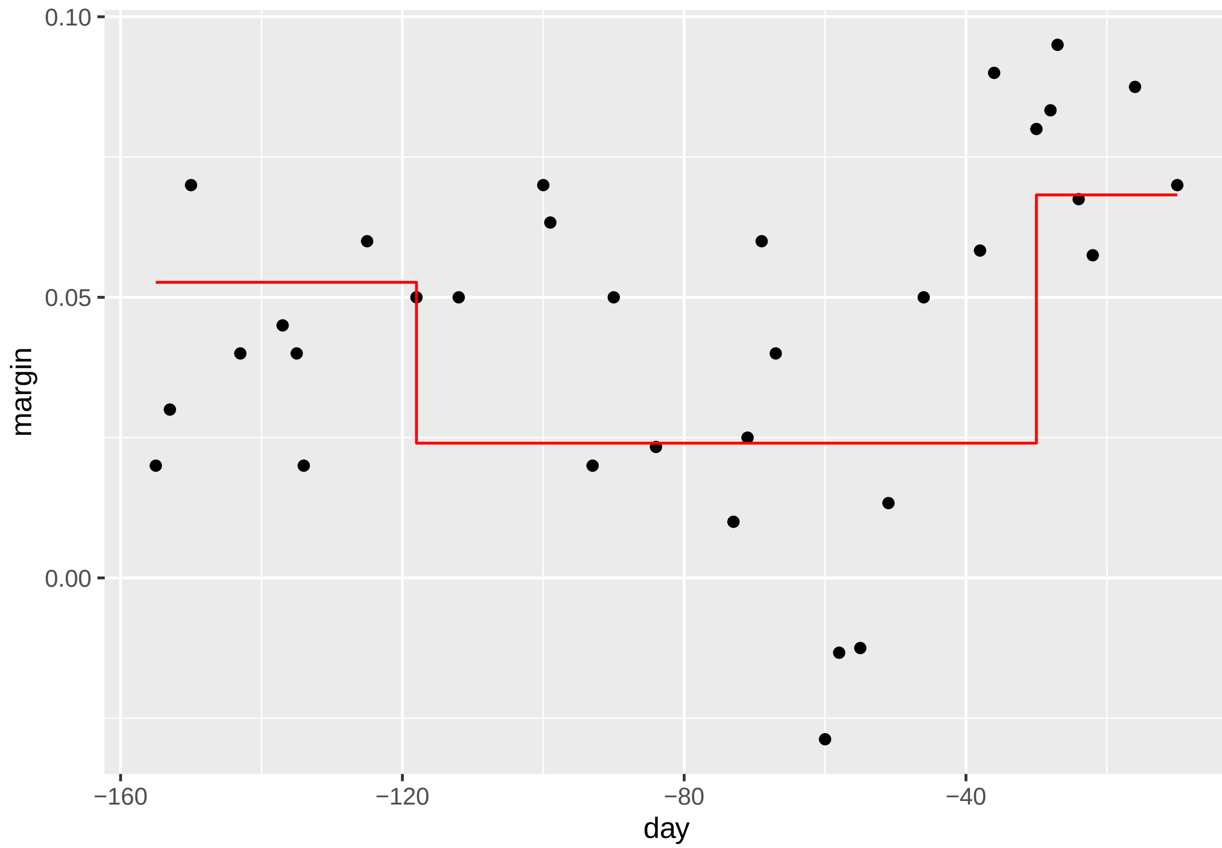
3. Using the following 10-fold cross-validation, find the optimal cp using the "one-standard-error" rule. Calculate the mse and plot the final model using your testing dataset

```
# Create the cross-validation dataset
set.seed(31416)
poll.folds <- vfold_cv(poll.train.tbl, v = 10)


poll.model <-
  decision_tree(cost_complexity=tune()) %>%
  set_mode("regression") %>%
  set_engine("rpart")

  poll.recipe <- recipe(margin ~ day, data=poll.train.tbl)


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

  poll.grid <-
    grid_regular(cost_complexity(), levels = 10)

  poll.res <-
    tune_grid(
```

```
    poll.wflow,
    resamples = poll.folds,
    grid = poll.grid)

autoplot(poll.res)
```



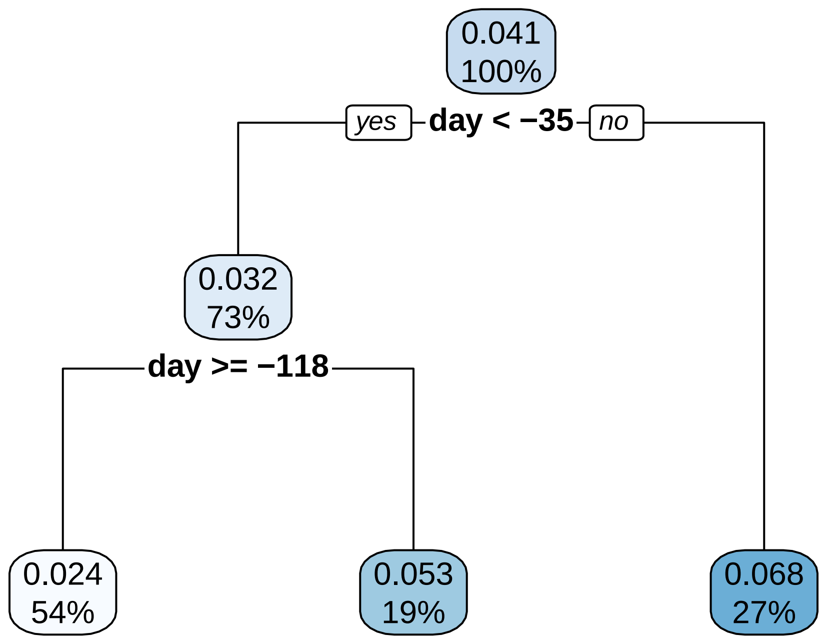

```
  best.parameter <- select_by_one_std_err(poll.res, desc(cost_complexity), metric = "rmse")
poll.final.wf <- finalize_workflow(poll.wflow, best.parameter)
poll.final.fit <- fit(poll.final.wf, data = poll.train.tbl)

#FINAL MODEL PLOTTED ON TEST DATA
augment(poll.final.fit, poll.test.tbl) %>%
    ggplot()+
    geom_point(aes(day,margin))+
    geom_step(aes(day,.pred), col="red")
```




```r
poll.final.tbl <- augment(poll.final.fit, poll.test.tbl)

#MSE
(rmse(poll.final.tbl, margin, .pred)$.estimate)^2
```

```
## [1] 0.0007345048
```

```r
  #FINAL MODEL TREE
poll.final.fit%>%
  extract_fit_engine()%>%
  rpart.plot(roundint = FALSE)
```



Back to decision trees.

We would like to revisit one of our favorite problems, digit classification, this time using decision trees.

To do that first, let's create a subset of the MNIST dataset

```
mnist <- read_mnist()
set.seed(2022)
index <- sample(nrow(mnist$train$images), 1000)
train.tbl <- as_tibble (mnist$train$images[index,]) %>%
  mutate(digit = factor(mnist$train$labels[index]))

index <- sample(nrow(mnist$test$images), 1000)
test.tbl <- as_tibble (mnist$test$images[index,]) %>%
  mutate(digit = factor(mnist$test$labels[index]))
```

And let's subset this dataset to just `1`s and `2`s

```
digits = c(1,2)

train.12.tbl = train.tbl %>%
  filter(digit %in% digits) %>%
  mutate(digit = factor(digit, levels=digits))

test.12.tbl = test.tbl %>%
```

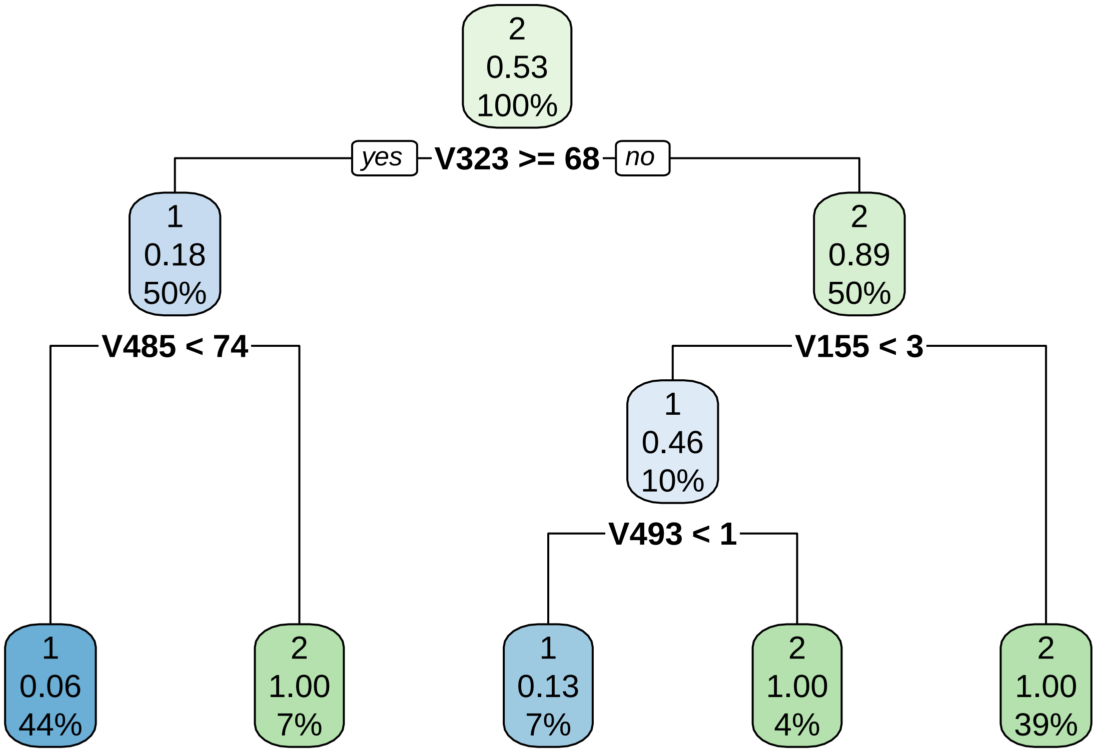
```
  filter(digit %in% digits) %>%
  mutate(digit = factor(digit, levels=digits))
```

And let's keep some plotting functions in case we need them

```
plotImage <- function(dat,size=28){
  imag <- matrix(dat,nrow=size)[,28:1]
  image(imag,col=grey.colors(256), xlab = "", ylab="")
}

plot_row <- function(tbl) {
  ntbl <- tbl %>%
    select(-digit)
  plotImage(as.matrix(ntbl))
}
```

4. Using default parameters create a decision tree that would distinguish between 1s and 2s. Visualize the decision tree using `rpart.plot`. What is the accuracy and the confusion matrix on the testing dataset?

```
digit.model <-
  decision_tree()%>%
  set_mode("classification")%>%
  set_engine("rpart")

digit.recipe <- recipe(digit ~ ., data = train.12.tbl)

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

digit.fit <- fit(digit.wflow, train.12.tbl)

digit.fit%>%
  extract_fit_engine()%>%
  rpart.plot(roundint = FALSE)
```




```
augment(digit.fit, test.12.tbl)%>%
  conf_mat(digit, .pred_class)
```

```
##           Truth
## Prediction   1   2
##          1 123  21
##          2   1  77
```

```
augment(digit.fit, test.12.tbl)%>%
  accuracy(digit, .pred_class)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.901
```

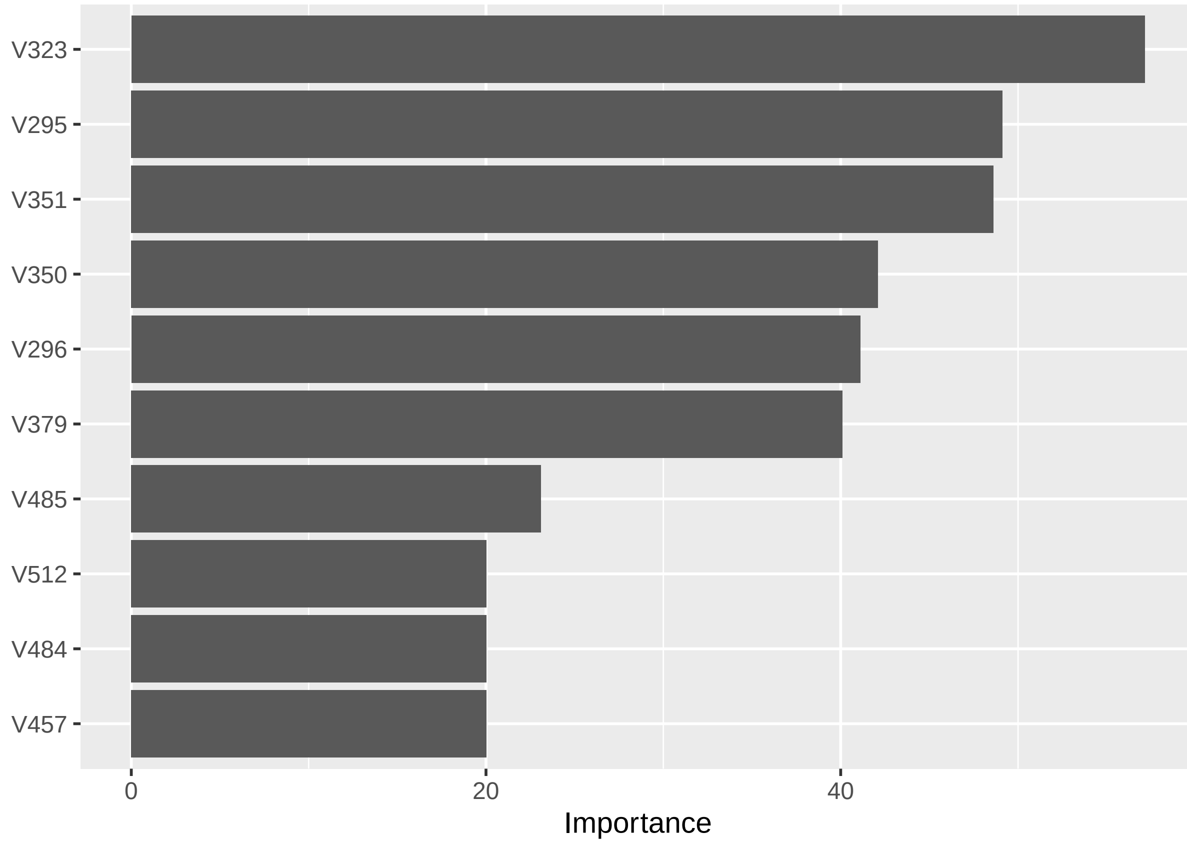
We get an accuracy of 90.1%, which is really good! We see that 2s are most often mistaken for 1s, not the other way around.

In decision trees we can quantify the importance of variables in the following. At each node a single variables is used to partition the data into two homogeneous groups and in doing so maximizes some measure of improvement. The importance of a variable x is the sum of the squared improvements over all internal nodes of the tree for which x was chosen as the partitioning variable.

Notice that in R we can use the `vip` library to calculate the importance in the following manner. Notice that we can get the information as a tibble using the function `vip:vi()`

```
library(vip)
```

```
digit.fit %>%
  extract_fit_engine() %>%
  vip::vip()
```

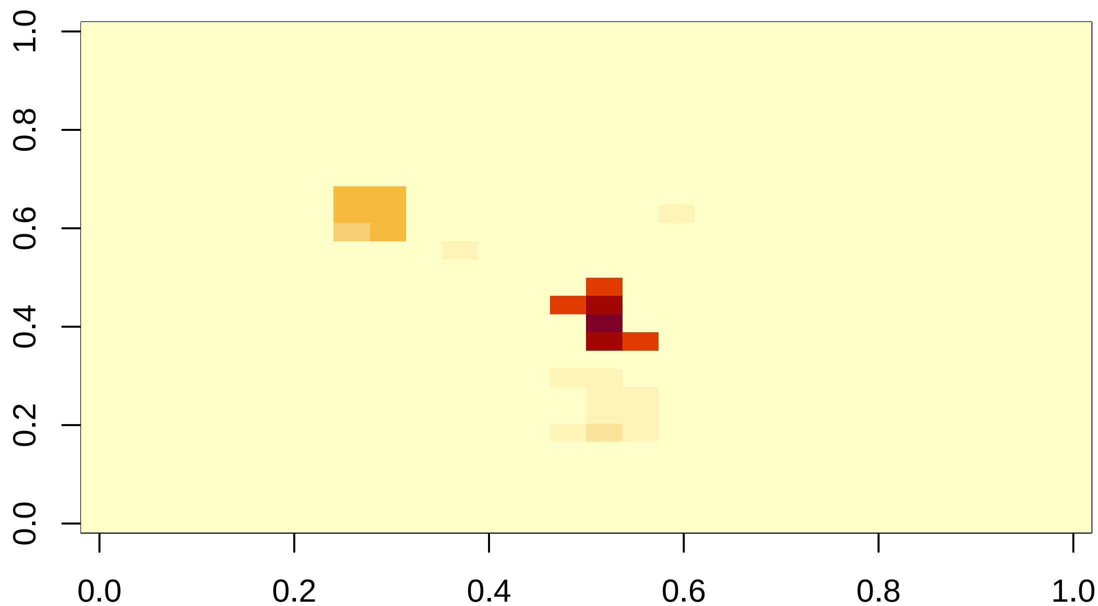



```
imp.tbl <- digit.fit %>%
  extract_fit_engine() %>%
  vip::vi()
imp.tbl
```

```
## # A tibble: 24 x 2
##    Variable Importance
##    <chr>         <dbl>
##  1 V323           57.1
##  2 V295           49.1
##  3 V351           48.6
##  4 V350           42.1
##  5 V296           41.1
##  6 V379           40.1
##  7 V485           23.1
##  8 V457           20.0
##  9 V484           20.0
## 10 V512           20.0
## # ... with 14 more rows
```

Finally we can create an image that will allow us to visualize the importance of those pixels (features)

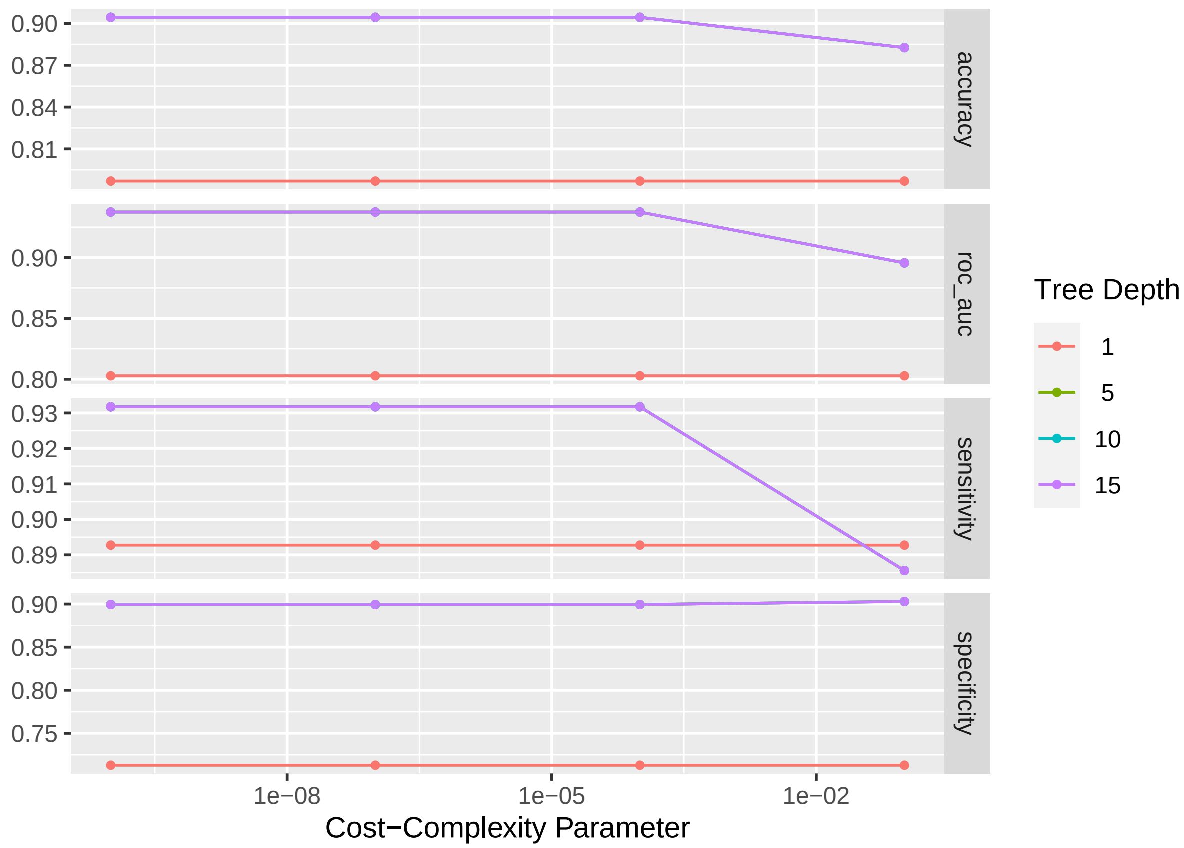
```r
imp.tbl <- imp.tbl %>%
  mutate(col=as.double(str_remove(Variable,"V")))

mat <- rep(0, 28*28)
mat[imp.tbl$col] <- imp.tbl$Importance
image(matrix(mat, 28, 28))
```



5. Find the optimal cp and tree_depth using 10-fold cross-validation and the one standard-error rule. What is your accuracy using your testing dataset? Create an image with most important features used by your model.

```r
set.seed(31416)
digit.folds <- vfold_cv(train.12.tbl, v = 10)


digit.model <-
  decision_tree(tree_depth = tune(), cost_complexity=tune()) %>%
  set_mode("classification") %>%
  set_engine("rpart")

  digit.recipe <- recipe(digit ~ ., data=train.12.tbl)


  digit.wflow <- workflow() %>%
    add_recipe(digit.recipe) %>%
    add_model(digit.model)
```

```r
digit.grid <-
  grid_regular(cost_complexity(), tree_depth(), levels = 4)

digit.res <-
  tune_grid(
    digit.wflow,
    resamples = digit.folds,
    grid = digit.grid,
    metrics = metric_set(accuracy, roc_auc, sensitivity, specificity))

autoplot(digit.res)
```




```r
best.parameters <- select_by_one_std_err(digit.res, desc(cost_complexity), tree_depth, metric = "accu
best.parameters
```

```
## # A tibble: 1 x 10
##   cost_complexity tree_depth .metric  .estimator  mean     n std_err .config
##             <dbl>      <int> <chr>    <chr>      <dbl> <int>   <dbl> <fct>
## 1          0.0001          5 accuracy binary     0.904    10  0.0203 Preprocess~
## # ... with 2 more variables: .best <dbl>, .bound <dbl>
```

```r
digit.final.wf <- finalize_workflow(digit.wflow, best.parameters)
digit.final.fit <- fit(digit.final.wf, data = train.12.tbl)

augment(digit.final.fit, test.12.tbl)%>%
  accuracy(digit, .pred_class)
```

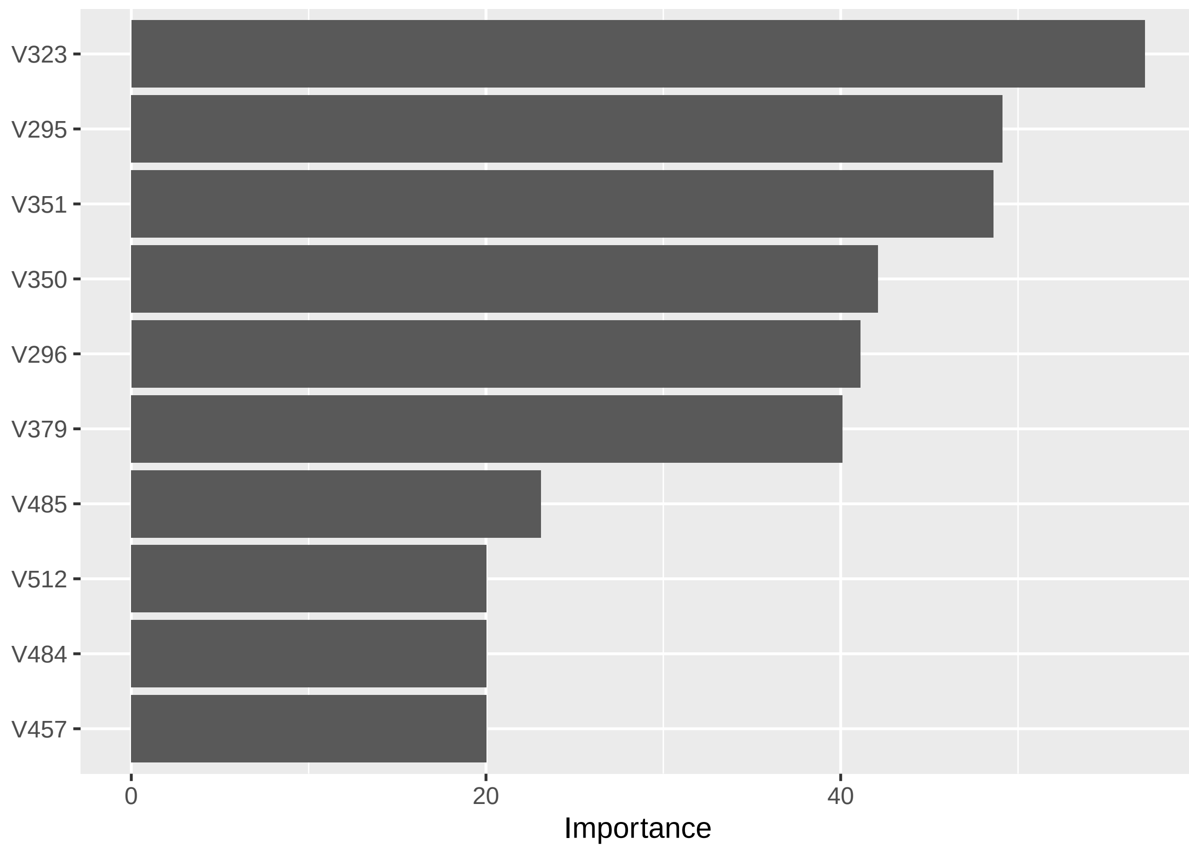
```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.901
```

```
augment(digit.final.fit, test.12.tbl)%>%
  conf_mat(digit, .pred_class)
```

```
##           Truth
## Prediction   1   2
##          1 123  21
##          2   1  77
```

```
# digit.final.fit%>%
#   extract_fit_engine()%>%
#   rpart.plot(roundint = FALSE)
```

```
digit.final.fit %>%
  extract_fit_engine() %>%
  vip::vip()
```

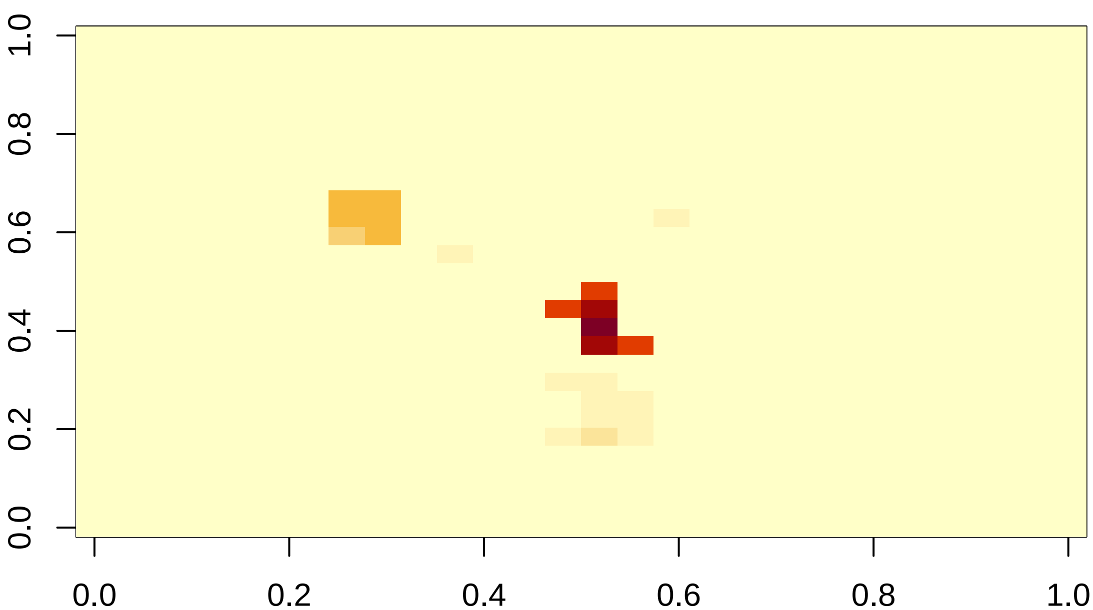



```
imp.tbl <- digit.final.fit %>%
  extract_fit_engine() %>%
  vip::vi()
imp.tbl
```

```
## # A tibble: 24 x 2
##    Variable Importance
```

```
##    <chr>          <dbl>
##   1 V323           57.1
##   2 V295           49.1
##   3 V351           48.6
##   4 V350           42.1
##   5 V296           41.1
##   6 V379           40.1
##   7 V485           23.1
##   8 V457           20.0
##   9 V484           20.0
##  10 V512           20.0
## # ... with 14 more rows
```

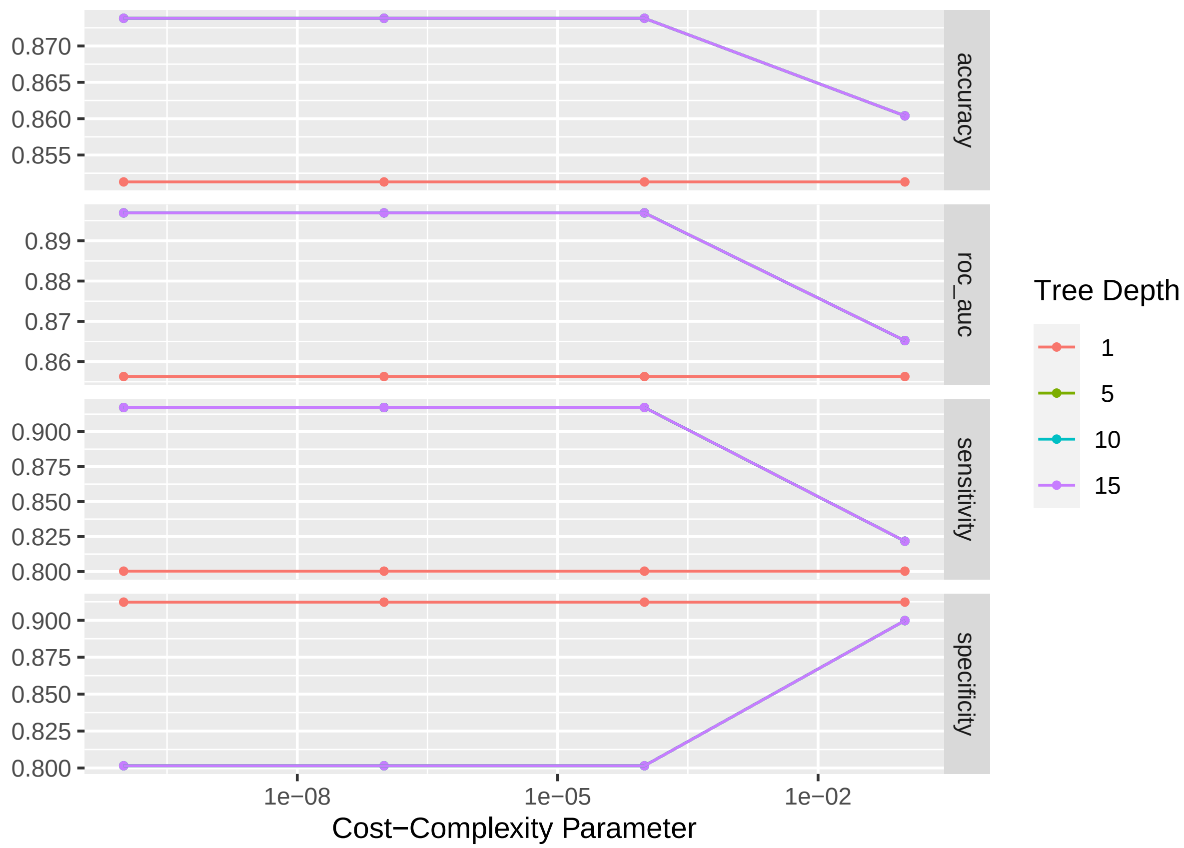
```
imp.tbl <- imp.tbl %>%
  mutate(col=as.double(str_remove(Variable,"V")))

mat <- rep(0, 28*28)
mat[imp.tbl$col] <- imp.tbl$Importance
image(matrix(mat, 28, 28))
```



We see an accuracy of 90.1%% from this optimized model, which is overall really good. The image is essentially the same to our first image (when we didn't use cross-validation).

6. Create an optimal decision tree (e.g. by optimizing `cp` and `tree_depth` for the pair of digits that you were given in your first challenge). What is your accuracy and confusion matrix using your testing dataset? Plot a couple of digits that get missclassified. Create an image with most important features used by your model.

```r
digits = c(2,3)

train.23.tbl = train.tbl %>%
  filter(digit %in% digits) %>%
  mutate(digit = factor(digit, levels=digits))

test.23.tbl = test.tbl %>%
  filter(digit %in% digits) %>%
  mutate(digit = factor(digit, levels=digits))

set.seed(31416)
digit.folds <- vfold_cv(train.23.tbl, v = 10)


digit.model <-
  decision_tree(tree_depth = tune(), cost_complexity=tune()) %>%
  set_mode("classification") %>%
  set_engine("rpart")

  digit.recipe <- recipe(digit ~ ., data=train.23.tbl)


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

  digit.grid <-
    grid_regular(cost_complexity(), tree_depth(), levels = 4)

  digit.res <-
    tune_grid(
      digit.wflow,
      resamples = digit.folds,
      grid = digit.grid,
      metrics = metric_set(accuracy, roc_auc, sensitivity, specificity))

  autoplot(digit.res)
```




```
best.parameters <- select_by_one_std_err(digit.res, desc(cost_complexity), tree_depth, metric = "accu
best.parameters
```

```
## # A tibble: 1 x 10
##   cost_complexity tree_depth .metric  .estimator  mean     n std_err .config
##             <dbl>      <int> <chr>    <chr>      <dbl> <int>   <dbl> <fct>
## 1             0.1          5 accuracy binary     0.860    10  0.0233 Preprocess~
## # ... with 2 more variables: .best <dbl>, .bound <dbl>
```

```
digit.final.wf <- finalize_workflow(digit.wflow, best.parameters)
digit.final.fit <- fit(digit.final.wf, data = train.23.tbl)

augment(digit.final.fit, test.23.tbl)%>%
  accuracy(digit, .pred_class)
```

```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy binary         0.901
```

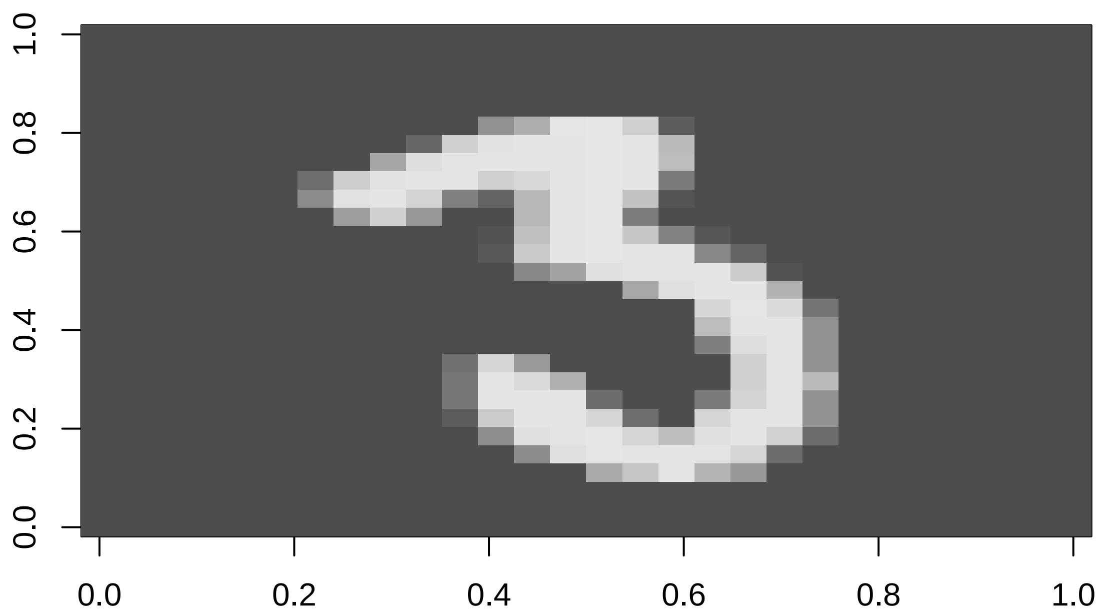
```
augment(digit.final.fit, test.23.tbl)%>%
  conf_mat(digit, .pred_class)
```

```
##           Truth
## Prediction  2  3
##          2 87  8
##          3 11 86
```

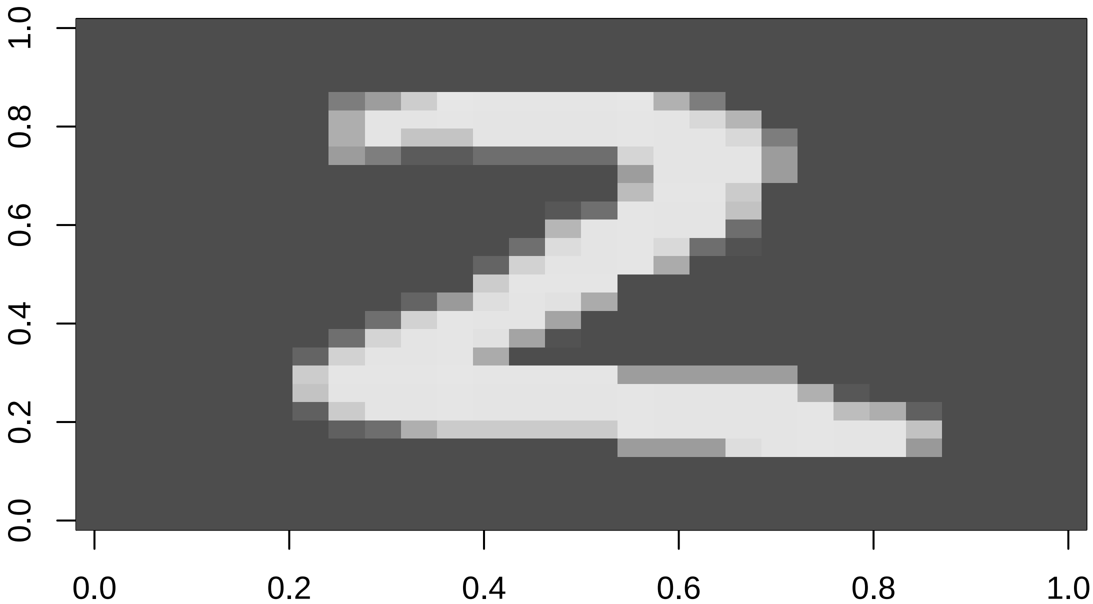
When we use a tree model on my digits (2s and 3s), we get accuracy of 90.1% again%, which is the same as the 1s and 2s accuracy.

We also see a confusion matrix that does pretty well - 8 3s are classified as 2s and 11 2s are classified as 3s.

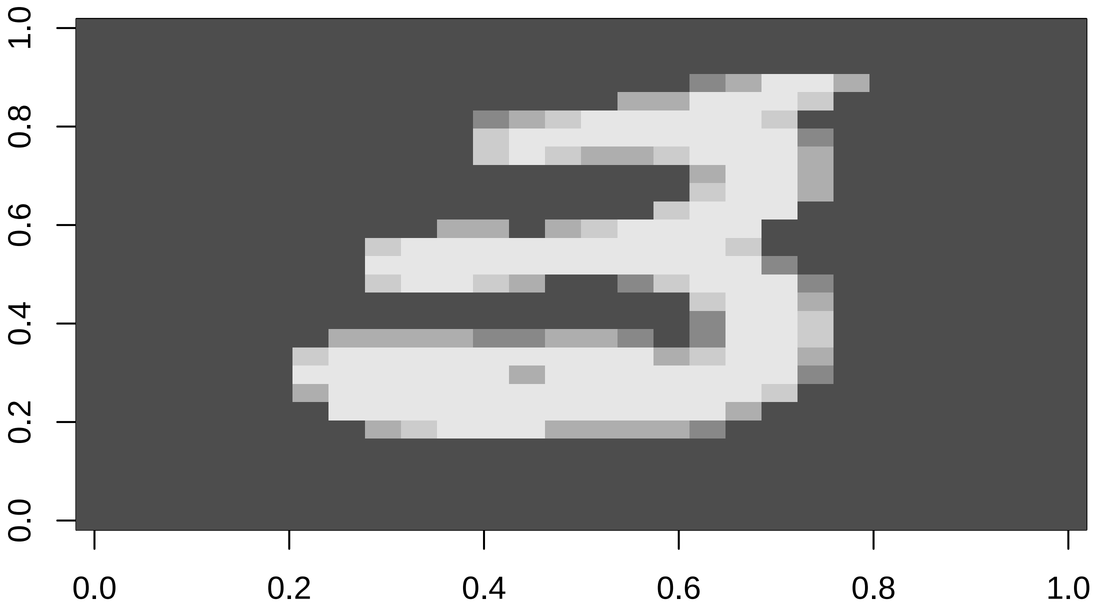
```
errors <- augment(digit.final.fit, test.23.tbl)%>%
  select(785:788, 1:784)%>%
  filter(digit != .pred_class)%>%
  select(-(2:4))

errors%>%
  slice(1)%>%
  plot_row()
```




```
errors%>%
  slice(2)%>%
  plot_row()
```

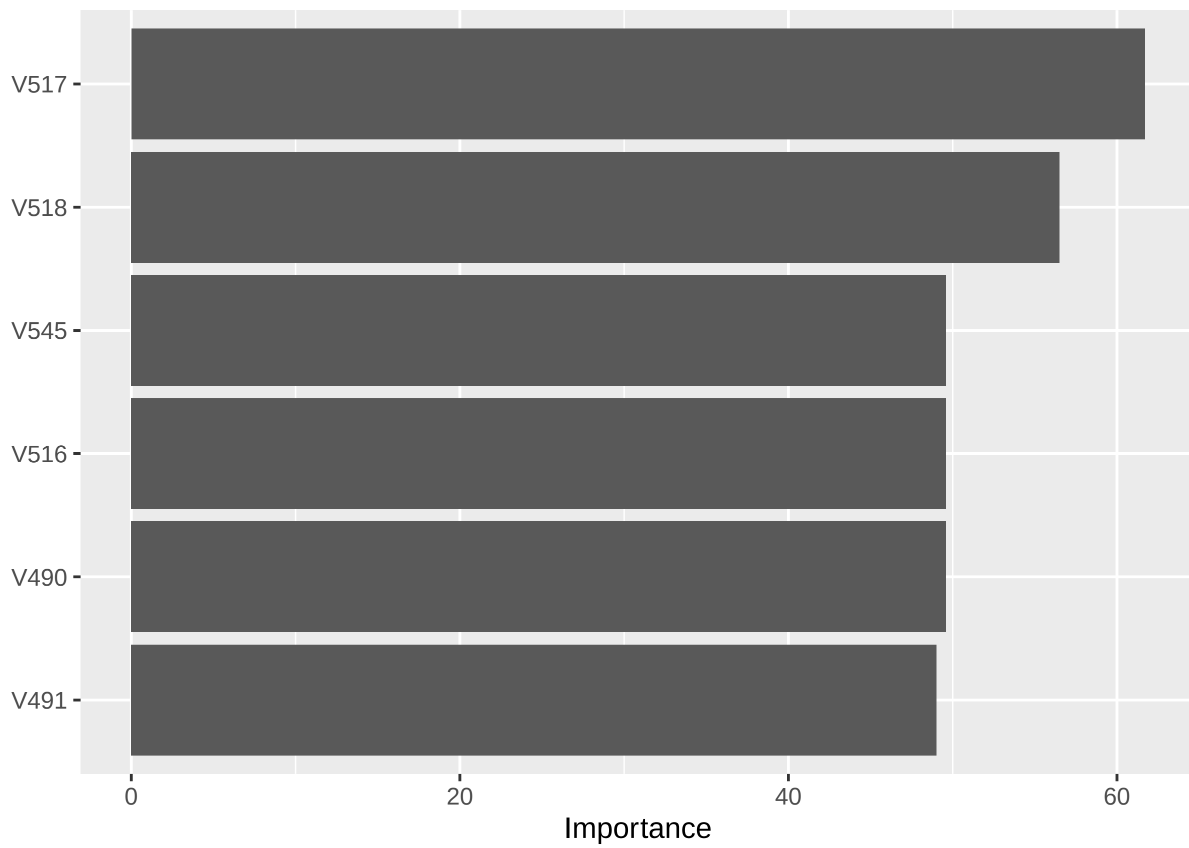



```
errors%>%
  slice(3)%>%
  plot_row()
```



Plotting Most Important Pixels

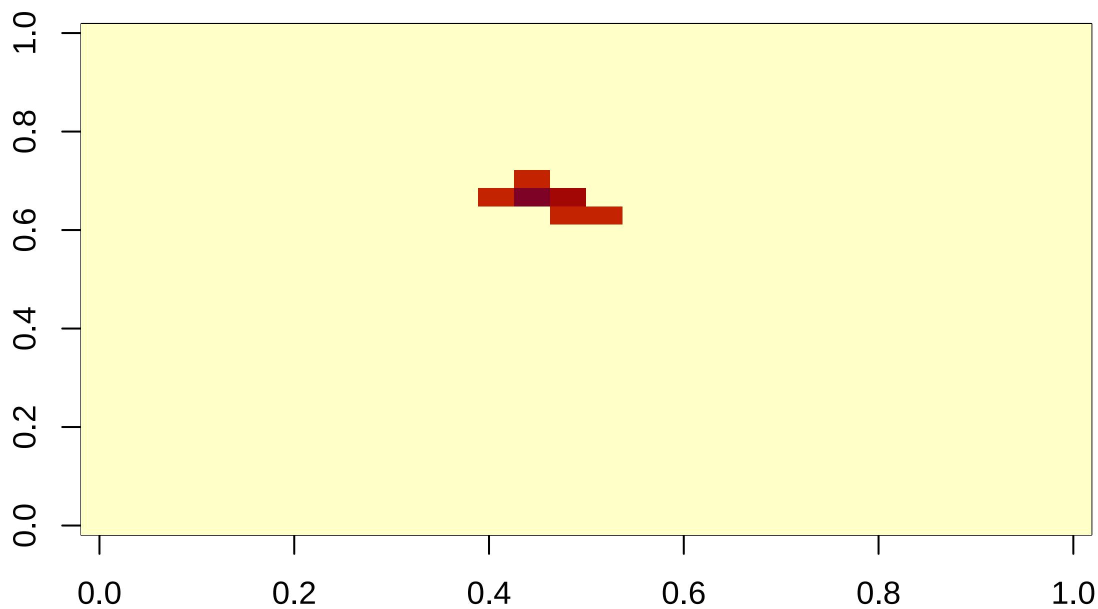
```
digit.final.fit %>%
  extract_fit_engine() %>%
  vip::vip()
```




```r
imp.tbl <- digit.final.fit %>%
  extract_fit_engine() %>%
  vip::vi()
imp.tbl
```

```
## # A tibble: 6 x 2
##   Variable Importance
##   <chr>         <dbl>
## 1 V517           61.7
## 2 V518           56.5
## 3 V490           49.6
## 4 V516           49.6
## 5 V545           49.6
## 6 V491           49.0
```

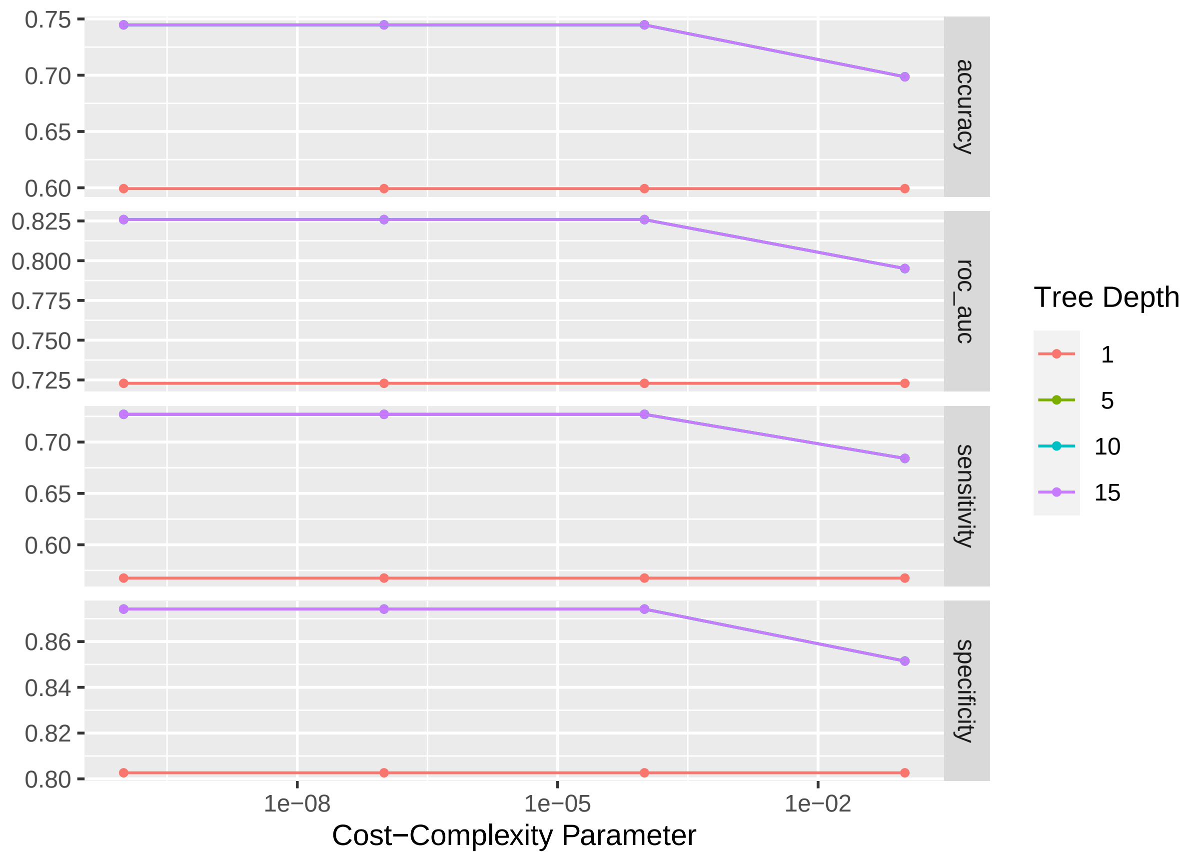
```r
imp.tbl <- imp.tbl %>%
  mutate(col=as.double(str_remove(Variable,"V")))

mat <- rep(0, 28*28)
mat[imp.tbl$col] <- imp.tbl$Importance
image(matrix(mat, 28, 28))
```



7. Create a new dataset by adding 5s to the mix (or another digit, in case 5 was in your original pair of digits). Repeat the steps outlined in exercise 6 for this new dataset.

```
digits = c(2,3,5)

train.235.tbl = train.tbl %>%
  filter(digit %in% digits) %>%
  mutate(digit = factor(digit, levels=digits))

test.235.tbl = test.tbl %>%
  filter(digit %in% digits) %>%
  mutate(digit = factor(digit, levels=digits))

set.seed(31416)
digit.folds <- vfold_cv(train.235.tbl, v = 10)


digit.model <-
  decision_tree(tree_depth = tune(), cost_complexity=tune()) %>%
  set_mode("classification") %>%
  set_engine("rpart")

  digit.recipe <- recipe(digit ~ ., data=train.235.tbl)


  digit.wflow <- workflow() %>%
    add_recipe(digit.recipe) %>%
```

```
    add_model(digit.model)

  digit.grid <-
    grid_regular(cost_complexity(), tree_depth(), levels = 4)

  digit.res <-
    tune_grid(
      digit.wflow,
      resamples = digit.folds,
      grid = digit.grid,
      metrics = metric_set(accuracy, roc_auc, sensitivity, specificity))

  autoplot(digit.res)
```




```
  best.parameters <- select_by_one_std_err(digit.res, desc(cost_complexity), tree_depth, metric = "accu
  best.parameters
```

```
## # A tibble: 1 x 10
##   cost_complexity tree_depth .metric  .estimator  mean     n std_err .config
##             <dbl>      <int> <chr>    <chr>      <dbl> <int>   <dbl> <fct>
## 1          0.0001          5 accuracy multiclass 0.745    10  0.0196 Preprocess~
## # ... with 2 more variables: .best <dbl>, .bound <dbl>
```

```
digit.final.wf <- finalize_workflow(digit.wflow, best.parameters)
digit.final.fit <- fit(digit.final.wf, data = train.235.tbl)

augment(digit.final.fit, test.235.tbl)%>%
```

```
  accuracy(digit, .pred_class)
```

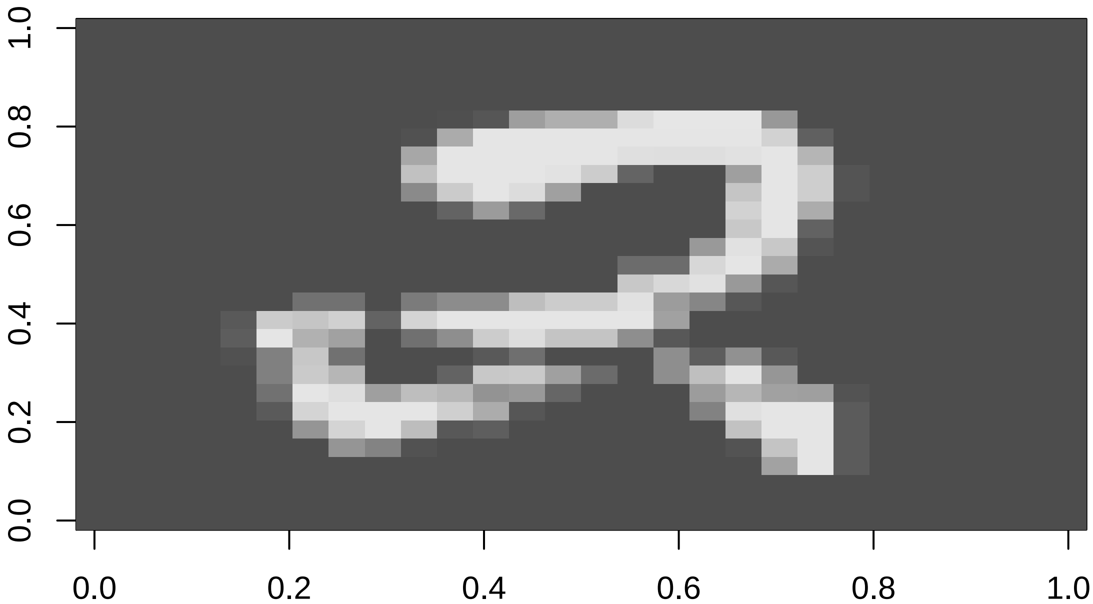
```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.809
```

```
augment(digit.final.fit, test.235.tbl)%>%
  conf_mat(digit, .pred_class)
```

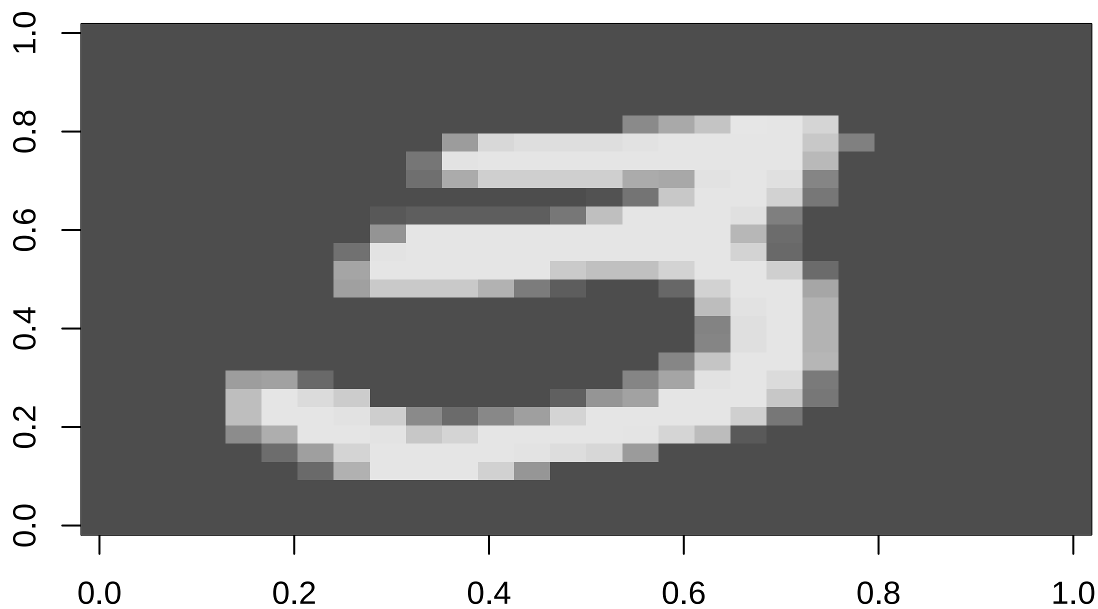
```
##           Truth
## Prediction  2  3  5
##          2 94 19 12
##          3  1 62  5
##          5  3 13 69
```

When we add 5s in, we see an accuracy of 80.9%, which is better than we were able to do in the challenge. When looking at the confusion matrix, we see that a lot of 3s and 5s got mixed up for 2s, in addition to some 3s getting predicted as 5s.

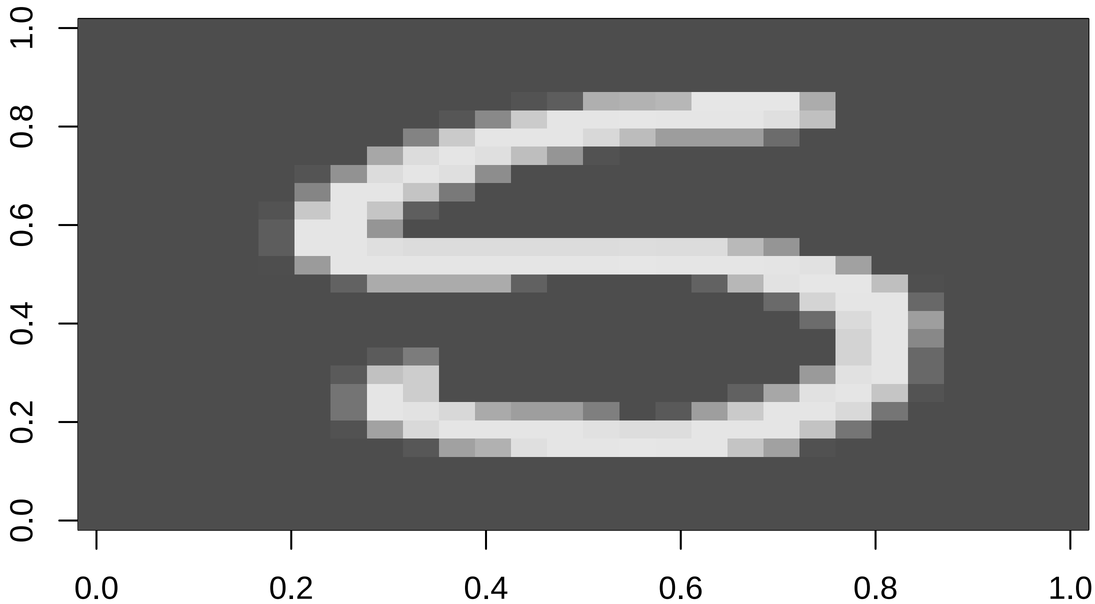
```
errors.235 <- augment(digit.final.fit, test.235.tbl)%>%
  select(785:788, 1:784)%>%
  filter(digit != .pred_class)%>%
  select(-(2:4))

#Plot a 2 that was wrong
errors.235%>%
  slice(12)%>%
  plot_row()
```




```r
#Plot a 3 that was wrong
errors.235%>%
  slice(2)%>%
  plot_row()
```

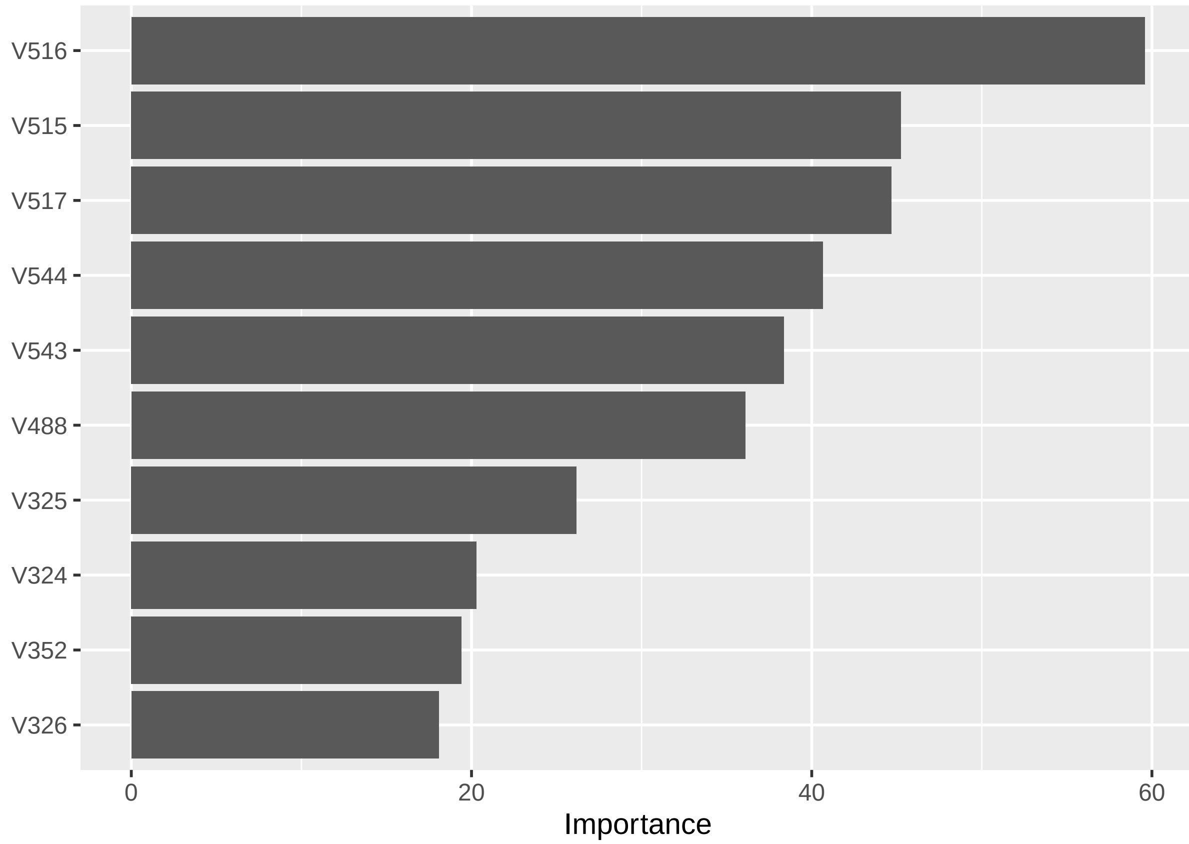



```r
#Plot a 5 that was wrong
errors.235%>%
  slice(3)%>%
  plot_row()
```



Plotting Most Important Pixels

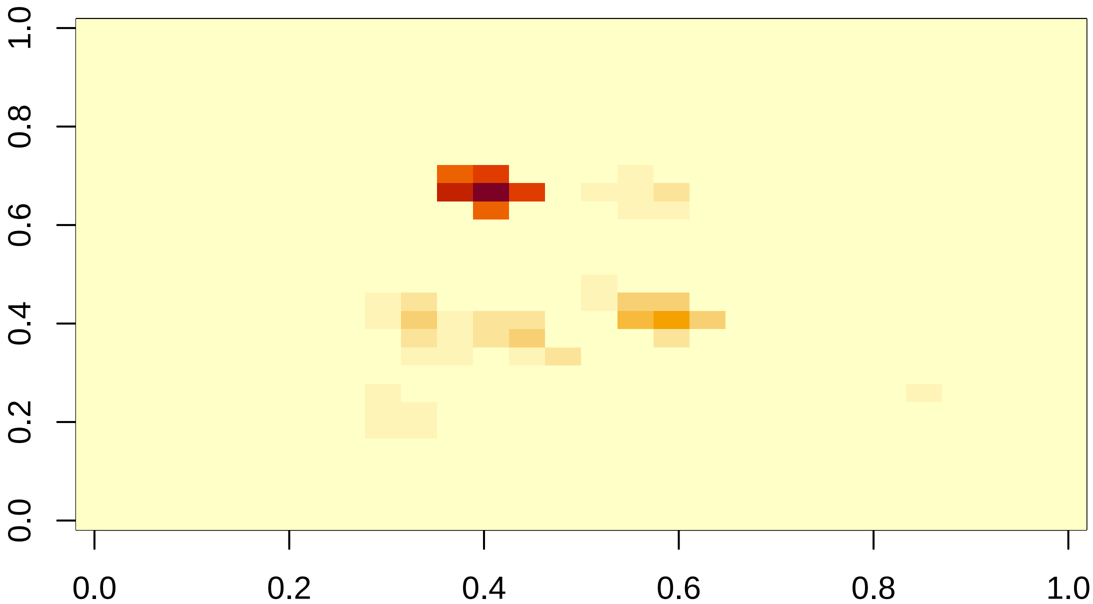
```
digit.final.fit %>%
  extract_fit_engine() %>%
  vip::vip()
```




```
imp.tbl <- digit.final.fit %>%
  extract_fit_engine() %>%
  vip::vi()
imp.tbl
```

```
## # A tibble: 49 x 2
##    Variable Importance
##    <chr>         <dbl>
##  1 V516           59.6
##  2 V515           45.3
##  3 V517           44.7
##  4 V544           40.7
##  5 V543           38.4
##  6 V488           36.1
##  7 V325           26.2
##  8 V324           20.3
##  9 V352           19.4
## 10 V326           18.1
## # ... with 39 more rows
```

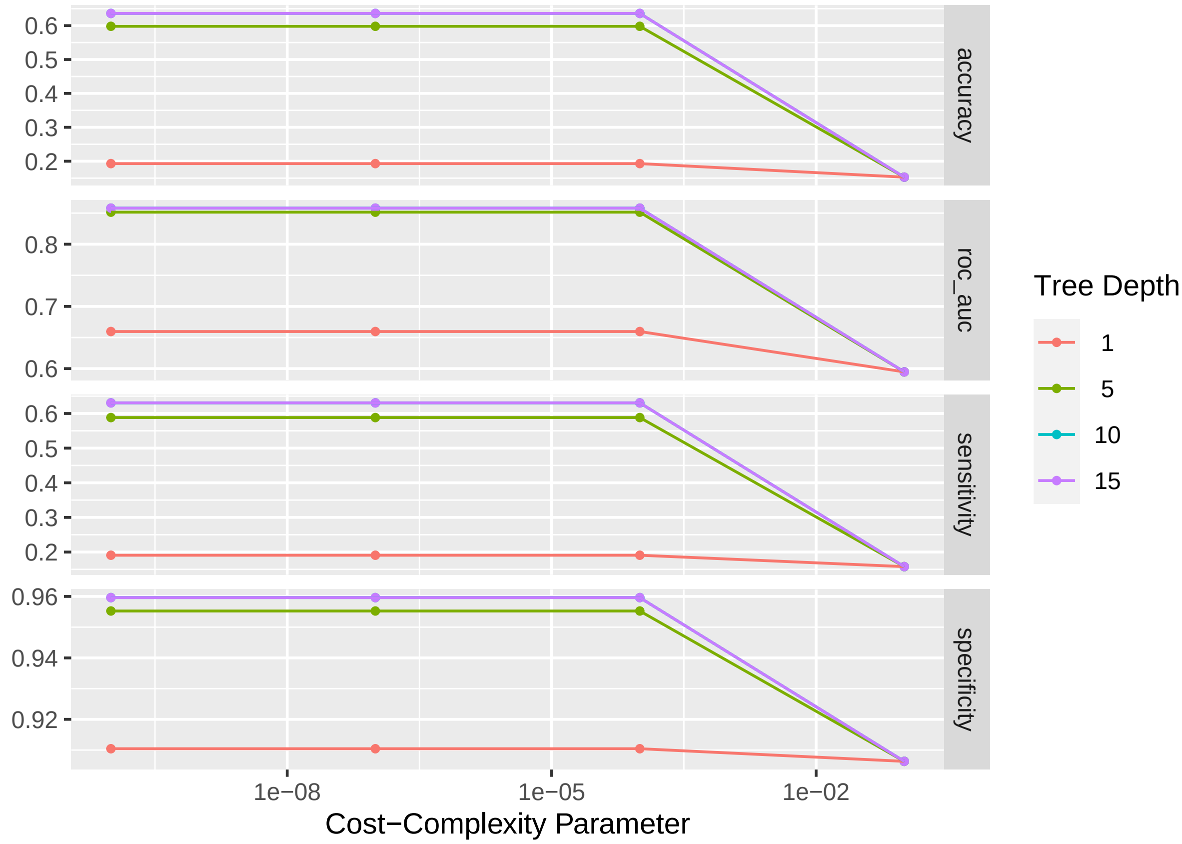
```
imp.tbl <- imp.tbl %>%
  mutate(col=as.double(str_remove(Variable,"V")))

mat <- rep(0, 28*28)
mat[imp.tbl$col] <- imp.tbl$Importance
image(matrix(mat, 28, 28))
```



8. This time train an optimal classification tree using `train.tbl` and evaluate using `test.tbl` for identifying the 10 digits by repeating the steps from exercise 6. What pairs of digits get confused the most? Plot a couple of them.

```
set.seed(31416)
digit.folds <- vfold_cv(train.tbl, v = 10)


digit.model <-
  decision_tree(tree_depth = tune(), cost_complexity=tune()) %>%
  set_mode("classification") %>%
  set_engine("rpart")

  digit.recipe <- recipe(digit ~ ., data=train.tbl)


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

  digit.grid <-
    grid_regular(cost_complexity(), tree_depth(), levels = 4)

  digit.res <-
    tune_grid(
      digit.wflow,
      resamples = digit.folds,
```

```
    grid = digit.grid,
    metrics = metric_set(accuracy, roc_auc, sensitivity, specificity))

  autoplot(digit.res)
```




```
  best.parameters <- select_by_one_std_err(digit.res, desc(cost_complexity), tree_depth, metric = "accu
  best.parameters
```

```
## # A tibble: 1 x 10
##   cost_complexity tree_depth .metric  .estimator  mean     n std_err .config
##             <dbl>      <int> <chr>    <chr>      <dbl> <int>   <dbl> <fct>
## 1          0.0001         10 accuracy multiclass 0.636    10  0.0190 Preprocess~
## # ... with 2 more variables: .best <dbl>, .bound <dbl>
```

```
digit.final.wf <- finalize_workflow(digit.wflow, best.parameters)
digit.final.fit <- fit(digit.final.wf, data = train.tbl)

augment(digit.final.fit, test.tbl)%>%
  accuracy(digit, .pred_class)
```

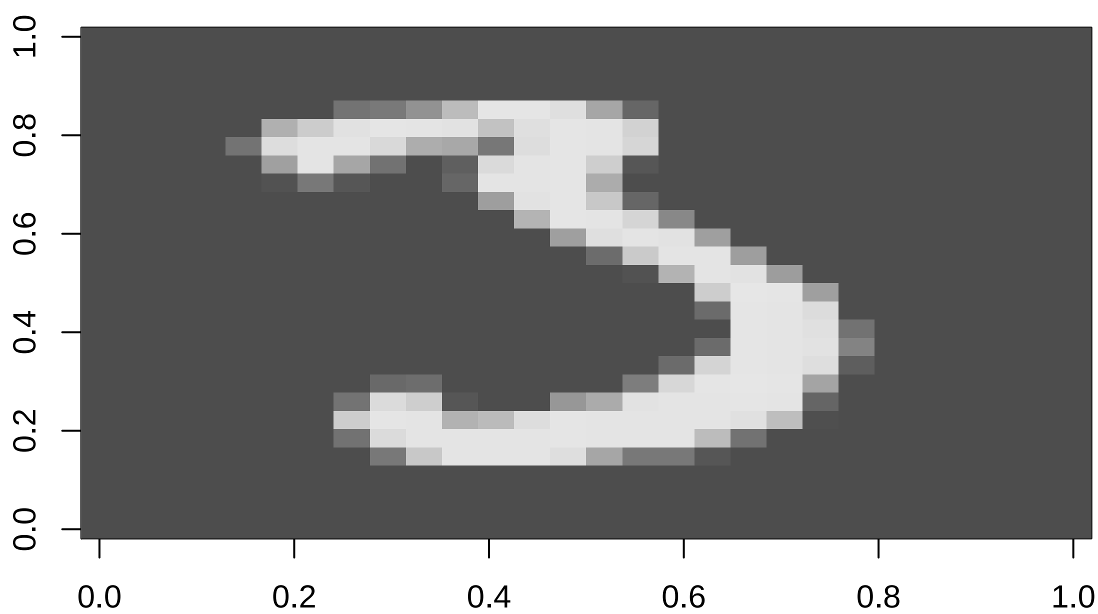
```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.663
```

```
augment(digit.final.fit, test.tbl)%>%
  conf_mat(digit, .pred_class)
```

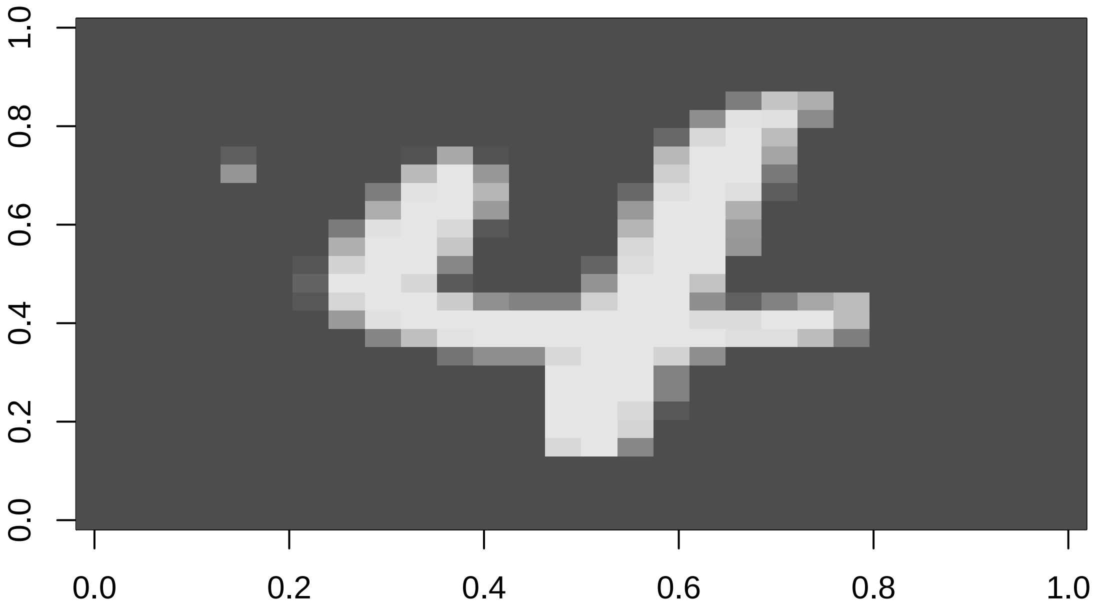
```
##           Truth
## Prediction   0   1   2   3   4   5   6   7   8   9
##          0  73   0   3   6   4   6   4   1   6   1
##          1   0 114   3   2   0   0   0   3   3   0
##          2   5   5  70   8   7   1  14   5   7   2
##          3   0   2   2  47   1   6   0   0   2   1
##          4   4   0   1   0  42   1   2   2   9  11
##          5   2   1   1  22  11  58   3   1   5  19
##          6   2   1   7   2  18   7  62   4   9   2
##          7   3   0   5   2   1   2   4  77   0   1
##          8   0   1   1   4   5   5  10   0  49   5
##          9   0   0   5   1  13   0   1   9   2  71
```

We now have all 10 digits accounted for, and we see an accuracy of 66.3%, which is pretty good given there are 10 possibilities! By looking at the confusion matrix, we see that the 3s and 9s are often mistaken for 5s and 4s are often mistaken for 5s, 6s, and 9s.

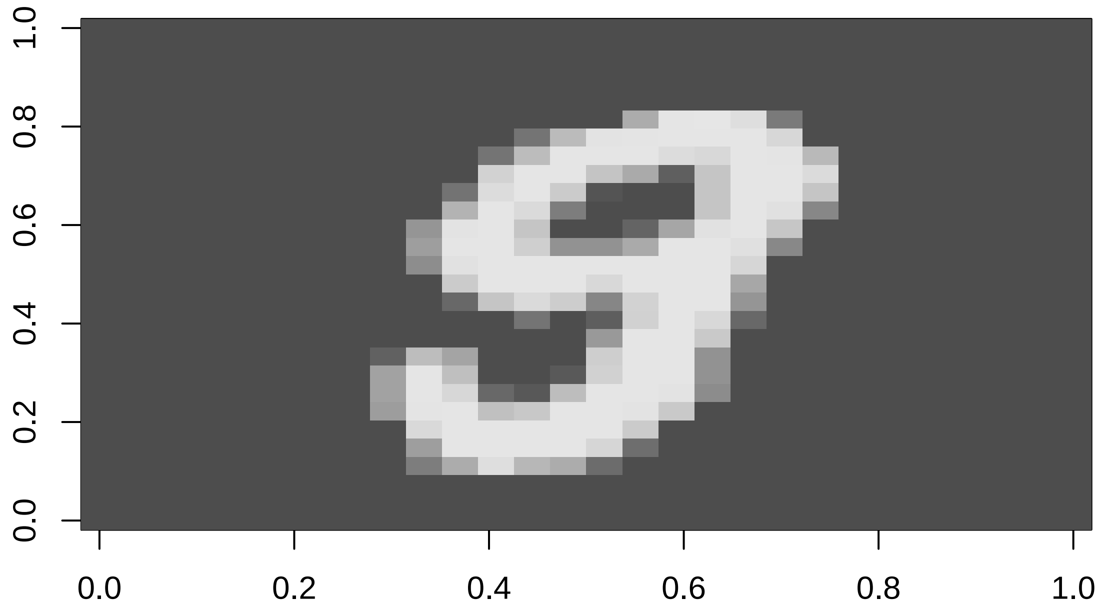
```
errors.all <- augment(digit.final.fit, test.tbl)%>%
  select(785:788, 1:784)%>%
  filter(digit != .pred_class)%>%
  select(-(2:4))

#Plot a 3 that was predicted as a 5
errors.all%>%
  slice(4)%>%
  plot_row()
```



```r
#Plot a 4 that was predicted as a 6
errors.all%>%
  slice(44)%>%
  plot_row()
```

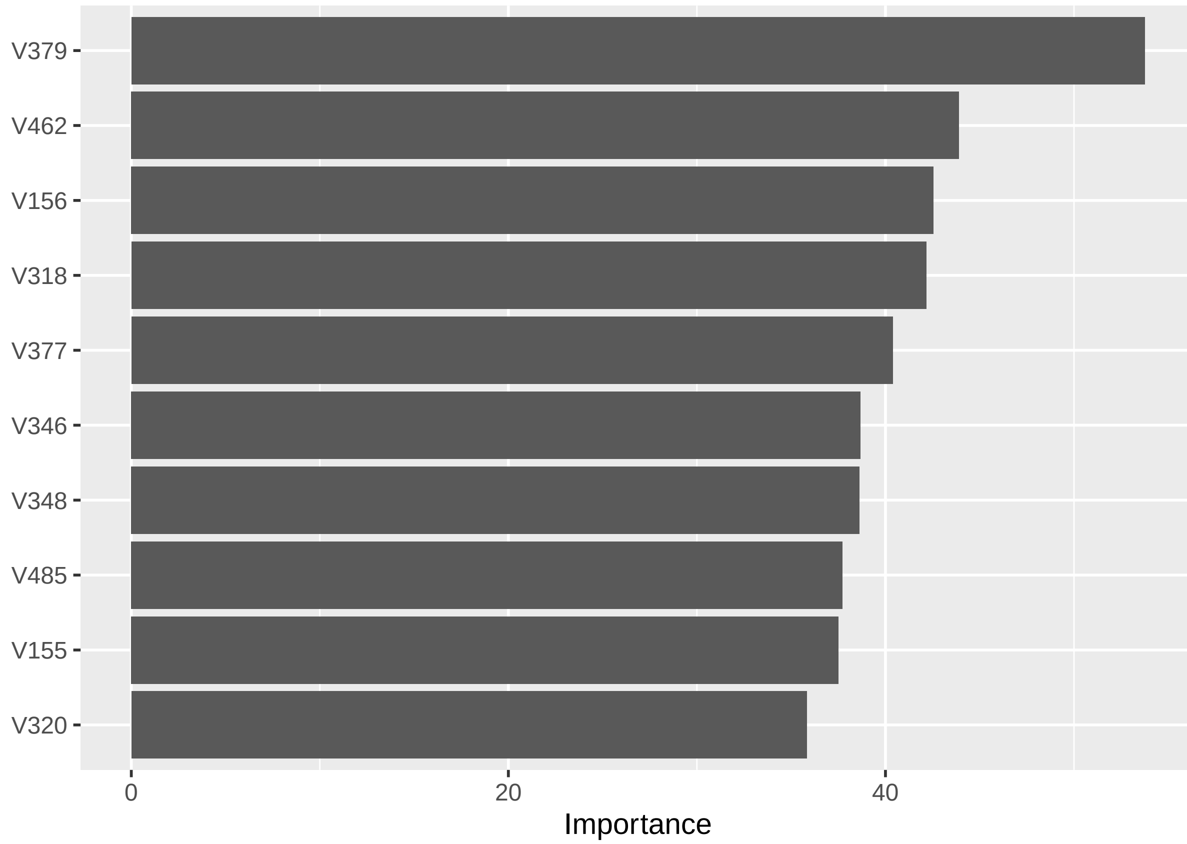



```r
#Here's a 9 that got predicted as a 5
errors.all%>%
  slice(31)%>%
  plot_row()
```



Plotting Most Important Pixels

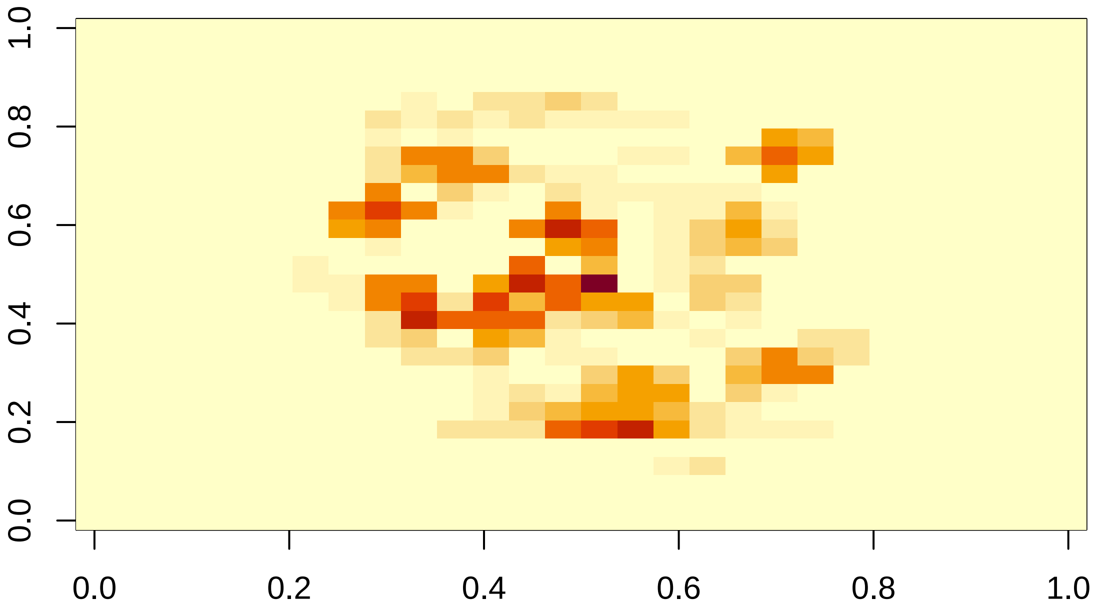
```
digit.final.fit %>%
  extract_fit_engine() %>%
  vip::vip()
```




```r
imp.tbl <- digit.final.fit %>%
  extract_fit_engine() %>%
  vip::vi()
imp.tbl
```

```
## # A tibble: 185 x 2
##    Variable Importance
##    <chr>         <dbl>
##  1 V379           53.8
##  2 V462           43.9
##  3 V156           42.6
##  4 V318           42.2
##  5 V377           40.4
##  6 V346           38.7
##  7 V348           38.6
##  8 V485           37.7
##  9 V155           37.5
## 10 V320           35.8
## # ... with 175 more rows
```

```r
imp.tbl <- imp.tbl %>%
  mutate(col=as.double(str_remove(Variable,"V")))

mat <- rep(0, 28*28)
mat[imp.tbl$col] <- imp.tbl$Importance
image(matrix(mat, 28, 28))
```

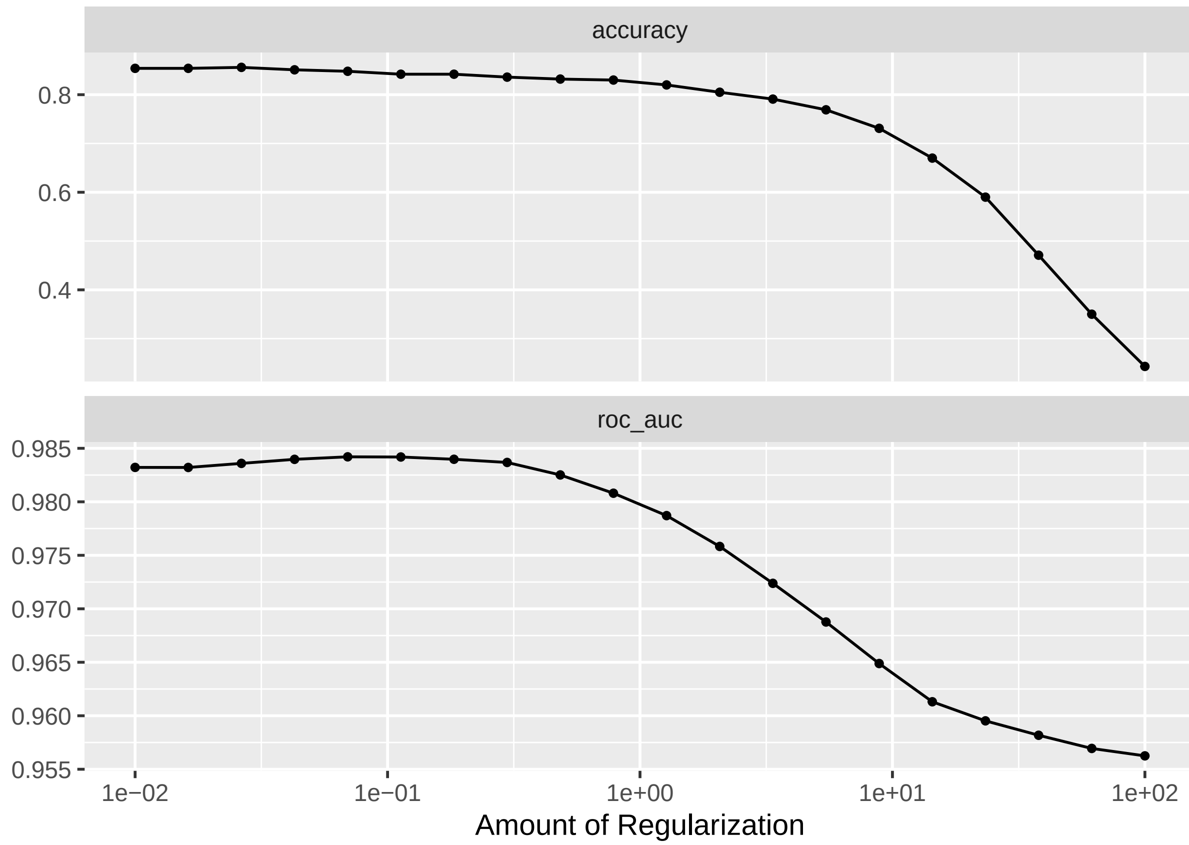


This image doesn't tell us as much becuase when classifying all 10 digits it's harder to pick out spots that are most important, but we see the concentration of most important pixels in the middle of the image, as we'd expect.

9. Same as exercise 8, but this time try a ridge model. Don't forget to optimize the penalty parameter.

```
set.seed(31416)
digit.folds <- vfold_cv(train.tbl, v = 10)

digit.ridge.model <-
  multinom_reg(mixture = 0, penalty = tune())%>%
  set_mode("classification")%>%
  set_engine("glmnet")

#I didn't normalize because all predictors should be on same scale
  digit.ridge.recipe <- recipe(digit ~ ., data=train.tbl)


  digit.ridge.wflow <- workflow() %>%
    add_recipe(digit.ridge.recipe) %>%
    add_model(digit.ridge.model)

  penalty.grid <-
    grid_regular(penalty(range = c(-2, 2)), levels = 20)


  digit.ridge.res <-
```

```
    tune_grid(
      digit.ridge.wflow,
      resamples = digit.folds,
      grid = penalty.grid)

  autoplot(digit.ridge.res)
```




```
  best.penalty <- select_by_one_std_err(digit.ridge.res, desc(penalty), metric = "accuracy")
  best.penalty
```

```
## # A tibble: 1 x 9
##   penalty .metric  .estimator  mean     n std_err .config           .best .bound
##     <dbl> <chr>    <chr>      <dbl> <int>   <dbl> <fct>             <dbl>  <dbl>
## 1  0.0695 accuracy multiclass 0.848    10  0.0112 Preprocessor1_Mo~ 0.856  0.844
```

```
digit.final.ridge.wf <- finalize_workflow(digit.ridge.wflow, best.penalty)
digit.final.ridge.fit <- fit(digit.final.ridge.wf, data = train.tbl)

augment(digit.final.ridge.fit, test.tbl)%>%
  accuracy(digit, .pred_class)
```

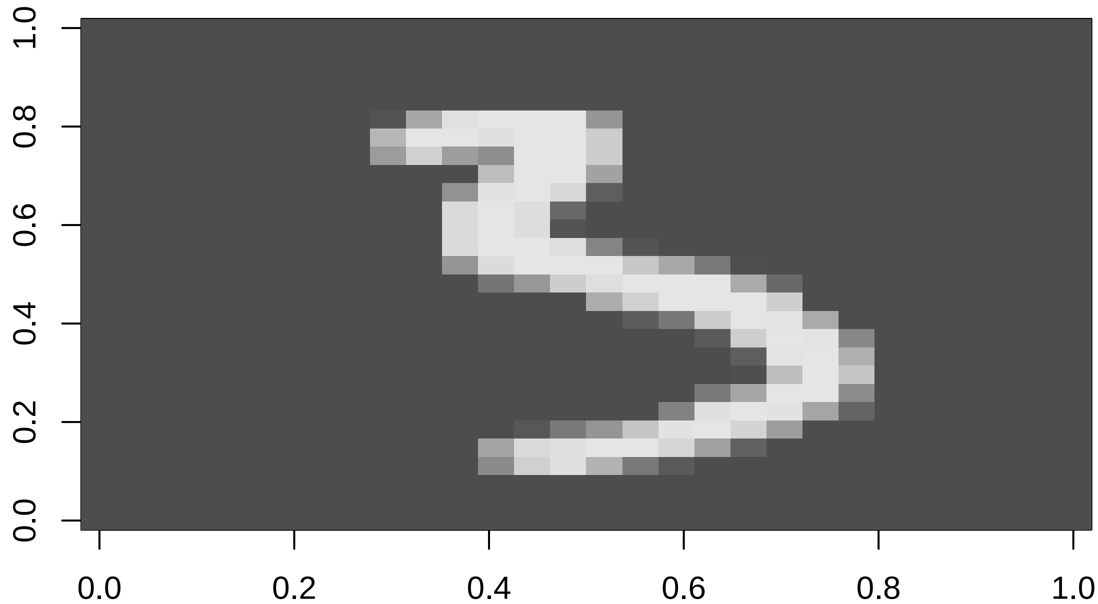
```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.887
```

```
augment(digit.final.ridge.fit, test.tbl)%>%
  conf_mat(digit, .pred_class)
```

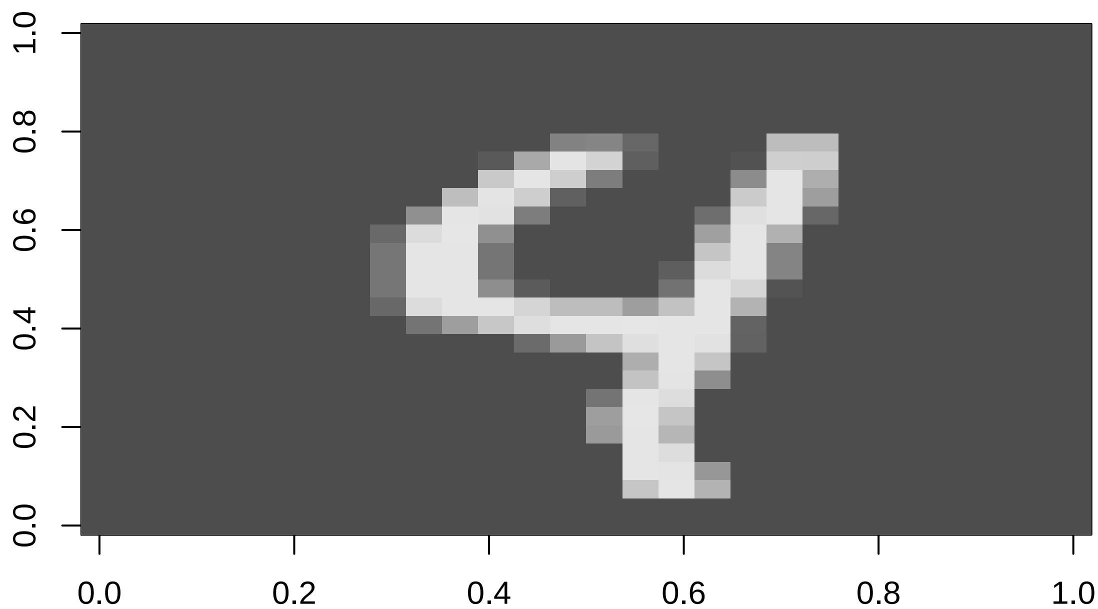
```
##           Truth
## Prediction   0   1   2   3   4   5   6   7   8   9
##          0  86   0   0   1   0   0   2   0   3   5
##          1   0 122   0   0   0   0   0   1   2   1
##          2   2   0  81   3   1   1   2   4   1   0
##          3   0   1   5  78   0   2   0   1   2   0
##          4   0   0   1   0  90   2   3   0   1   5
##          5   0   0   0   4   0  77   1   0   5   2
##          6   1   0   2   3   0   2  89   1   0   0
##          7   0   0   5   1   1   0   1  90   0   1
##          8   0   1   3   4   1   2   2   0  76   1
##          9   0   0   1   0   9   0   0   5   2  98
```

When we run a ridge model, we get an accuracy of 88.7%, which is very impressive especially in comparison to the tree model (around a 22% improvement). We see that 4s are still mixed up with 9s and 3s are predicted as 5s once again, but at a lower rate. We also see some 7s predicted as 9s.

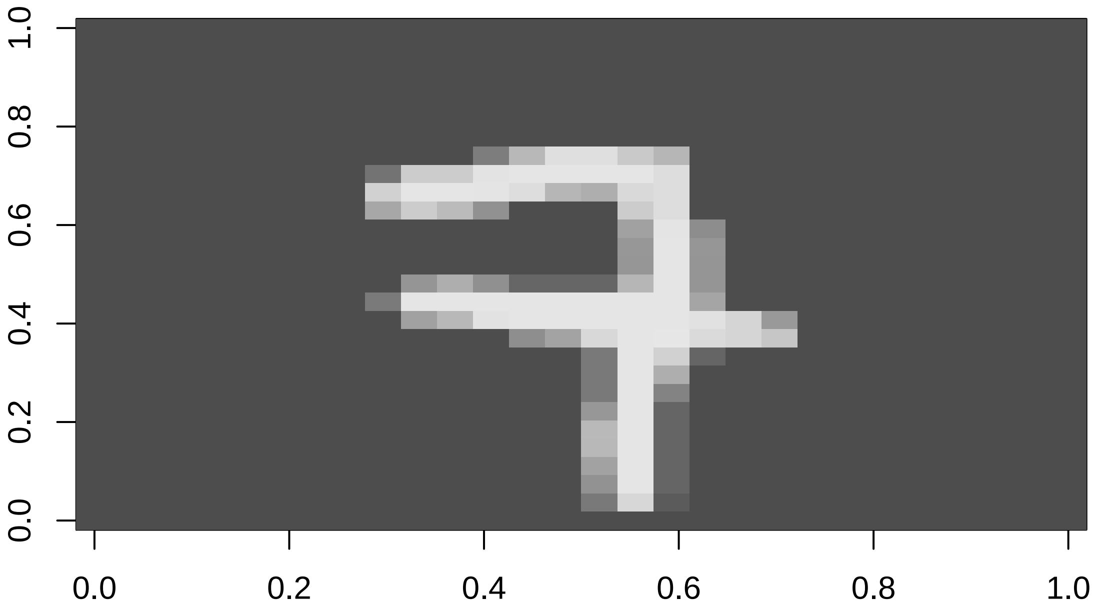
```
errors.ridge.all <- augment(digit.final.ridge.fit, test.tbl)%>%
  select(785:788, 1:784)%>%
  filter(digit != .pred_class)%>%
  select(-(2:4))

#Plot a 3 that was predicted as a 5
errors.ridge.all%>%
  slice(18)%>%
  plot_row()
```




```
#Plot a 4 that was predicted as a 9
errors.ridge.all%>%
  slice(22)%>%
  plot_row()
```

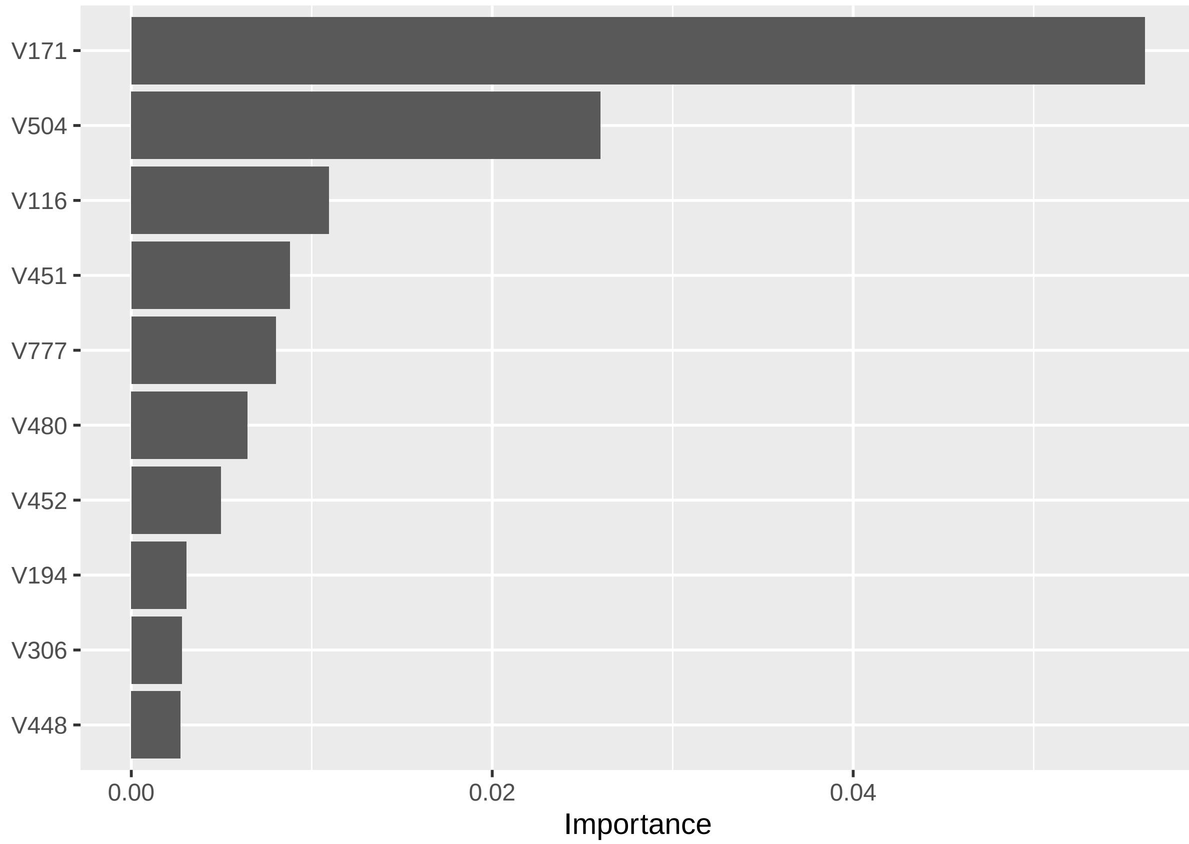



```
#Here's a 7 that got predicted as a 9
errors.ridge.all%>%
  slice(30)%>%
  plot_row()
```



Plotting Most Important Pixels

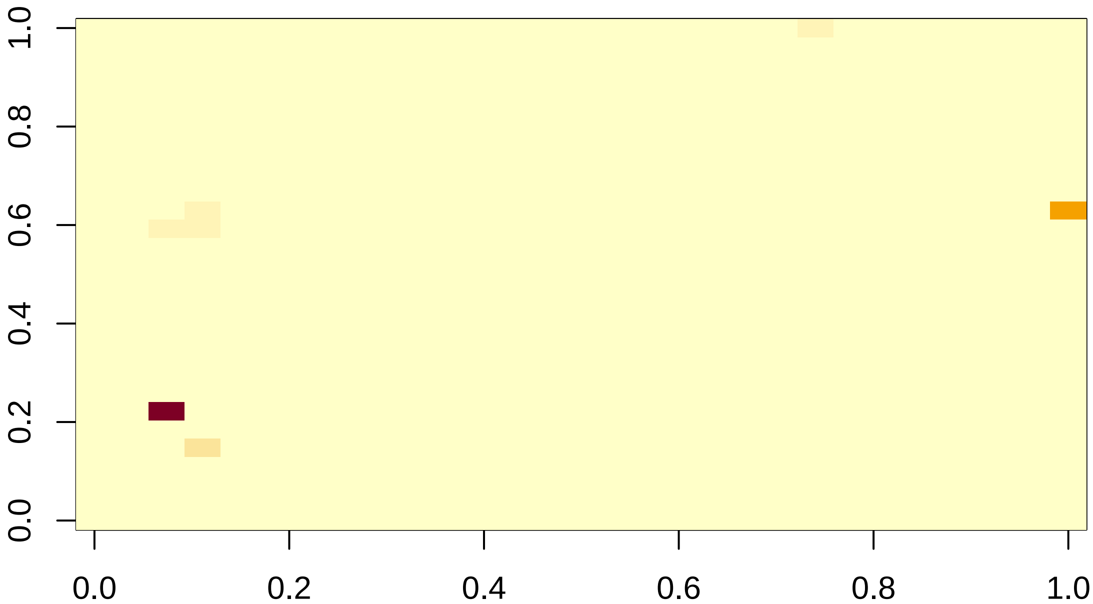
```
digit.final.ridge.fit %>%
  extract_fit_engine() %>%
  vip::vip()
```




```
imp.tbl <- digit.final.ridge.fit %>%
  extract_fit_engine() %>%
  vip::vi()
imp.tbl
```

```
## # A tibble: 784 x 3
##    Variable Importance Sign
##    <chr>         <dbl> <chr>
##  1 V171        0.0562  NEG
##  2 V504        0.0260  NEG
##  3 V116        0.0110  NEG
##  4 V451        0.00878 NEG
##  5 V777        0.00800 NEG
##  6 V480        0.00644 POS
##  7 V452        0.00496 POS
##  8 V194        0.00305 NEG
##  9 V306        0.00279 NEG
## 10 V448        0.00273 NEG
## # ... with 774 more rows
```

```
imp.tbl <- imp.tbl %>%
  mutate(col=as.double(str_remove(Variable,"V")))

mat <- rep(0, 28*28)
mat[imp.tbl$col] <- imp.tbl$Importance
image(matrix(mat, 28, 28))
```

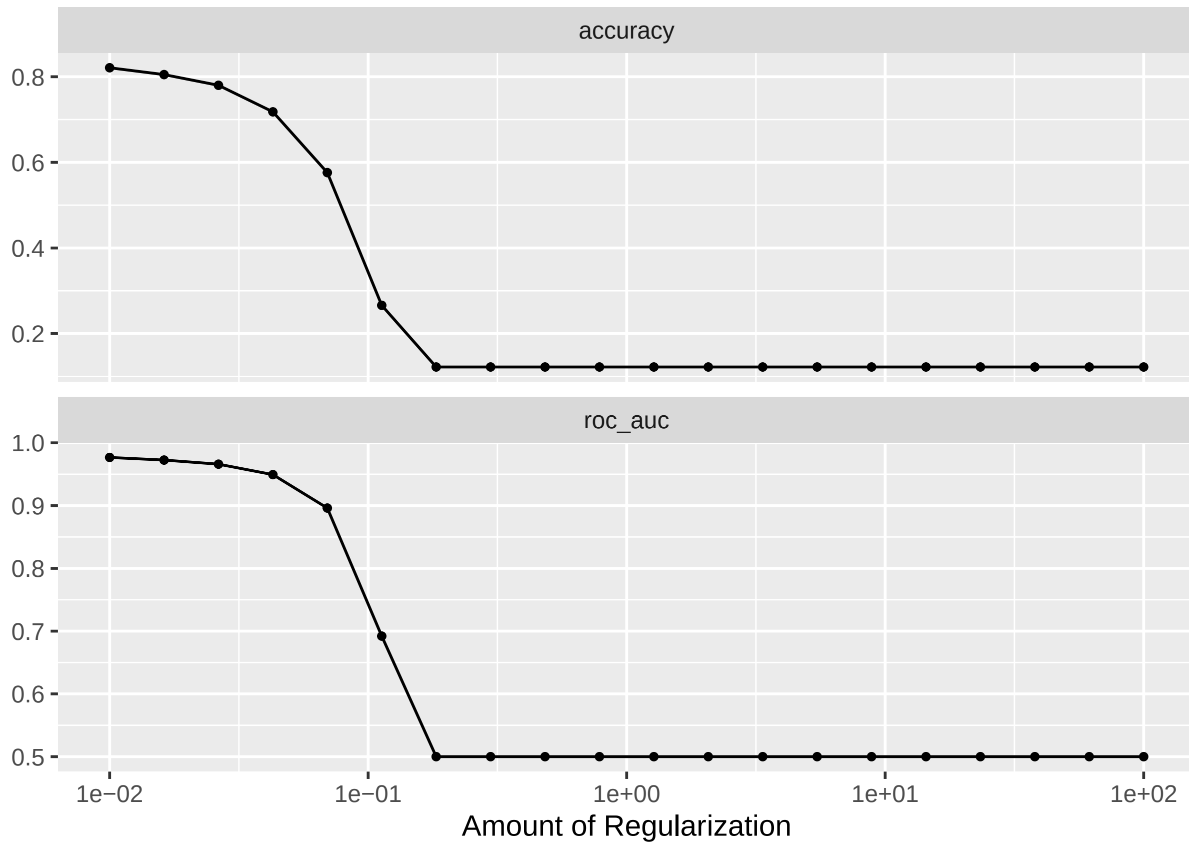


Here, we see pixels on the outer edge of the image are most important, which is an interesting note that I don't really have a good explanation for.

10. Same as exercises 8 and 9, but this time use a LASSO model. Compare and contrast the accuracy of the 3 approaches and the images corresponding to the most important features for the 3 approaches.

```
set.seed(31416)
digit.folds <- vfold_cv(train.tbl, v = 10)

digit.lasso.model <-
  multinom_reg(mixture = 1, penalty = tune())%>%
  set_mode("classification")%>%
  set_engine("glmnet")

#I didn't normalize because all predictors should be on same scale
#I also didn't have the step_zv because I didn't get an error
  digit.lasso.recipe <- recipe(digit ~ ., data=train.tbl)


  digit.lasso.wflow <- workflow() %>%
    add_recipe(digit.lasso.recipe) %>%
    add_model(digit.lasso.model)

  penalty.grid <-
    grid_regular(penalty(range = c(-2, 2)), levels = 20)
```

```r
digit.lasso.res <-
  tune_grid(
    digit.lasso.wflow,
    resamples = digit.folds,
    grid = penalty.grid)

autoplot(digit.lasso.res)
```




```r
best.penalty <- select_by_one_std_err(digit.lasso.res, desc(penalty), metric = "accuracy")
best.penalty
```

```
## # A tibble: 1 x 9
##   penalty .metric  .estimator  mean     n std_err .config            .best .bound
##     <dbl> <chr>    <chr>      <dbl> <int>   <dbl> <fct>              <dbl>  <dbl>
## 1    0.01 accuracy multiclass 0.821    10  0.0122 Preprocessor1_Mo~ 0.821  0.809
```

```r
digit.final.lasso.wf <- finalize_workflow(digit.lasso.wflow, best.penalty)
digit.final.lasso.fit <- fit(digit.final.lasso.wf, data = train.tbl)

augment(digit.final.lasso.fit, test.tbl)%>%
  accuracy(digit, .pred_class)
```

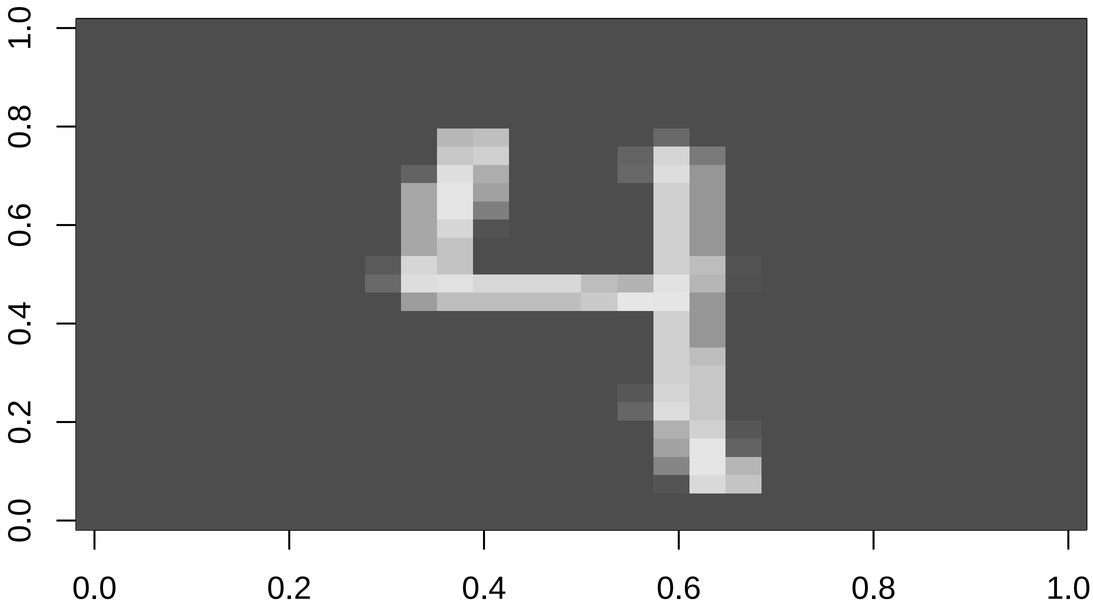
```
## # A tibble: 1 x 3
##   .metric  .estimator .estimate
##   <chr>    <chr>          <dbl>
## 1 accuracy multiclass     0.846
```

```r
augment(digit.final.lasso.fit, test.tbl)%>%
  conf_mat(digit, .pred_class)
```

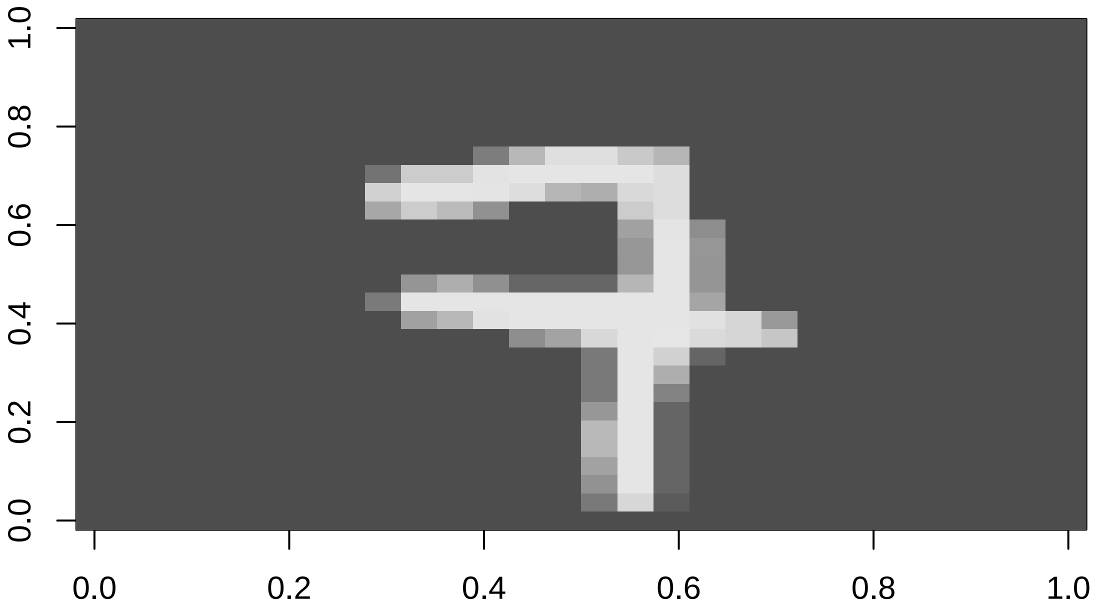
```
##           Truth
## Prediction   0   1   2   3   4   5   6   7   8   9
##          0  86   0   0   0   0   1   1   1   2   2
##          1   0 122   3   0   2   0   0   1   6   1
##          2   1   1  81   2   0   2   1   3   1   0
##          3   0   0   4  77   0   4   0   4   5   1
##          4   0   0   4   0  87   3   3   0   2  11
##          5   1   1   0   7   3  71   3   1   8   2
##          6   1   0   0   3   1   2  89   0   1   0
##          7   0   0   4   2   1   1   1  84   0   5
##          8   0   0   1   3   0   2   2   0  61   3
##          9   0   0   1   0   8   0   0   8   6  88
```

When we run a lasso model, we get an accuracy of 84.6%, which is pretty impressive especially in comparison to the tree model (around an 18% improvement). In comparison to the ridge model, it is slightly worse (around 4% points less accurate). We see that 4s are still mixed up with 9s and some 7s are predicted as 9s. We also see some 9s predicted as 4s.

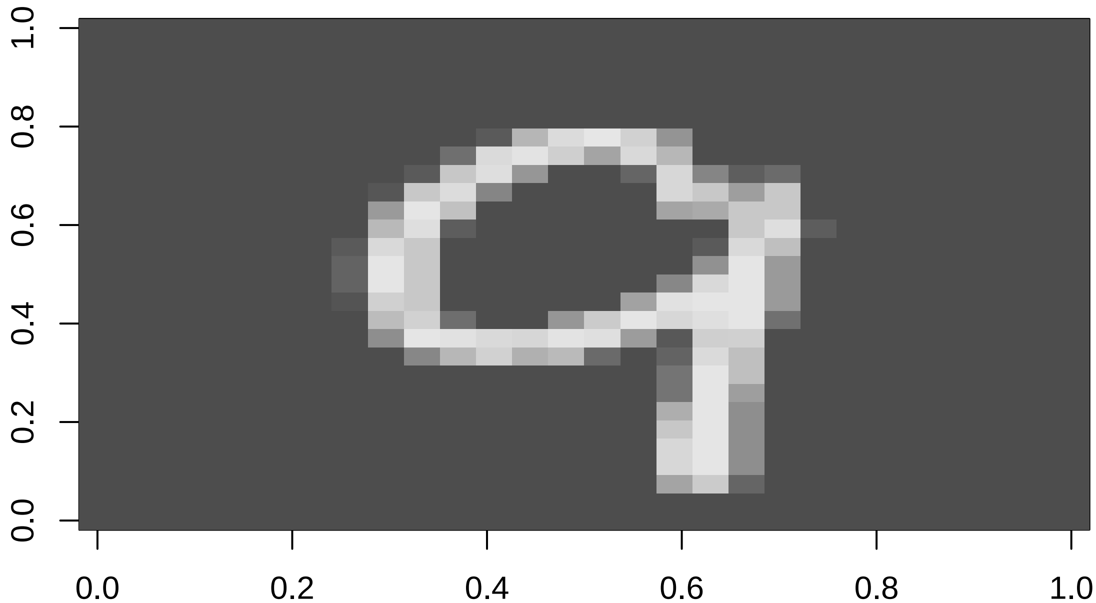
```r
errors.lasso.all <- augment(digit.final.lasso.fit, test.tbl)%>%
  select(785:788, 1:784)%>%
  filter(digit != .pred_class)%>%
  select(-(2:4))

#Plot a 4 that was predicted as a 9
errors.lasso.all%>%
  slice(142)%>%
  plot_row()
```




```
#Plot a 7 that was predicted as a 0
errors.lasso.all%>%
  slice(34)%>%
  plot_row()
```

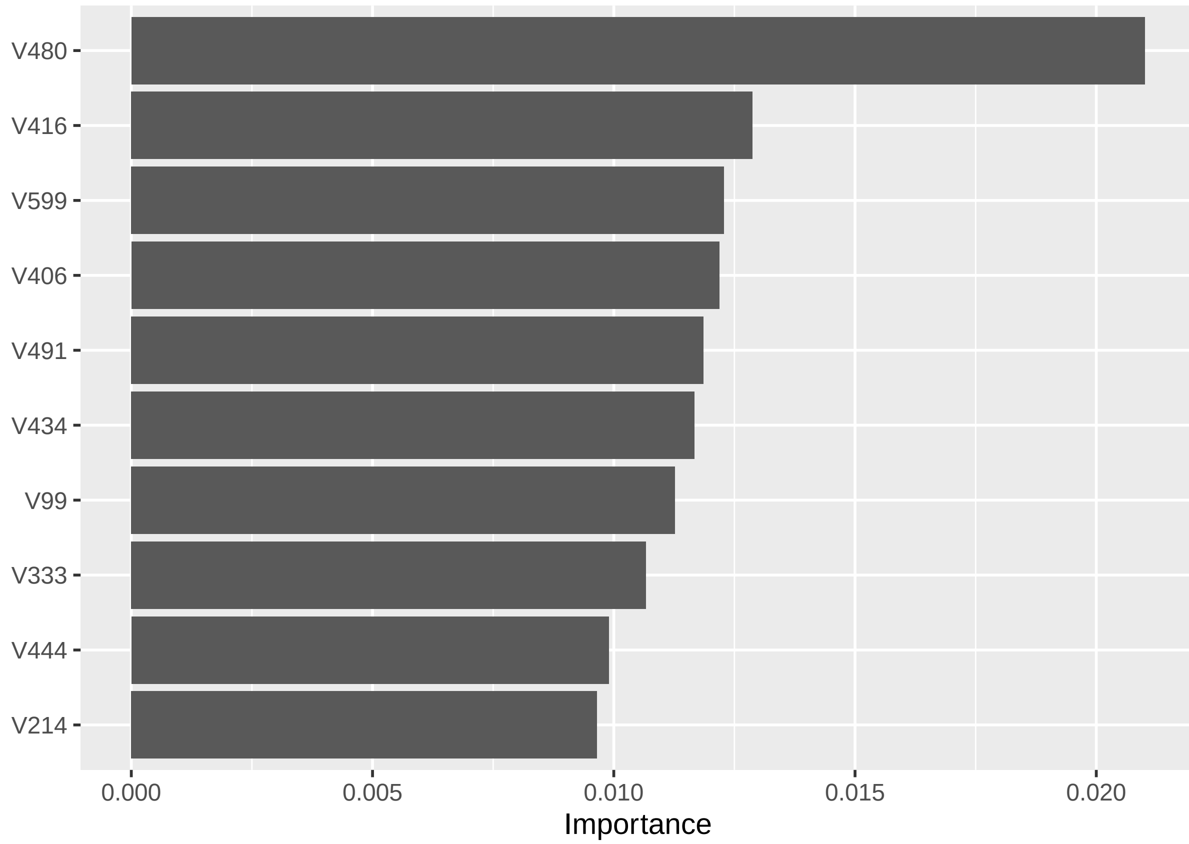



```
#Here's a 9 that got predicted as a 4
errors.lasso.all%>%
  slice(8)%>%
  plot_row()
```



Plotting Most Important Pixels

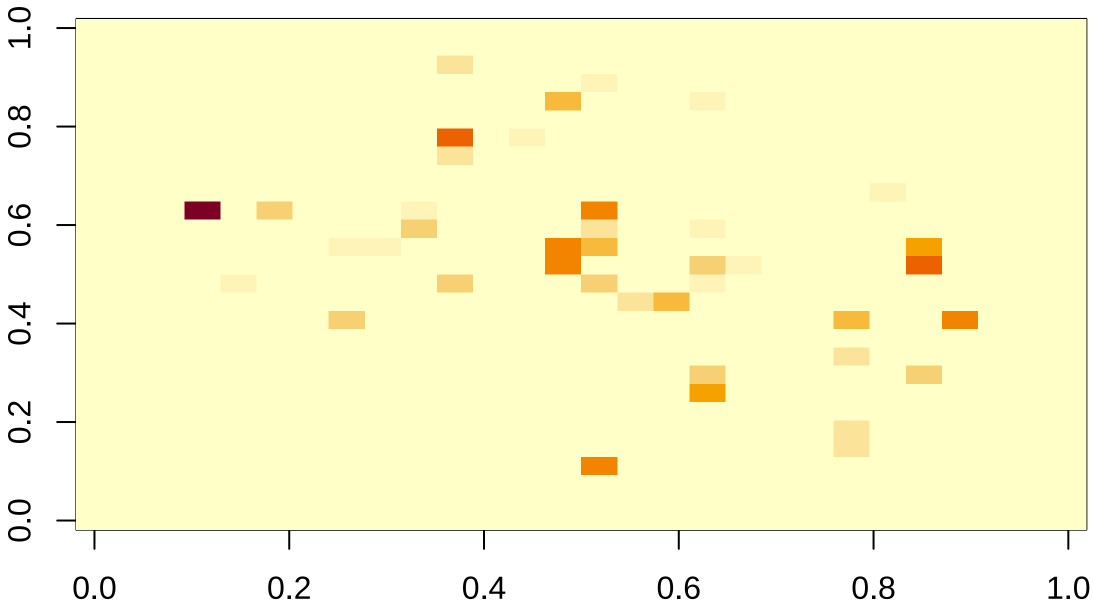
```
digit.final.lasso.fit %>%
  extract_fit_engine() %>%
  vip::vip()
```




```
imp.tbl <- digit.final.lasso.fit %>%
  extract_fit_engine() %>%
  vip::vi()
imp.tbl
```

```
## # A tibble: 784 x 3
##    Variable Importance Sign
##    <chr>         <dbl> <chr>
##  1 V480        0.0210  POS
##  2 V416        0.0129  POS
##  3 V599        0.0123  POS
##  4 V406        0.0122  NEG
##  5 V491        0.0119  NEG
##  6 V434        0.0117  NEG
##  7 V99         0.0113  POS
##  8 V333        0.0107  POS
##  9 V444        0.00990 POS
## 10 V214        0.00965 POS
## # ... with 774 more rows
```

```
imp.tbl <- imp.tbl %>%
  mutate(col=as.double(str_remove(Variable,"V")))

mat <- rep(0, 28*28)
mat[imp.tbl$col] <- imp.tbl$Importance
image(matrix(mat, 28, 28))
```



In the end, we see that ridge proved the most effective in terms of accuracy, followed by lasso. Decision trees had the lowest accuracy of the 3 methods. When we look at the 3 methods, ridge has most important pixels spread out around the edges of the frame, while the decision tree and lasso models had the middle of the frame as the most important pixels (although lasso was a little more spread out). Overall, the most important features for ridge were pretty stark, while the decision tree and lasso were pretty similar - the pixels around the edges were most important for ridge while the middle ones were more important for lasso and decision tree (but especially the decision tree).