Machine Learning with the `MicrosoftML` Package

Ali Zaidi

2017-09-14

Contents

Chapter 1

Prerequisites

This workshop covers the fundamentals of statistical machine learning with the MicrosoftML package.

MicrosoftML is a package that works in tandem with the RevoScaleR package and Microsoft R Server. In order to use the `MicrosoftML` and `RevoScaleR` libraries, you need an installation of Microsoft R Server or Microsoft R Client. You can download **Microsoft R Server** through MSDN here:

1. R Server for Linux
2. R Server for Hadoop
3. R Server for Windows
4. R Server for SQL Server (In-Database)

You can download **Microsoft R Client** through the following sources:

1. R Client for Windows
2. R Client for Linux
3. R Client Docker Image

1.1 MicrosoftML References

- Cheat Sheet
- Overview of MicrosoftML Functions
- Function Reference
- Quickstarts
- MSDN Site for Microsoft R Server
- MSDN Site for Microsoft R Client
  - R Client Overview

1.2 Useful Resources

- Introduction to Statistical Learning Video Lectures
  - My favorite course on statistical learning
  - Worth doing twice!
- Introduction to Statistical Learning Textbook
  - *Baby Statistical Learning*
- Elements of Statistical Learning Textbook
  - **Papa Statistical Learning**

- R for Data Science
  - Covers the core tools in the `tidyverse` ecosystem for data science and analytics
- Microsoft R for Data Science
  - My two-day workshop on Microsoft R Server for Data Science
- Microsoft R Server and Spark
  - My 1.5 day course on Microsoft R Server and Spark Integration
- Scalable Data Science with Microsoft R Server and Spark
  - In progress book on data science with Microsoft R Server and Spark

Chapter 2

Exploratory Data Analysis and Feature Engineering

Import your favorite libraries and set up your favorite plotting theme.

```
library(tidyverse)
theme_set(theme_minimal())
```

Now let's import some data. Our data source is 1990-census level of housing in California.

We'll use easy the `readr` package to load in our data. You could equivalently use `read.csv`, or `data.table::fread` if you wanted incredible speed. We'll later see how to use data sources using the `RevoScaleR` readers.

Since we're using `readr`, our data is a `tbl`. This means we will get some `dplyr` and `tibble` features, and it'll behave a little differently than a traditional `data.frame`.

```
housing <- read_csv("data/housing.csv")
class(housing)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
glimpse(housing)
```

```
## Observations: 20,640
## Variables: 10
## $ longitude          <dbl> -122.23, -122.22, -122.24, -122.25, -122.25...
## $ latitude           <dbl> 37.88, 37.86, 37.85, 37.85, 37.85, 37.85, 3...
## $ housing_median_age <dbl> 41, 21, 52, 52, 52, 52, 52, 52, 42, 52, 52,...
## $ total_rooms        <dbl> 880, 7099, 1467, 1274, 1627, 919, 2535, 310...
## $ total_bedrooms     <dbl> 129, 1106, 190, 235, 280, 213, 489, 687, 66...
## $ population         <dbl> 322, 2401, 496, 558, 565, 413, 1094, 1157, ...
## $ households         <dbl> 126, 1138, 177, 219, 259, 193, 514, 647, 59...
## $ median_income      <dbl> 8.3252, 8.3014, 7.2574, 5.6431, 3.8462, 4.0...
## $ median_house_value <dbl> 452600, 358500, 352100, 341300, 342200, 269...
## $ ocean_proximity    <chr> "NEAR BAY", "NEAR BAY", "NEAR BAY", "NEAR B...
```

```
housing
```

```
## # A tibble: 20,640 x 10
##    longitude latitude housing_median_age total_rooms total_bedrooms
##        <dbl>    <dbl>              <dbl>       <dbl>          <dbl>
```

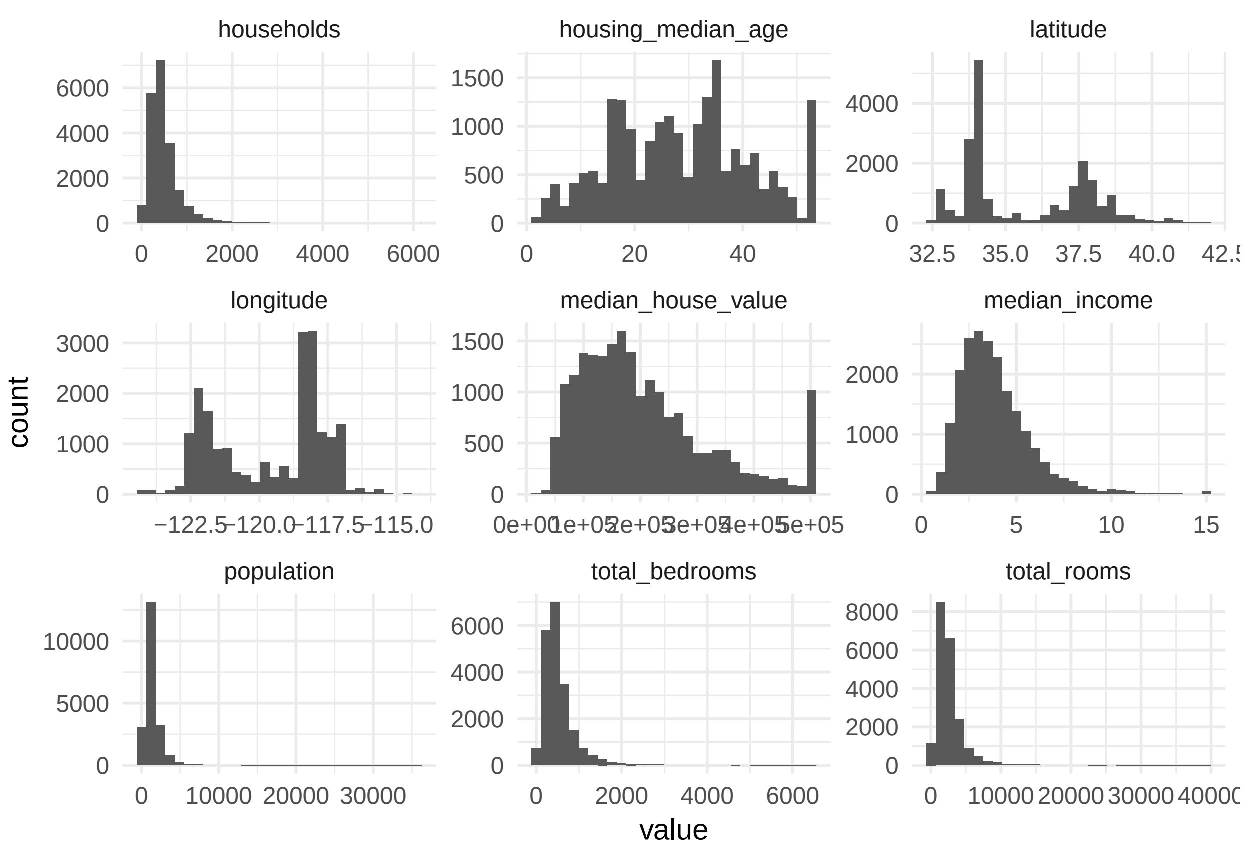
```
## 1    -122.23    37.88                     41        880            129
## 2    -122.22    37.86                     21       7099           1106
## 3    -122.24    37.85                     52       1467            190
## 4    -122.25    37.85                     52       1274            235
## 5    -122.25    37.85                     52       1627            280
## 6    -122.25    37.85                     52        919            213
## 7    -122.25    37.84                     52       2535            489
## 8    -122.25    37.84                     52       3104            687
## 9    -122.26    37.84                     42       2555            665
## 10   -122.25    37.84                     52       3549            707
## # ... with 20,630 more rows, and 5 more variables: population <dbl>,
## #   households <dbl>, median_income <dbl>, median_house_value <dbl>,
## #   ocean_proximity <chr>
```

2.1 Visualize Densities

Suppose we want to visualize the distribution of the numeric columns, such as **housing_median_age**. How would you visualize that density?

```
housing %>% keep(is_double) %>%
  gather %>%
  ggplot(aes(x = value)) + geom_histogram() + facet_wrap(~key, scales = "free")
```

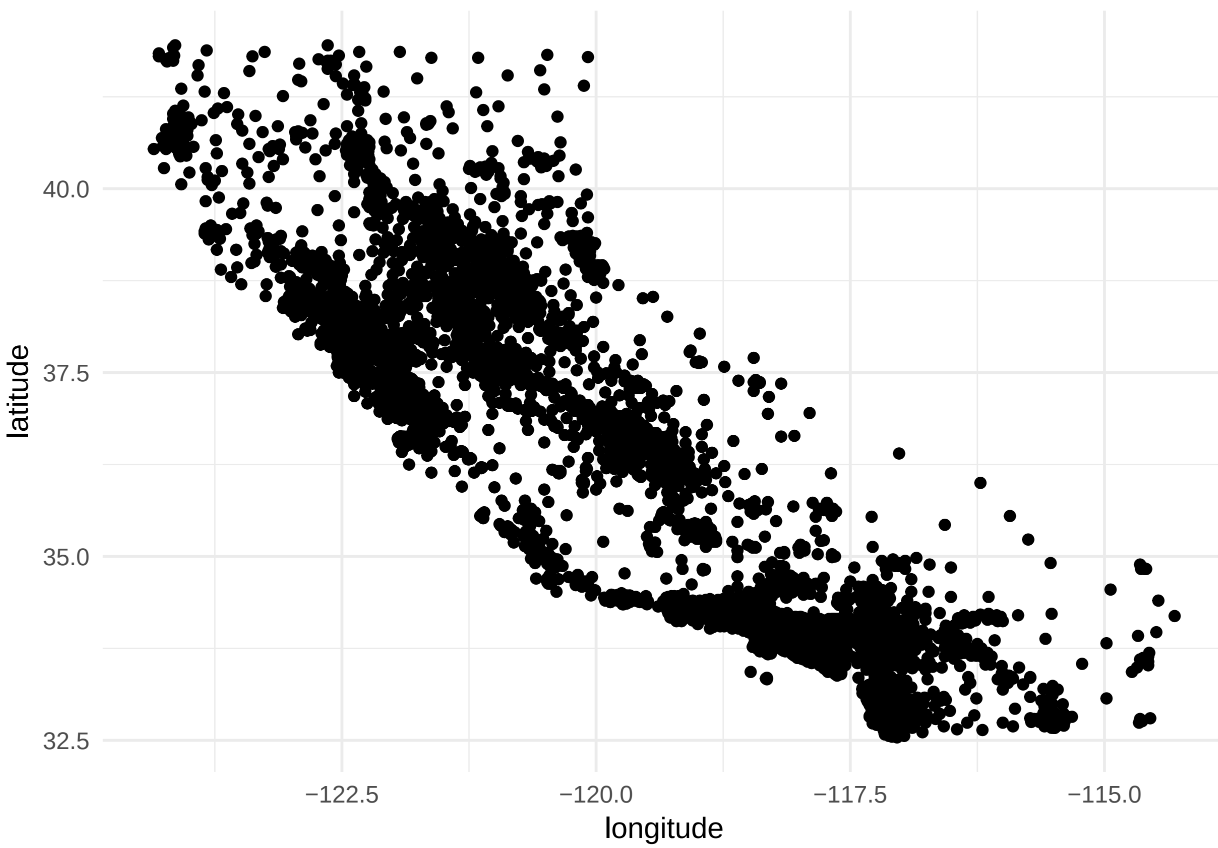
```
## Warning: Removed 207 rows containing non-finite values (stat_bin).
```



2.2 Spatial Visualizations

The histograms of the longitude and latitude columns seem like they are in some reasonable range of data. Let's visualize the locations.

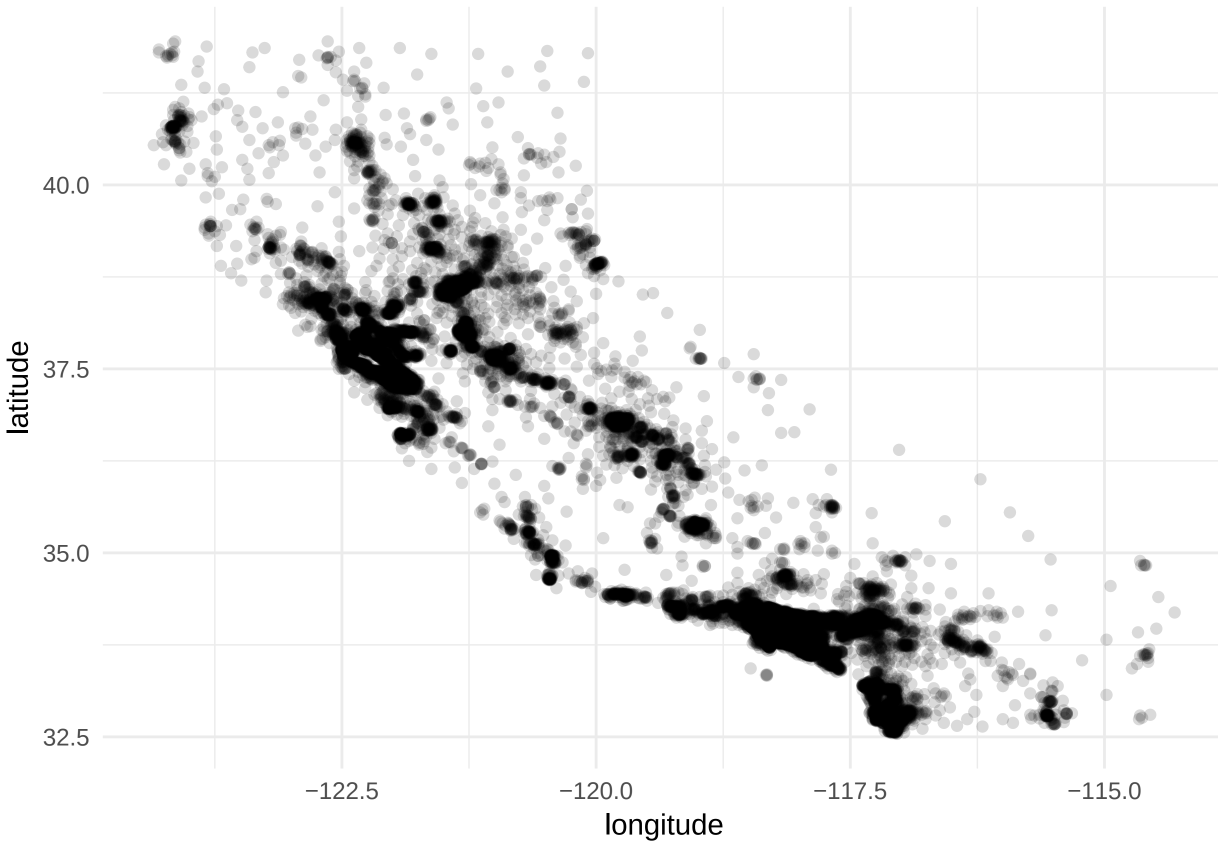
```
housing %>% ggplot(aes(x = longitude, y = latitude)) +
  geom_point()
```



Looks indeed like California!

We can improve this chart in many ways. Let's first add some transparency so we can get a better sense of the *density* of points:

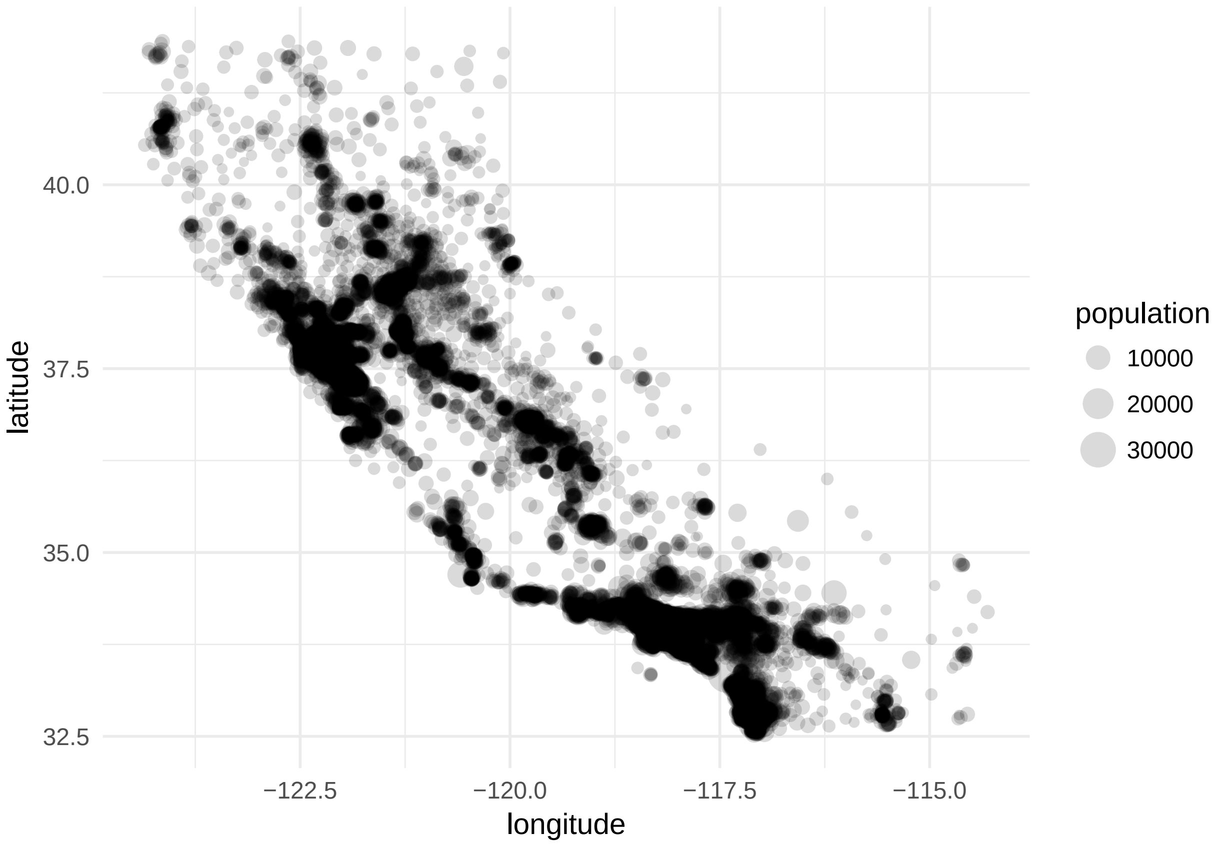
```
housing %>% ggplot(aes(x = longitude, y = latitude)) +
  geom_point(alpha = 0.15)
```



Nice, we can already see some dense clusters for higher population regions like the Bay Area, Sacramento and Los Angeles.

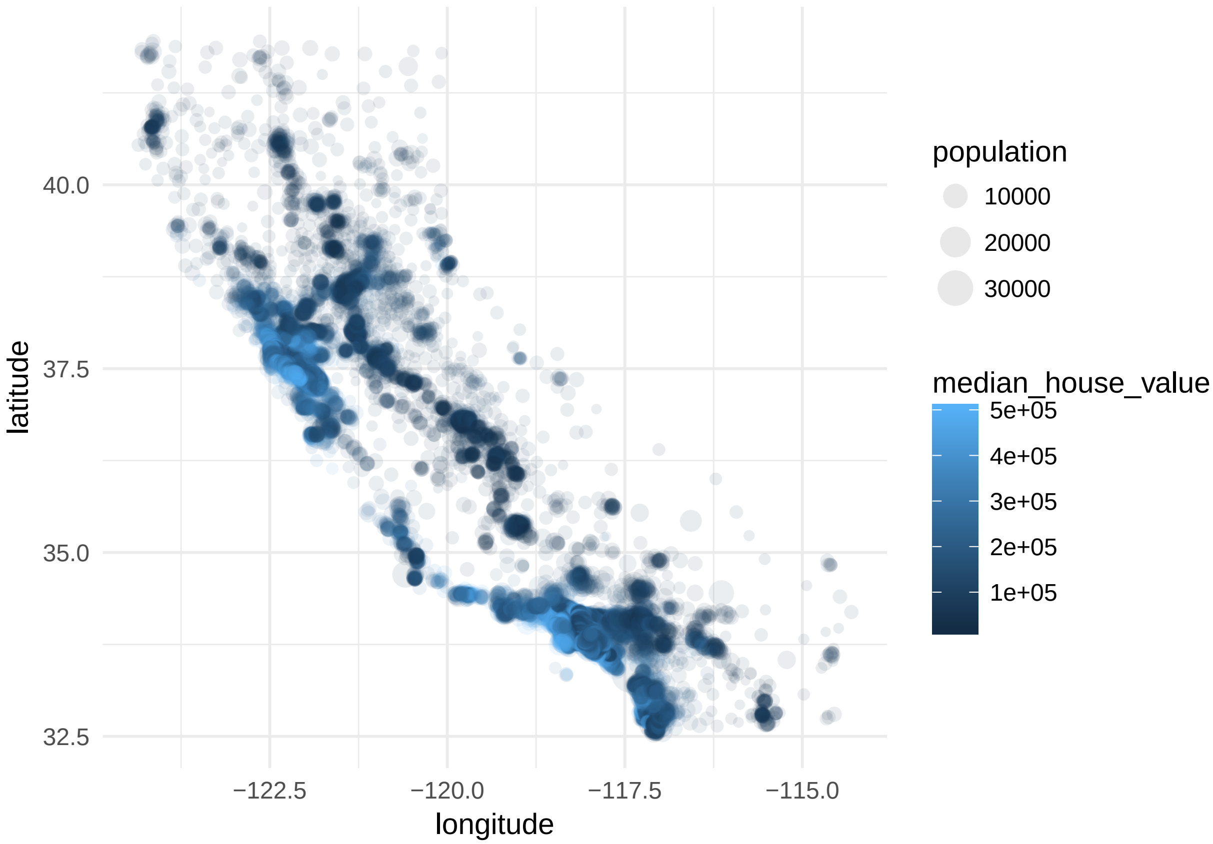
Let's add some additional attributes, like the population:

```
housing %>% ggplot(aes(x = longitude, y = latitude,
                       size = population)) +
  geom_point(alpha = 0.15)
```



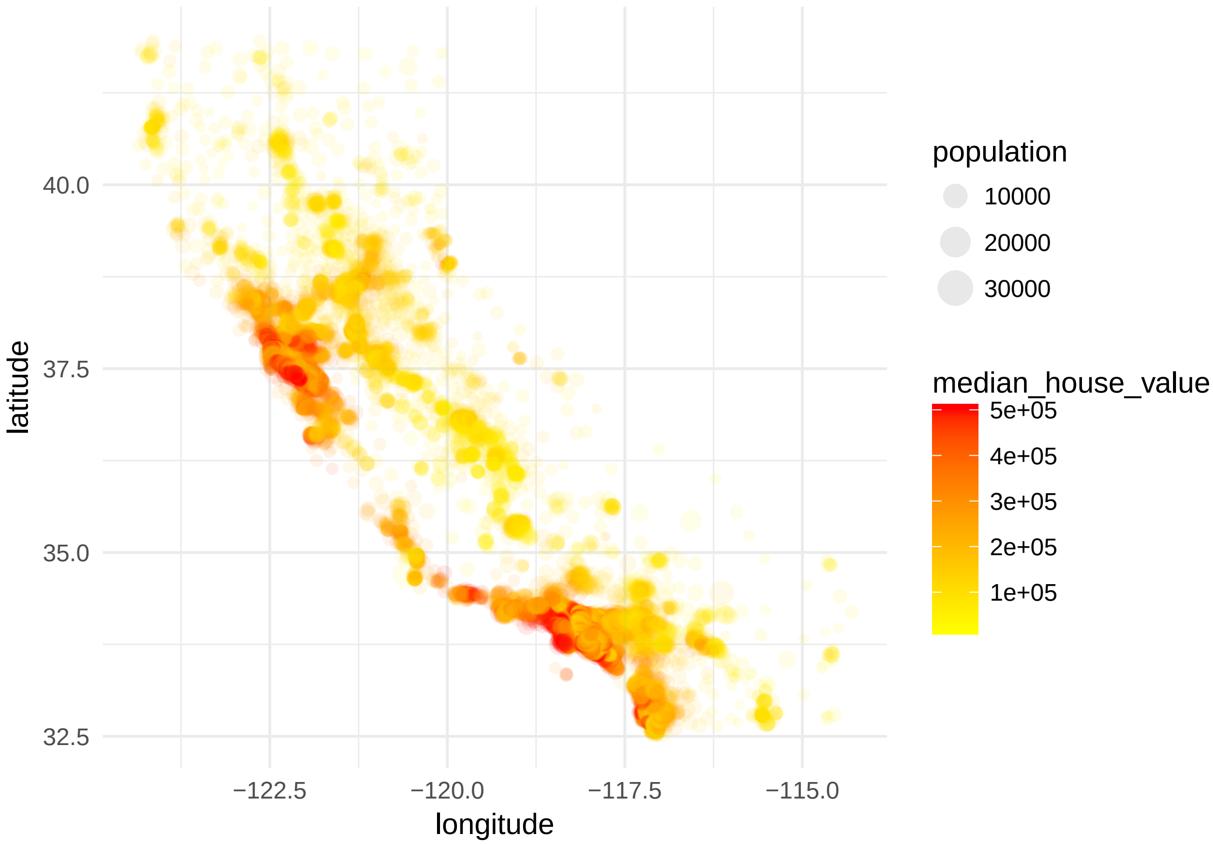
And now let's see if we can add a gradient fill for another numeric value, like the median housing price:

```
housing %>% ggplot(aes(x = longitude, y = latitude,
                       size = population,
                       colour = median_house_value)) +
  geom_point(alpha = 0.095)
```



Interesting, we can definitely see the higher price ranges in LA and the Bay Area. Let's see if we can change the colour scheme to get an even better visualization:

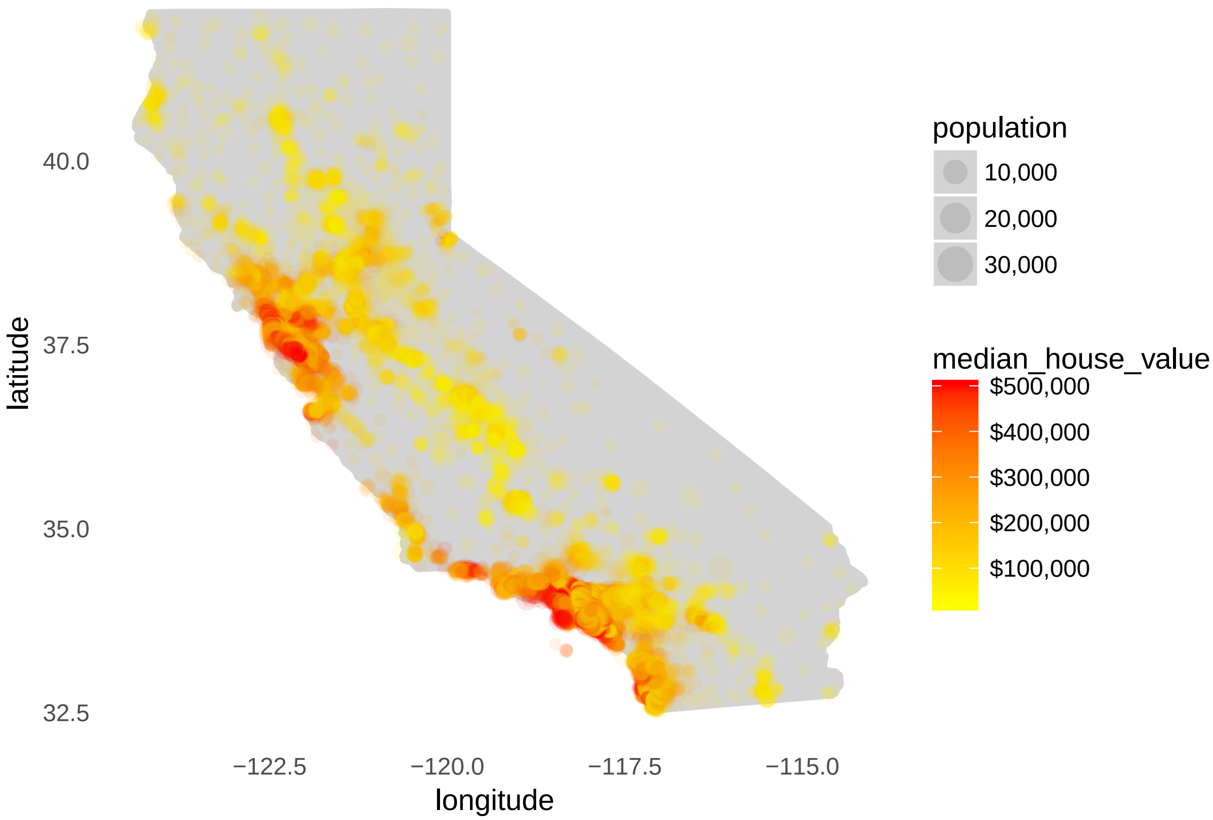
```
housing %>% ggplot(aes(x = longitude, y = latitude,
                       size = population,
                       colour = median_house_value)) +
  geom_point(alpha = 0.095) +
  scale_colour_gradient(low = "yellow", high = "red")
```



If you want to plot the points on top of an actual polygon of the state map, you can do that using `geom_map` and using the `map_data` function in ggplot2. I find it a bit messy, but it might be worthwile if you are plotting various regions/states.

While we're at it, let's put some final themes on our plot to make it more aesthetically pleasing.

```
housing %>%
  mutate(state = "california") %>%
  ggplot(aes(x = longitude, y = latitude,
             size = population,
             colour = median_house_value)) +
  geom_map(map = filter(map_data("state"), region == "california"),
           aes(map_id = state),
           fill = "lightgrey",
           colour = "lightgrey") +
  geom_point(alpha = 0.1) +
  theme(panel.grid.major = element_blank(),
        panel.grid.minor = element_blank()) +
  scale_size_continuous(labels = scales::comma) +
  scale_colour_gradient(low = "yellow", high = "red",
                        labels = scales::dollar)
```



2.3 Exercises

1. How does proximity to the ocean effect median house values? Try to visualize housing prices the above using the ocean proximity variable as the colour/fill aesthetic and median_house_value as the size aesthetic.
2. We saw that faceting could allow us to compare distributions rather easily. Try a facetted plot where you facet by quantiles of the `median_income` column. This will allow you to compare the distribution of housing values, populations/demographics across different income groups. The `dplyr::mutate`, `stats::quntile`, and `ggplot2::facet_wrap` functions should be useful here.

Chapter 3

Regression Models

3.1 Splitting into Train and Test Sets

Let's sample our data into train and test sets. In order to do this efficiently, we will use the `RevoScaleR` package.

We'll first create a `RxXdfData` object, which is a more efficient and scalable data structure than R `data.frames`. Their primary distinction is that they do not reside in memory, but on-disk.

```
library(tidyverse)
library(dplyrXdf)
library(foreach)
library(doRSR)
library(MicrosoftML)
theme_set(theme_minimal())

out_xdf <- file.path(
                     "data",
                     "housing.xdf")

housing_xdf <- rxDataStep(inData = housing,
                          outFile = out_xdf,
                          maxRowsByCols = nrow(housing)*ncol(housing),
                          rowsPerRead = 5000,
                          overwrite = TRUE)

housing_xdf %<>% factorise(ocean_proximity) %>%
  persist(out_xdf, overwrite = TRUE)
```

The `RevoScaleR` and `MicrosoftML` functions are primarily prefixed with `rx`. In this function below, we will use the `rxSplit` function to split our data into train and test sets. Observe that since our data is now on-disk, and compromises of multiple blocks, we have to use the `.rxNumRows` argument to inform the session how many rows are currently being processed in the current block:

```
split_xdf <- function(data) {

    splits <- rxSplit(data,
                      outFileSuffixes = c("Train", "Test", "Validate"),
                         splitByFactor = "splitVar",
```

```
                             overwrite = TRUE,
                             transforms = list(splitVar = factor(
                               sample(c("Train", "Test", "Validate"),
                                      size = .rxNumRows,
                                      replace = TRUE,
                                      prob = c(0.65, 0.25, 0.1)),
                               levels = c("Train", "Test", "Validate"))),
                             rngSeed = 123,
                             consoleOutput = TRUE)
  return(splits)
}


splits <- split_xdf(housing_xdf)
names(splits) <- c("train", "test", "validate")
```

Now that we have our train and test sets, we can conduct begin to train our models.

3.2 Training Regression Learners

Let's train our first regression model.

We can start with the a `glm` model. GLMs, short for generalized linear models, are a general class of linear algorithms. In this exercise, our goal is to predict the median housing value given the other variables.

```
lin_mod <- rxLinMod(median_house_value ~ housing_median_age + total_rooms + total_bedrooms +
                      population + households + median_income + ocean_proximity,
                    data = splits$train)
```

That was pretty easy, but let's generalize our approach so that we can estimate a variety of models quickly and efficiently.

First, we'll create a wrapper function to automatically create our model matrix for us dynamically from our data.

```
make_form <- function(xdf = housing_xdf,
                      resp_var = "median_house_value",
                      vars_to_skip = c("splitVar", "longitude",
                                       "latitude")) {

  library(stringr)

  non_incl <- paste(vars_to_skip, collapse = "|")

  x_names <- names(xdf)

  features <- x_names[!str_detect(x_names, resp_var)]
  features <- features[!str_detect(features, non_incl)]

  form <- as.formula(paste(resp_var, paste0(features, collapse = " + "),
                           sep  = " ~ "))

  return(form)
}
```

```
make_form(xdf = splits$train)
```

```
## median_house_value ~ housing_median_age + total_rooms + total_bedrooms +
##     population + households + median_income + ocean_proximity
## <environment: 0xbbeb960>
```

Now let's create a modeling wrapper, which will take our dataset, a formula, and a model, and train it for us.

```
estimate_model <- function(xdf_data = splits$train,
                           form = make_form(xdf = xdf_data),
                           model = rxLogit, ...) {

  rx_model <- model(form, data = xdf_data, ...)

  return(rx_model)


}
```

Now we can quickly iterate over our data and train models using different learning algorithms. For example, the above example suffers from the issue that we didn't scale our data prior to learning. This can have an adverse effect on the optimization function of the learning algorithm, as it'll favor the variables with more disperse scales.

We'll use the SDCA - Stochastic Dual Coordinate Ascent learning algorithm, which automatically applies a min-max scaling to our data prior to training.

```
sdca <- estimate_model(model = rxFastLinear, type = "regression")
```

```
## Automatically adding a MinMax normalization transform, use 'norm=Warn' or 'norm=No' to turn this behavior
## Using 2 threads to train.
## Automatically choosing a check frequency of 2.
## Warning: Skipped 128 instances with missing features/label during training
## Auto-tuning parameters: maxIterations = 110.
## Auto-tuning parameters: L2 = 0.0001.
## Auto-tuning parameters: L1Threshold (L1/L2) = 0.
## Using best model from iteration 78.
## Not training a calibrator because it is not needed.
## Elapsed time: 00:00:00.9279494
## Elapsed time: 00:00:00.0009998
```

```
summary(sdca)
```

```
## Call:
## model(formula = form, data = xdf_data, type = "regression")
##
## SDCAR (RegressorTrainer) for: median_house_value~housing_median_age+total_rooms+total_bedrooms+populat
## Data: xdf_data (RxXdfData Data Source)
## File name: /home/alizaidi/bookdown-demo/housing.splitVar.Train.xdf
##
## First 12 of 12 Non-zero Coefficients:
## (Bias): -32979.58
## population: -958475.9
## median_income: 562343.2
## total_bedrooms: 376008.2
## households: 238733.6
```

```
## ocean_proximity.ISLAND: 200716.1
## ocean_proximity.NEAR OCEAN: 86880.88
## ocean_proximity.NEAR BAY: 79374.24
## total_rooms: -74753.57
## ocean_proximity.<1H OCEAN: 73038.17
## housing_median_age: 47560.33
## ocean_proximity.INLAND: 4924.849
```

3.3 Scoring Our Data on the Test Set

Now that we our model trained, we can score it on our test set.

Let's create a prediction XDF where we'll save our results to.

```
pred_xdf <- file.path("/home", system("whoami", intern = TRUE), "scored.xdf")
if (file.exists(pred_xdf)) file.remove(pred_xdf)
```

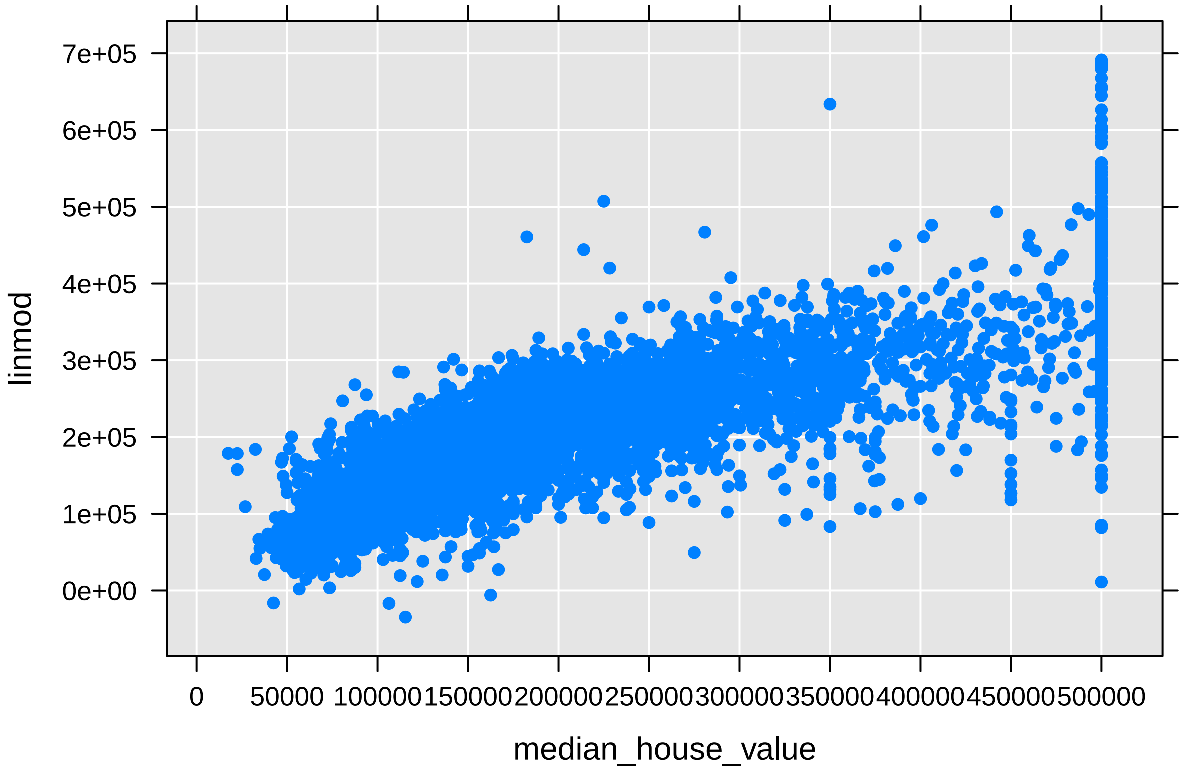
```
## [1] TRUE
```

```
scored_xdf <- RxXdfData(pred_xdf)
```

```
rxPredict(lin_mod, data = splits$test,
          outData = pred_xdf, writeModelVars = T,
          predVarNames = c("linmod"), overwrite = T)
rxGetInfo(pred_xdf)
```

```
## File name: /home/alizaidi/scored.xdf
## Number of observations: 5003
## Number of variables: 9
## Number of blocks: 5
## Compression type: zlib
```

```
rxLinePlot(linmod ~ median_house_value, data = pred_xdf, type = "p")
```

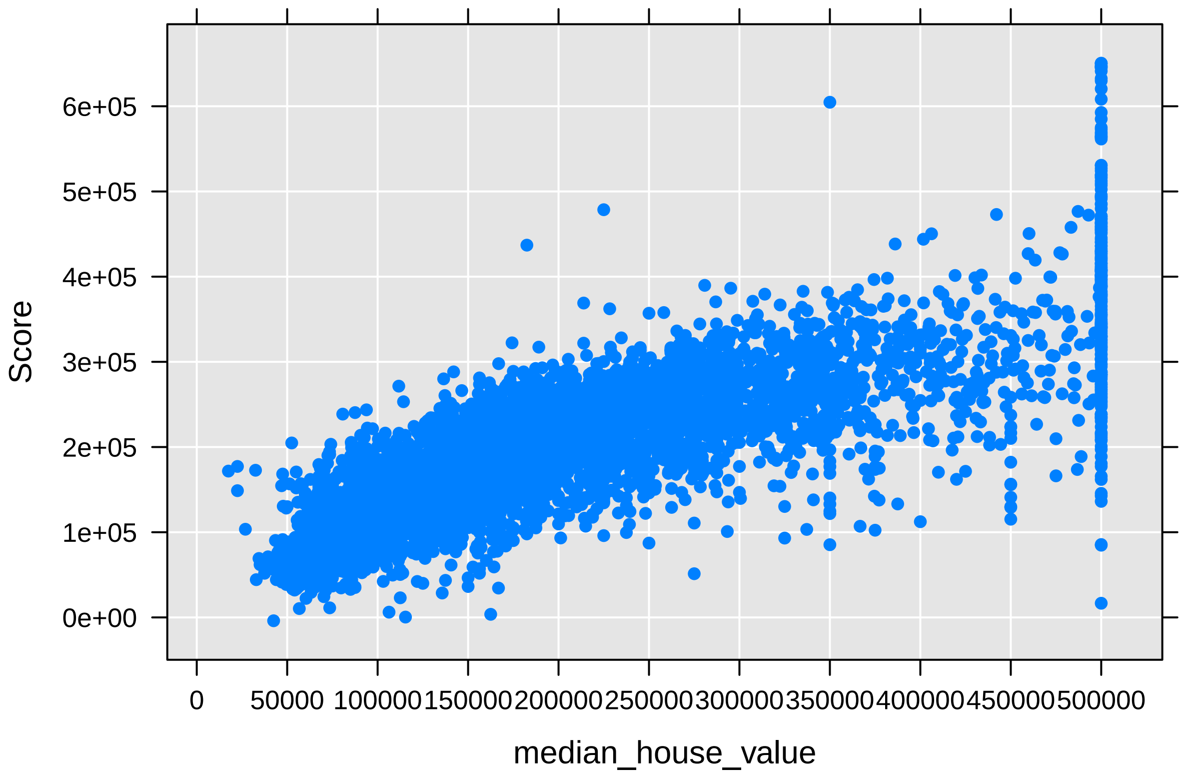


Let's also score our SDCA model:

```r
rxPredict(sdca, data = splits$test,
          outData = pred_xdf, writeModelVars = T)
```

```
## Elapsed time: 00:00:00.0249283
```

```r
# rxGetInfo(pred_xdf, numRows = 2)
rxLinePlot(Score ~ median_house_value, data = pred_xdf, type = "p")
```



3.4 Training Many Models Concurrently

Let's take our functions and train multiple models in parallel. We have already trained two linear models. Let's add two ensemble tree algorithms to the mix, `rxBTrees`, and simultaneously train a random forest using `rxDForest`.

To run them in parallel, we can use the foreach package with a local parallel backend.

```
rxSetComputeContext(RxLocalParallel())
registerDoRSR(computeContext = rxGetComputeContext())

models <- list("btrees" = rxBTrees,
               "forest" = rxDForest)
models <- foreach(i = models) %dopar% estimate_model(model = i)
names(models) <- c("btrees", "forest")
models
```

```
## $btrees
##
## Call:
## model(formula = form, data = xdf_data)
##
##
##       Loss function of boosted trees: bernoulli
##        Number of boosting iterations: 10
## No. of variables tried at each split: 2
##
##             OOB estimate of deviance: NA
##
```

```
## $forest
##
## Call:
## model(formula = form, data = xdf_data)
##
##
##              Type of decision forest: anova
##                      Number of trees: 10
## No. of variables tried at each split: 2
##
##           Mean of squared residuals: 4563008000
##                     % Var explained: 66
```

```
lapply(models, summary)
```

```
## $btrees
##           Length Class      Mode
## ntree      1     -none-     numeric
## mtry       1     -none-     numeric
## type       1     -none-     character
## forest    10     -none-     list
## oob.err    4     data.frame list
## init.pred  1     -none-     numeric
## params    65     -none-     list
## formula    3     formula    call
## call       3     -none-     call
##
## $forest
##         Length Class      Mode
## ntree    1     -none-     numeric
## mtry     1     -none-     numeric
## type     1     -none-     character
## forest  10     -none-     list
## oob.err  4     data.frame list
## params  65     -none-     list
## formula  3     formula    call
## call     3     -none-     call
```

3.5 Exercise

1. Use the `rxDTree` function to sit a single regression tree to this dataset.
2. Visualize the fit of your decision tree using the `RevoTreeView` library and it's `createTreeView` and `plot` functions.

Chapter 4

Classification Models for Computer Vision

4.1 Hand-Written Digit Classifiation

In this module, we will examine the MNIST dataset, which is a set of 70,000 images of digits handwritten by high school students and employees of the US Census Bureau.

MNIST is considered the "hello-world" of the machine-learning world, and is often a good place to start for understanding classification algorithms.

Let's load the MNIST dataset.

```
library(MicrosoftML)
library(tidyverse)
library(magrittr)
library(dplyrXdf)
theme_set(theme_minimal())

mnist_xdf <- file.path("data", "MNIST.xdf")
mnist_xdf <- RxXdfData(mnist_xdf)
```

Let's take a look at the data:

```
rxGetInfo(mnist_xdf)
```

```
## File name: /home/alizaidi/bookdown-demo/data/MNIST.xdf
## Number of observations: 70000
## Number of variables: 786
## Number of blocks: 7
## Compression type: zlib
```

Our dataset contains 70K records, and 786 columns. There are actually 784 features, because each image in the dataset is a 28x28 pixel image. The two additional columns are for the label, and a column with a pre-sampled train and test split.

4.2 Visualizing Digits

Let's make some visualizations to examine the MNIST data and see what we can use for a classifier to classify the digits.

```
mnist_df <- rxDataStep(inData = mnist_xdf, outFile = NULL,
                       maxRowsByCols = nrow(mnist_xdf)*ncol(mnist_xdf)) %>% tbl_df
```

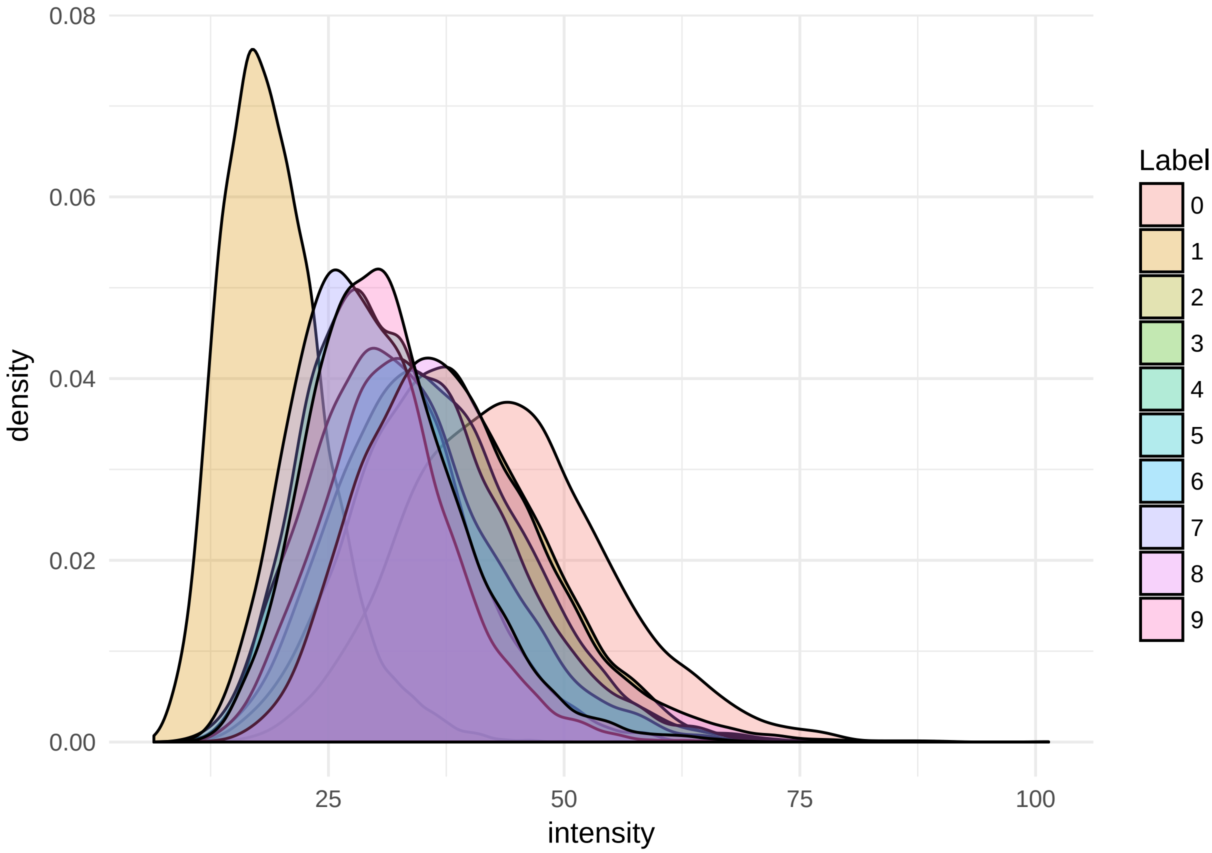
Let's see the average for each digit:

```
mnist_df %>%
  keep(is.numeric) %>%
  rowMeans() %>% data.frame(intensity = .) %>%
  tbl_df %>%
  bind_cols(mnist_df) %T>% print -> mnist_df
```

```
## # A tibble: 70,000 x 787
##    intensity  Label    V2    V3    V4    V5    V6    V7    V8    V9   V10
##        <dbl> <fctr> <int> <int> <int> <int> <int> <int> <int> <int> <int>
##  1  35.10842      5     0     0     0     0     0     0     0     0     0
##  2  39.66199      0     0     0     0     0     0     0     0     0     0
##  3  24.79974      4     0     0     0     0     0     0     0     0     0
##  4  21.85587      1     0     0     0     0     0     0     0     0     0
##  5  29.60969      9     0     0     0     0     0     0     0     0     0
##  6  37.75638      2     0     0     0     0     0     0     0     0     0
##  7  22.50765      1     0     0     0     0     0     0     0     0     0
##  8  45.74872      3     0     0     0     0     0     0     0     0     0
##  9  13.86990      1     0     0     0     0     0     0     0     0     0
## 10  27.93878      4     0     0     0     0     0     0     0     0     0
## # ... with 69,990 more rows, and 776 more variables: V11 <int>, V12 <int>,
## #   V13 <int>, V14 <int>, V15 <int>, V16 <int>, V17 <int>, V18 <int>,
## #   V19 <int>, V20 <int>, V21 <int>, V22 <int>, V23 <int>, V24 <int>,
## #   V25 <int>, V26 <int>, V27 <int>, V28 <int>, V29 <int>, V30 <int>,
## #   V31 <int>, V32 <int>, V33 <int>, V34 <int>, V35 <int>, V36 <int>,
## #   V37 <int>, V38 <int>, V39 <int>, V40 <int>, V41 <int>, V42 <int>,
## #   V43 <int>, V44 <int>, V45 <int>, V46 <int>, V47 <int>, V48 <int>,
## #   V49 <int>, V50 <int>, V51 <int>, V52 <int>, V53 <int>, V54 <int>,
## #   V55 <int>, V56 <int>, V57 <int>, V58 <int>, V59 <int>, V60 <int>,
## #   V61 <int>, V62 <int>, V63 <int>, V64 <int>, V65 <int>, V66 <int>,
## #   V67 <int>, V68 <int>, V69 <int>, V70 <int>, V71 <int>, V72 <int>,
## #   V73 <int>, V74 <int>, V75 <int>, V76 <int>, V77 <int>, V78 <int>,
## #   V79 <int>, V80 <int>, V81 <int>, V82 <int>, V83 <int>, V84 <int>,
## #   V85 <int>, V86 <int>, V87 <int>, V88 <int>, V89 <int>, V90 <int>,
## #   V91 <int>, V92 <int>, V93 <int>, V94 <int>, V95 <int>, V96 <int>,
## #   V97 <int>, V98 <int>, V99 <int>, V100 <int>, V101 <int>, V102 <int>,
## #   V103 <int>, V104 <int>, V105 <int>, V106 <int>, V107 <int>,
## #   V108 <int>, V109 <int>, V110 <int>, ...
```

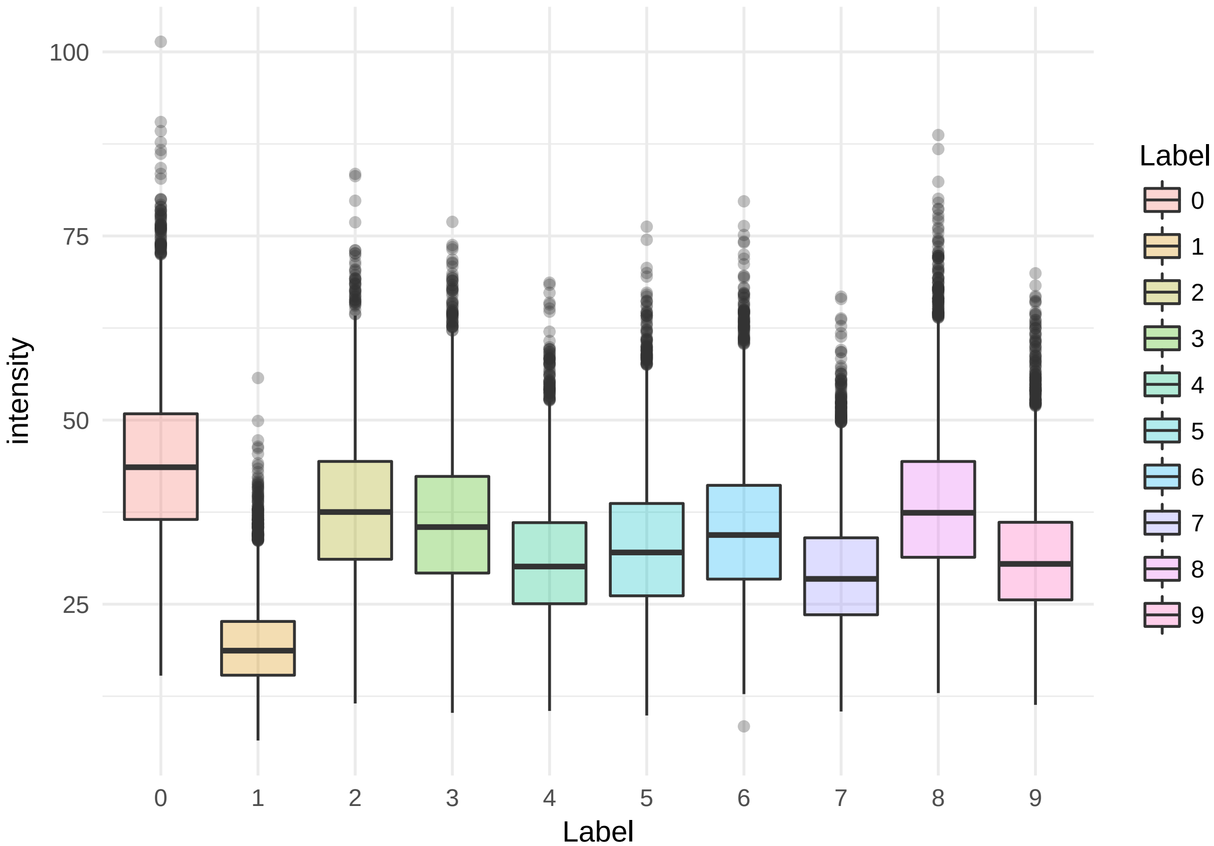
Visualize average intensity by label:

```
ggplot(mnist_df, aes(x = intensity, y = ..density..)) +
  geom_density(aes(fill = Label), alpha = 0.3)
```



Let's try a boxplot:

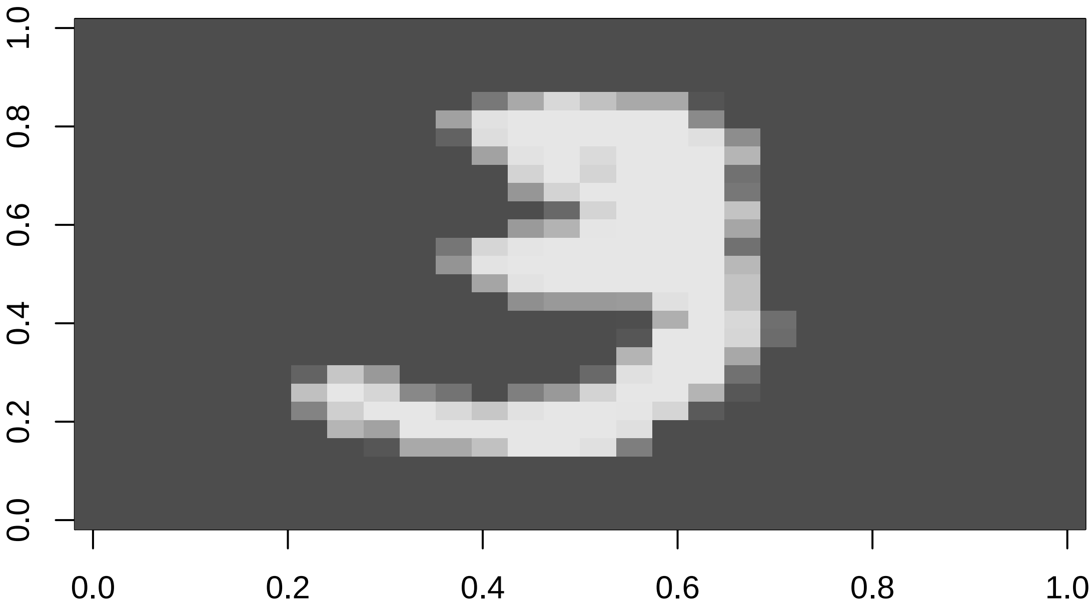
```
ggplot(mnist_df, aes(x = Label, y = intensity)) +
  geom_boxplot(aes(fill = Label), alpha = 0.3)
```



4.3 Visualize Digits

Let's plot a sample set of digits:

```
flip <- function(matrix) {

      apply(matrix, 2, rev)
}

plot_digit <- function(samp) {

  digit <- unlist(samp)
  m <- flip(matrix(rev(as.numeric(digit)), nrow = 28))
  image(m, col = grey.colors(255))

}

mnist_df[11, ] %>%
  select(-Label, -intensity, -splitVar) %>%
  sample_n(1) %>%
  rowwise() %>% plot_digit
```



4.4 Split the Data into Train and Test Sets

```
splits <- rxSplit(mnist_xdf,
                  splitByFactor = "splitVar",
                  overwrite = TRUE)
names(splits) <- c("train", "test")
```

Let's first train a softmax classifier using the `rxLogisticRegression`:

```
softmax <- estimate_model(xdf_data = splits$train,
                          form = make_form(splits$train,
                                           resp_var = "Label",
                                           vars_to_skip = c("splitVar")),
                          model = rxLogisticRegression,
                          type = "multiClass")
```

```
## Automatically adding a MinMax normalization transform, use 'norm=Warn' or 'norm=No' to turn this behavior
## LBFGS multi-threading will attempt to load dataset into memory. In case of out-of-memory issues, turn off
## Beginning optimization
## num vars: 7850
## improvement criterion: Mean Improvement
## L1 regularization selected 3699 of 7850 weights.
## Not training a calibrator because it is not needed.
## Elapsed time: 00:00:17.3789513
## Elapsed time: 00:00:00.0199858
```

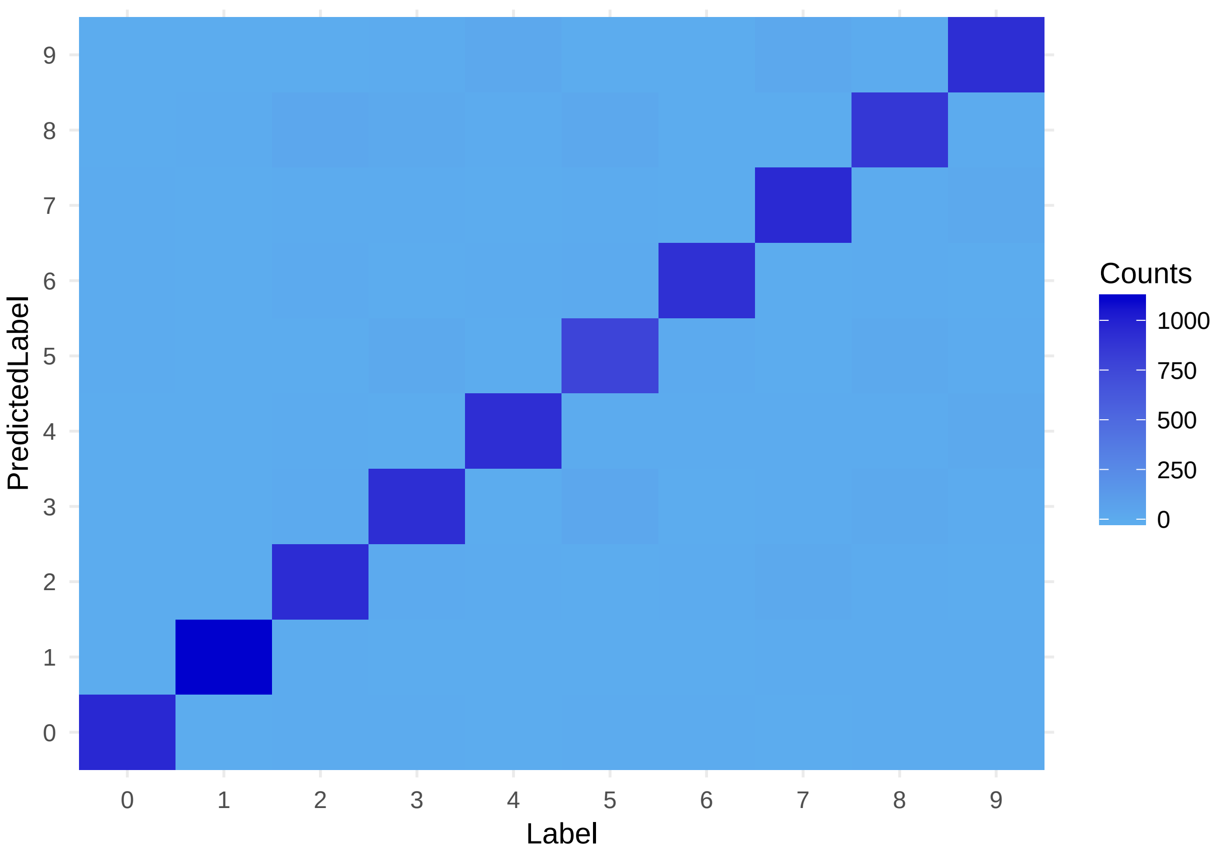
Let's see how we did. Let's examine our results on the train set:

```
softmax_scores <- rxPredict(modelObject = softmax,
                            data = splits$test,
                            outData = tempfile(fileext = ".xdf"),
                            overwrite = TRUE,
                            extraVarsToWrite = "Label")
```

```
## Elapsed time: 00:00:00.9645786
```

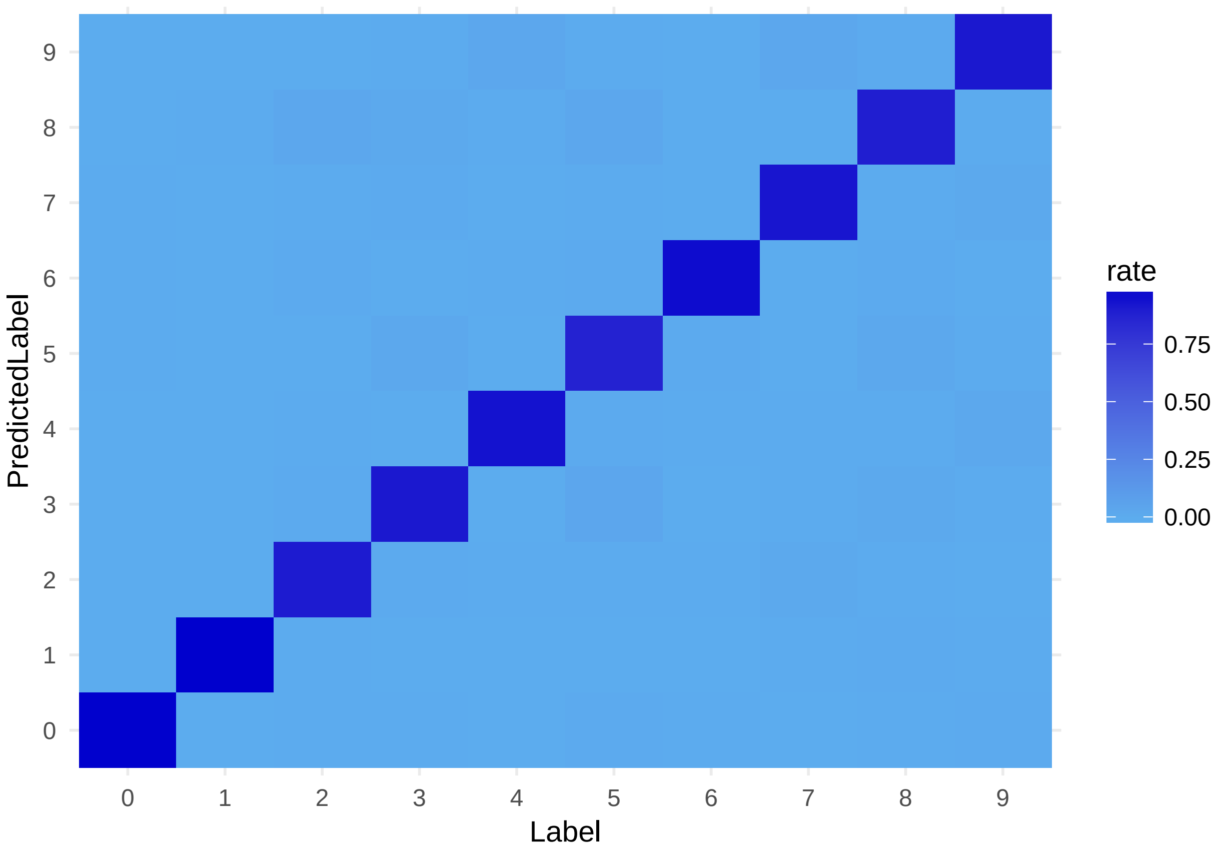
We can make a confusion matrix of all our results:

```
rxCube( ~ Label : PredictedLabel , data = softmax_scores,
       returnDataFrame = TRUE) -> softmax_scores_df

softmax_scores_df %>% ggplot(aes(x = Label, y = PredictedLabel,
                                 fill = Counts)) +
  geom_raster() +
  scale_fill_continuous(low = "steelblue2", high = "mediumblue")
```



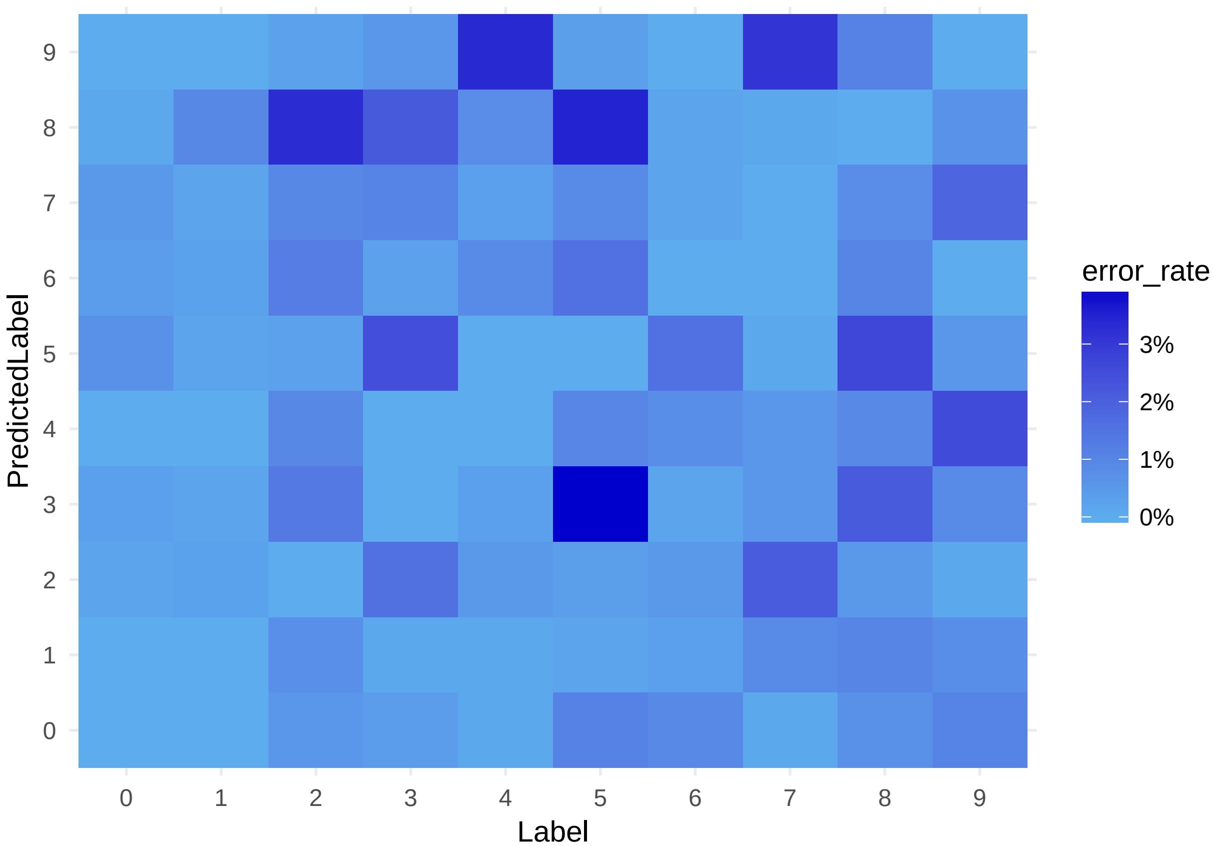
Here we are plotting the raw counts. This might unfairly represent the more populated classes. Let's weight each count by the total number of samples in that class:

```
label_rates <- softmax_scores_df %>%
  tbl_df %>%
  group_by(Label) %>%
  mutate(rate = Counts/sum(Counts))

label_rates %>% ggplot(aes(x = Label, y = PredictedLabel, fill = rate)) +
  geom_raster() +
  scale_fill_continuous(low = "steelblue2", high = "mediumblue")
```



Let's fill out all the correct scores with zeros so we can see the errors more clearly:

```
label_rates %>%
  mutate(error_rate = ifelse(Label == PredictedLabel,
                             0, rate)) %>%
  ggplot(aes(x = Label, y = PredictedLabel, fill = error_rate)) +
  geom_raster() +
  scale_fill_continuous(low = "steelblue2", high = "mediumblue",
                        labels = scales::percent)
```



4.5 Exercises

1. Take a look at David Robinson's tweet on using a single pixel to distinguish between pairs of digits.
2. You can find his gist saved in the Rscripts directory.

Chapter 5

Convolutional Neural Networks for Computer Vision

In the previous section we conducted multi-class classification using a softmax regression algorithm. The most popular approach for image classification is now convolutional neural networks. This module describes how to use convolutional networks.

MicrosoftML uses the Net# specification for defining neural network architectures. In the `../nnet` directory, we have already created the specifications for you.

Examine the architecture in "MNIST.nn". In this network, we have two convolutional layers and one fully connected layer.

```r
library(tidyverse)
library(MicrosoftML)
theme_set(theme_minimal())

rxNeuralNetFile <- file.path("nnet/MNIST.nn")
nn <- readChar(rxNeuralNetFile, file.info(rxNeuralNetFile)$size)
nnet_fit <- rxNeuralNet(make_form(splits$train,
                                  resp_var = "Label",
                                  vars_to_skip = c("splitVar")),
                        data = splits$train,
                        type = "multiClass",
                        numIterations = 9,
                        netDefinition = nn,
                        initWtsDiameter = 1.0,
                        normalize = "No")
```

```
## Not adding a normalizer.
## Using: SSE Math
## Loading net from:
## ***** Net definition *****
##   const T = true;
##   const F = false;
##   input Picture [28, 28];
##   hidden Convolve1 [5, 12, 12] from Picture convolve {
##     InputShape = [28, 28];
##     KernelShape = [5, 5];
##     Stride = [2, 2];
```

```
##     MapCount = 5;
##   }
##   hidden Convolve2 [50, 4, 4] from Convolve1 convolve {
##     InputShape = [5, 12, 12];
##     KernelShape = [1, 5, 5];
##     Stride = [1, 2, 2];
##     Sharing = [F, T, T];
##     MapCount = 10;
##   }
##   hidden Full3 [100] from Convolve2 all;
##   output Result [10] from Full3 all;
## ***** Reduced *****
##   const T = true;
##   const F = false;
##   input Picture [28, 28];
##   hidden Convolve1 [5, 12, 12] from Picture convolve {
##     InputShape = [28, 28];
##     KernelShape = [5, 5];
##     Stride = [2, 2];
##     MapCount = 5;
##   }
##   hidden Convolve2 [50, 4, 4] from Convolve1 convolve {
##     InputShape = [5, 12, 12];
##     KernelShape = [1, 5, 5];
##     Stride = [1, 2, 2];
##     Sharing = [false, true, true];
##     MapCount = 10;
##   }
##   hidden Full3 100 from Convolve2 all;
##   output Result 10 from Full3 all;
## ***** End net definition *****
## Input count: 784
## Output count: 10
## Output Function: SoftMax
## Loss Function: CrossEntropy
## PreTrainer: NoPreTrainer
## ___________________________________________________________________
## Starting training...
## Learning rate: 0.001000
## Momentum: 0.000000
## InitWtsDiameter: 1.000000
## ___________________________________________________________________
## Initializing 3 Hidden Layers, 82540 Weights...
## Estimated Pre-training MeanError = 4.142561
## Iter:1/9, MeanErr=1.582342(-61.80%), 590.01M WeightUpdates/sec
## Iter:2/9, MeanErr=0.651257(-58.84%), 598.09M WeightUpdates/sec
## Iter:3/9, MeanErr=0.474688(-27.11%), 603.13M WeightUpdates/sec
## Iter:4/9, MeanErr=0.390425(-17.75%), 616.42M WeightUpdates/sec
## Iter:5/9, MeanErr=0.344977(-11.64%), 595.88M WeightUpdates/sec
## Iter:6/9, MeanErr=0.314282(-8.90%), 606.88M WeightUpdates/sec
## Iter:7/9, MeanErr=0.290820(-7.47%), 609.71M WeightUpdates/sec
## Iter:8/9, MeanErr=0.273693(-5.89%), 609.05M WeightUpdates/sec
## Iter:9/9, MeanErr=0.256158(-6.41%), 631.54M WeightUpdates/sec
## Done!
```

```
## Estimated Post-training MeanError = 0.247149
## ___________________________________________________________________
## Not training a calibrator because it is not needed.
## Elapsed time: 00:01:25.7559992
```

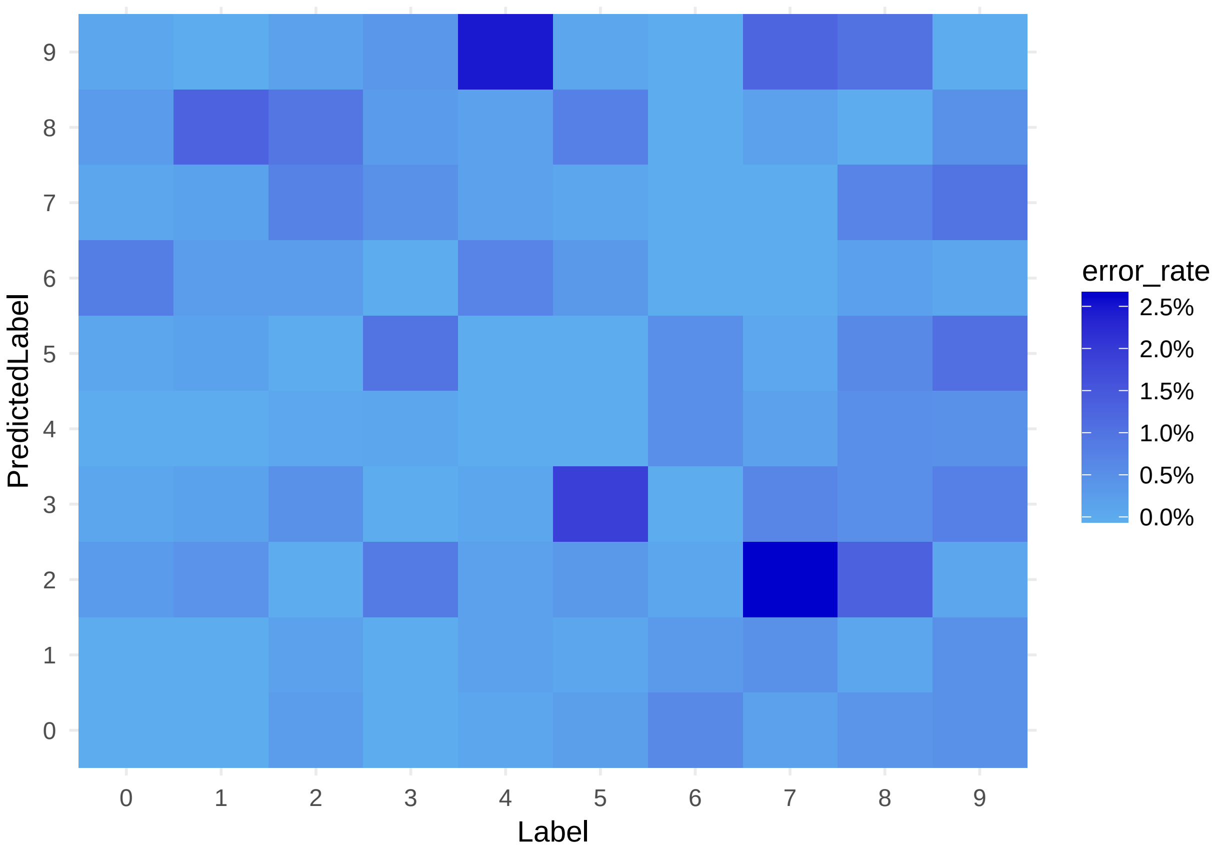
As in the previous section with linear classifiers, we can create our confusion matrices:

```
nnet_score <- rxPredict(modelObject = nnet_fit,
                        data = splits$test,
                        outData = tempfile(fileext = ".xdf"),
                        overwrite = TRUE,
                        extraVarsToWrite = "Label")
```

```
## Elapsed time: 00:00:01.3485847
```

Now that we have our scored results, let's put them in a confusion matrix:

```
rxCube( ~ Label : PredictedLabel , data = nnet_score,
       returnDataFrame = TRUE) -> nnet_scores_df

nnet_scores_df %>%
  tbl_df %>%
  group_by(Label) %>%
  mutate(rate = Counts/sum(Counts)) %>%
  mutate(error_rate = ifelse(Label == PredictedLabel,
                             0, rate)) %>%
  ggplot(aes(x = Label, y = PredictedLabel, fill = error_rate)) +
  geom_raster() +
  scale_fill_continuous(low = "steelblue2", high = "mediumblue",
                        labels = scales::percent)
```



Just judging from the label it looks we have already done better than the linear classifier.

5.1 LeNet-5

Convolutional neural networks were popularized by Yann LeCun. In this section, we'll fit his model from 1998, effectionately called LeNet-5.

The network differs from the previous implementation in that there are now more layers, but in between layers there is a pooling/sampling layer. This helps preventing the neural network from overfitting in between layers and allows for extracting higher-order representations from the data.

Because this neural network has significantly more weights to learn, it'll take a while longer, especially if we aren't using GPUs (which would give us at least 5-7x speed improvement). If you're especially impatient, you can lower the `numIterations` parameter.

```
rxNeuralNetFile <- file.path("nnet/LeCun5.nn")
lecun <- readChar(rxNeuralNetFile, file.info(rxNeuralNetFile)$size)
system.time(lenet_fit <- rxNeuralNet(make_form(splits$train,
                                               resp_var = "Label",
                                               vars_to_skip = c("splitVar")),
                                     data = splits$train,
                                     type = "multiClass",
                                     numIterations = 9,
                                     netDefinition = lecun,
                                     initWtsDiameter = 1.0,
                                     normalize = "No"))
```

```
## Not adding a normalizer.
## Using: SSE Math
## Loading net from:
## ***** Net definition *****
##   const T = true;
##   const F = false;
##   input Picture [28, 28];
##   hidden Convolve1 [6, 28, 28] from Picture convolve {
##     Padding = T;
##     InputShape = [28, 28];
##     KernelShape = [5, 5];
##     MapCount = 6;
##   }
##   hidden Subsample2 [6, 14, 14] linear from Convolve1 convolve {
##     InputShape = [6, 28, 28];
##     KernelShape = [1, 2, 2];
##     Stride = [1, 2, 2];
##     Sharing = [F, T, T];
##   }
##   hidden Convolve3 [16, 10, 10] from Subsample2 convolve {
##     InputShape = [6, 14, 14];
##     KernelShape = [6, 5, 5];
##     MapCount = 16;
##   }
##   hidden Subsample4 [16, 5, 5] linear from Convolve3 convolve {
##     InputShape = [16, 10, 10];
##     KernelShape = [1, 2, 2];
##     Stride = [1, 2, 2];
##     Sharing = [F, T, T];
##   }
##   hidden Convolve5 [120] from Subsample4 convolve {
##     InputShape = [16, 5, 5];
##     KernelShape = [16, 5, 5];
##     MapCount = 120;
##   }
##   hidden Full6 [84] from Convolve5 all;
##   output Result [10] softmax from Full6 all;
## ***** Reduced *****
##   const T = true;
##   const F = false;
##   input Picture [28, 28];
##   hidden Convolve1 [6, 28, 28] from Picture convolve {
##     Padding = true;
##     InputShape = [28, 28];
##     KernelShape = [5, 5];
##     MapCount = 6;
##   }
##   hidden Subsample2 [6, 14, 14] linear from Convolve1 convolve {
##     InputShape = [6, 28, 28];
##     KernelShape = [1, 2, 2];
##     Stride = [1, 2, 2];
##     Sharing = [false, true, true];
##   }
##   hidden Convolve3 [16, 10, 10] from Subsample2 convolve {
```

```
##     InputShape = [6, 14, 14];
##     KernelShape = [6, 5, 5];
##     MapCount = 16;
##   }
##   hidden Subsample4 [16, 5, 5] linear from Convolve3 convolve {
##     InputShape = [16, 10, 10];
##     KernelShape = [1, 2, 2];
##     Stride = [1, 2, 2];
##     Sharing = [false, true, true];
##   }
##   hidden Convolve5 120 from Subsample4 convolve {
##     InputShape = [16, 5, 5];
##     KernelShape = [16, 5, 5];
##     MapCount = 120;
##   }
##   hidden Full6 84 from Convolve5 all;
##   output Result 10 softmax from Full6 all;
## ***** End net definition *****
## Input count: 784
## Output count: 10
## Output Function: SoftMax
## Loss Function: CrossEntropy
## PreTrainer: NoPreTrainer
## ___________________________________________________________________
## Starting training...
## Learning rate: 0.001000
## Momentum: 0.000000
## InitWtsDiameter: 1.000000
## ___________________________________________________________________
## Initializing 6 Hidden Layers, 61816 Weights...
## Estimated Pre-training MeanError = 4.016721
## Iter:1/9, MeanErr=1.566115(-61.01%), 78.16M WeightUpdates/sec
## Iter:2/9, MeanErr=0.508592(-67.53%), 80.91M WeightUpdates/sec
## Iter:3/9, MeanErr=0.350416(-31.10%), 81.29M WeightUpdates/sec
## Iter:4/9, MeanErr=0.274144(-21.77%), 81.72M WeightUpdates/sec
## Iter:5/9, MeanErr=0.231398(-15.59%), 80.61M WeightUpdates/sec
## Iter:6/9, MeanErr=0.200124(-13.52%), 80.79M WeightUpdates/sec
## Iter:7/9, MeanErr=0.180827(-9.64%), 80.66M WeightUpdates/sec
## Iter:8/9, MeanErr=0.165253(-8.61%), 80.67M WeightUpdates/sec
## Iter:9/9, MeanErr=0.154163(-6.71%), 80.19M WeightUpdates/sec
## Done!
## Estimated Post-training MeanError = 0.140622
## ___________________________________________________________________
## Not training a calibrator because it is not needed.
## Elapsed time: 00:07:32.4435112
```

```
##    user  system elapsed
##    0.04    0.00  452.57
```

As before, let's score our pretty model:

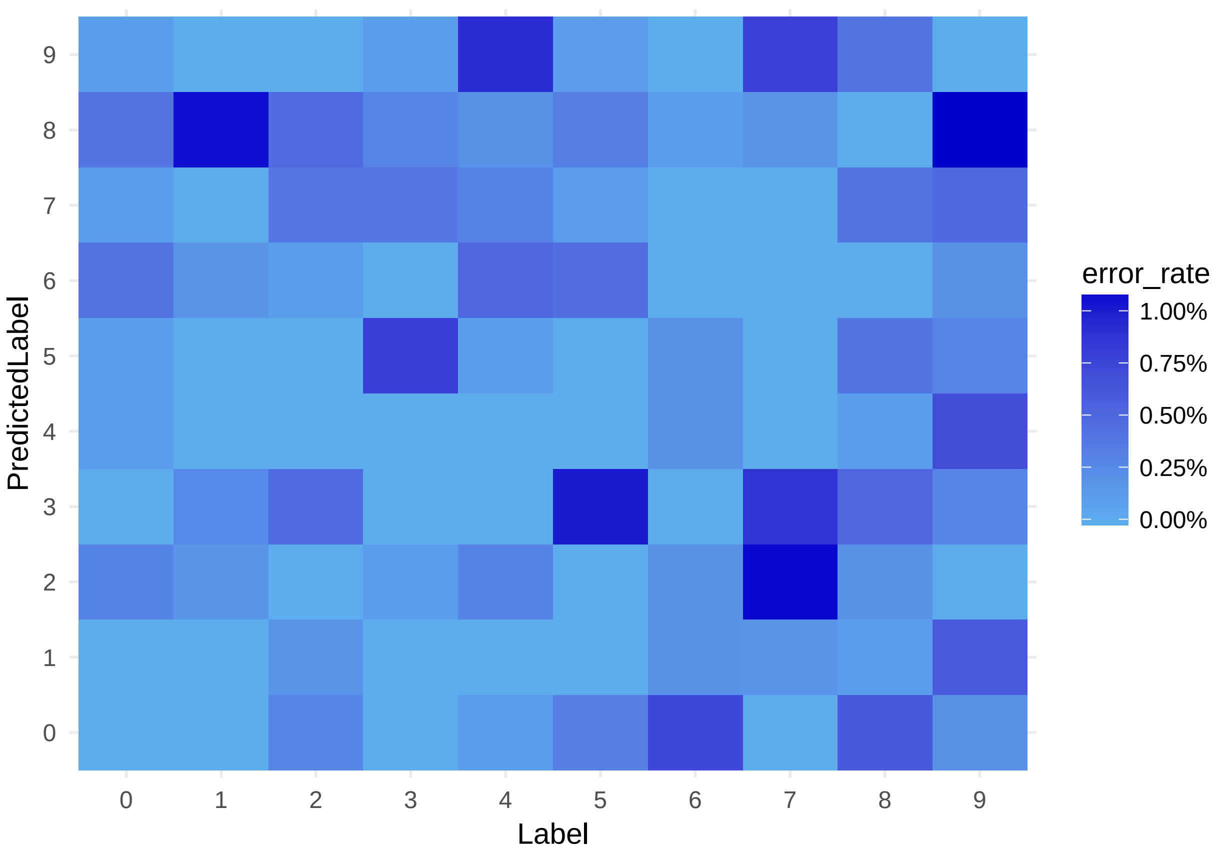
```
lescores <- rxPredict(modelObject = lenet_fit,
                      data = splits$test,
                      outData = tempfile(fileext = ".xdf"),
                      overwrite = TRUE,
```

```
                              extraVarsToWrite = "Label")
```

```
## Elapsed time: 00:00:03.4389931
```

and visualize our error rates:

```
rxCube( ~ Label : PredictedLabel , data = lescores,
       returnDataFrame = TRUE) -> le_scores_df

le_scores_df %>%
  tbl_df %>%
  group_by(Label) %>%
  mutate(rate = Counts/sum(Counts)) %>%
  mutate(error_rate = ifelse(Label == PredictedLabel,
                              0, rate)) %>%
  ggplot(aes(x = Label, y = PredictedLabel, fill = error_rate)) +
  geom_raster() +
  scale_fill_continuous(low = "steelblue2", high = "mediumblue",
                        labels = scales::percent)
```



Looks even better!

5.2 Model Metrics

While our visualizations provide some insight into our model's improvement, let's try to calculate emprical metrics of our models' performacne.

The three metrics we'll focus on are "accuracy", "precision", and "recall". Accuracy simply measures how many of our estimates we classified correctly. While simple and intuitive, it does not account for class-imbalances. For example, if 99% of our data is in class A, and we simply use the rule that everything is class A, we'll get an accuracy of 99%. Sounds impressive, but probably not going to win any Turing tests.

5.2.1 Accuracy

To calculate accuracy, we can simply measure the sum of our confusion matrix's diagonal over all values. Our data was in a long format to make it amenable for visualizations using ggplot2. Here we'll use `tidyr` to put it into a wide format amenable for calculating model metrics quickly and efficiently.

```
calc_accuracy <- function(scores_df) {

  library(tidyr)

  scores_df <- as.data.frame(scores_df)
  scores_conf <- scores_df %>% spread(PredictedLabel, Counts)

  scores_conf <- as.matrix(scores_conf[, 2:ncol(scores_conf)])
  sum(diag(scores_conf))/sum(scores_conf)

}

sprintf("Accuracy of the softmax model is %s", calc_accuracy(softmax_scores_df))
```

```
## [1] "Accuracy of the softmax model is 0.927"
```

```
sprintf("Accuracy of the convolutional model is %s", calc_accuracy(nnet_scores_df))
```

```
## [1] "Accuracy of the convolutional model is 0.9628"
```

```
sprintf("Accuracy of the LeCun-5 model is %s", calc_accuracy(le_scores_df))
```

```
## [1] "Accuracy of the LeCun-5 model is 0.977"
```

5.2.2 Precision

Precision is another measure of model performance. It calculates the ratio of true positives to all values, i.e., how precise your model is in classifying any digit. To calculate precision, we'll take the diagonal of our confusion matrix over the sum of that column.

```
calc_precision <- function(scores_df) {

  library(tidyr)

  scores_df <- as.data.frame(scores_df)
  scores_conf <- scores_df %>% spread(PredictedLabel, Counts)

  scores_conf <- as.matrix(scores_conf[, 2:ncol(scores_conf)])
  diag(scores_conf)/colSums(scores_conf)

}

calc_precision(softmax_scores_df)
```

```
##         0         1         2         3         4         5         6
## 0.9513406 0.9636049 0.9375629 0.9065880 0.9311044 0.9013921 0.9421488
##         7         8         9
## 0.9332024 0.8810976 0.9128713
```

5.2.3 Recall

Lastly, we can calculate recall. Recall is a measure of how relevant the predictions are for the given class, i.e., how many of the actual classes were properly predicted. In this case, we'll sum over the predicted labels rather than the actual labels:

```
calc_recall <- function(scores_df) {

  library(tidyr)

  scores_df <- as.data.frame(scores_df)
  scores_conf <- scores_df %>% spread(PredictedLabel, Counts)

  scores_conf <- as.matrix(scores_conf[, 2:ncol(scores_conf)])
  diag(scores_conf)/rowSums(scores_conf)

}

calc_recall(softmax_scores_df)
```

```
##         1         2         3         4         5         6         7
## 0.9775510 0.9797357 0.9021318 0.9128713 0.9358452 0.8710762 0.9519833
##         8         9        10
## 0.9241245 0.8901437 0.9137760
```

5.2.4 Visualzing our Metrics

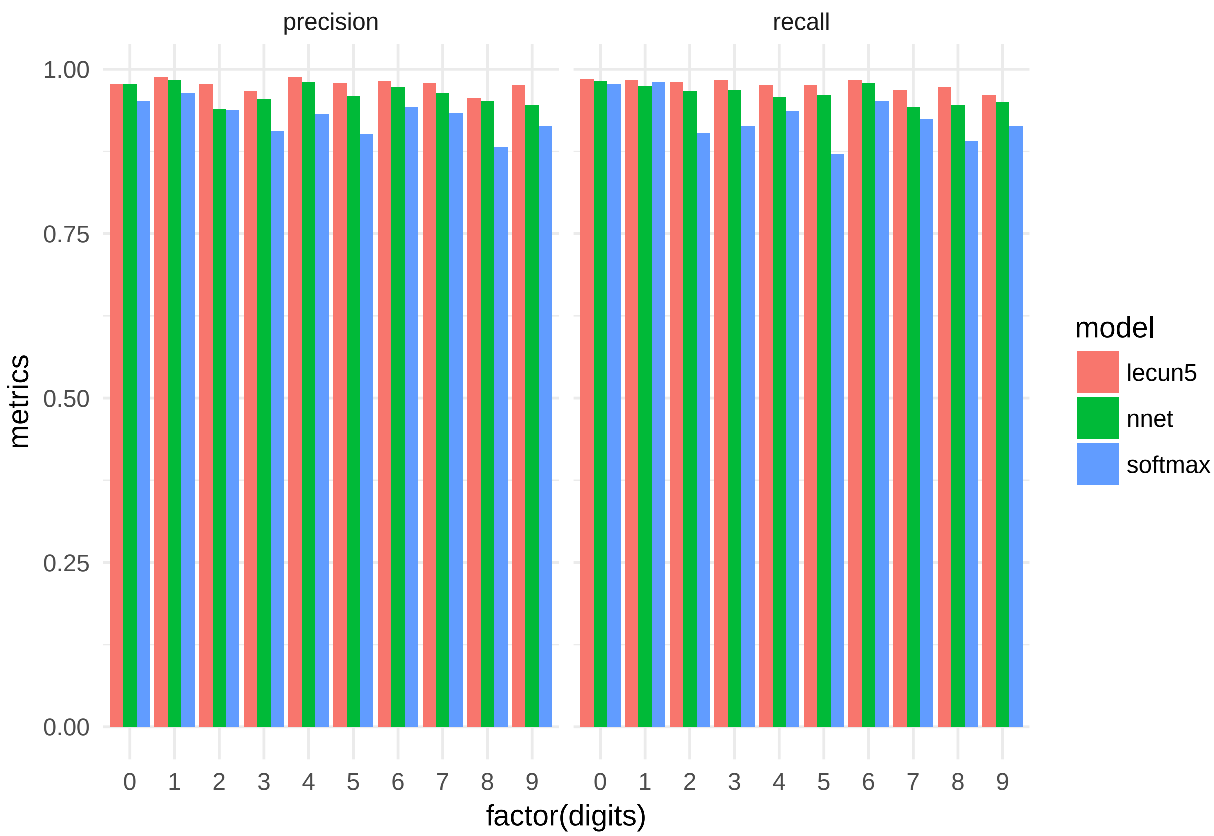
Let's calculate the metrics for all three of our mdoels and visualize them.

```
results <- list(softmax = softmax_scores_df,
                nnet = nnet_scores_df,
                lecun5 = le_scores_df)

metrics_df <- data.frame(
  map_df(results, calc_precision),
  digits = 0:9,
  metric = rep("precision", 10)
) %>%
  bind_rows(data.frame(
    map_df(results, calc_recall),
    digits = 0:9,
    metric = rep("recall", 10))
  )
```

```
## Warning in bind_rows_(x, .id): Unequal factor levels: coercing to character

## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector
```

```
## Warning in bind_rows_(x, .id): binding character and factor vector,
## coercing into character vector
```

```
metrics_df %>% gather(model, metrics, -digits, -metric) %>%
  ggplot(aes(x = factor(digits),
             y = metrics,
             fill = model)) +
  geom_bar(stat = 'identity', position = "dodge") +
  facet_wrap(~metric) + theme_minimal()
```



Chapter 6

Natural Language Processing

6.1 Text Classification

Let's take a look at using MML to estimate a model that would be very hard to do with RevoScaleR.

In particular, there are virtually no functionality in RevoScaleR for handling large text data. We will use MML to transform text data into useful features that we can use in a logistic regression learner. In order to deal with the high cardinality of text data, we will use the penalized regression models in MML.

6.1.1 IMDB Data

Our data is taken from the paper **Learning Word Vectors for Sentiment Analysis** written in 2011 by Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. The paper and data are available here: http://ai.stanford.edu/~amaas/data/sentiment/. I've already downloaded and converted the data into an XDF. Please see the 1-ingest-data.R script if you are interested in the ingestion process.

```
library(MicrosoftML)
library(tidyverse)
library(d3wordcloud)
train_xdf <- RxXdfData("data/imdb-train.xdf")
test_xdf <- RxXdfData("data/imdb-test.xdf")
```

6.1.2 Feature Transformers

MicrosoftML has a set of functions for feature engineering. In this example, let's take a look at creating sparse word vectors.

We'll use the featurizeText function to convert our text data into numeric columns. In particular, we'll ask for new columns with tri-grams after removing stopwords, punctuations, and numbers.

We can do this transform directly in our modeling call, and in particular, we'll train logistic regression models and a fast gradient boosted tree model:

```
system.time(logit_model <- rxLogisticRegression(sentiment ~ reviewTran,
                                                data = train_xdf,
                                                l1Weight = 0.05,
                                                l2Weight = 0.01,
```

```
                                                    mlTransforms =
                                                      list(featurizeText(
                                                        vars = c(reviewTran = "review"),
                                                        language = "English",
                                                        stopwordsRemover = stopwordsDefault(),
                                                        wordFeatureExtractor =
                                                          ngramCount(ngramLength = 3,
                                                                     weighting = "tfidf",
                                                                     maxNumTerms = 1e+09),
                                                        keepNumbers = FALSE,
                                                        keepPunctuations = FALSE)
                                                        )
                                                    )
            )
```

```
## Not adding a normalizer.
## Automatically converting column 'sentiment' into a factor.
## LBFGS multi-threading will attempt to load dataset into memory. In case of out-of-memory issues, turn off
## Beginning optimization
## num vars: 4308438
## improvement criterion: Mean Improvement
## L1 regularization selected 28533 of 4308438 weights.
## Not training a calibrator because it is not needed.
## Elapsed time: 00:02:56.9912387
## Elapsed time: 00:00:52.3671094
```

```
##    user  system elapsed
##   0.184   0.104 230.296
```

```
system.time(fast_trees <- rxFastTrees(sentiment ~ reviewTran,
                                       data = train_xdf,
                                       mlTransforms =
                                                  list(featurizeText(
                                                    vars = c(reviewTran = "review"),
                                                    language = "English",
                                                    stopwordsRemover = stopwordsDefault(),
                                                    wordFeatureExtractor =
                                                      ngramCount(ngramLength = 3,
                                                                 weighting = "tfidf",
                                                                 maxNumTerms = 1e+09),
                                                    keepNumbers = FALSE,
                                                    keepPunctuations = FALSE)
                                                    )
                                      )
            )
```

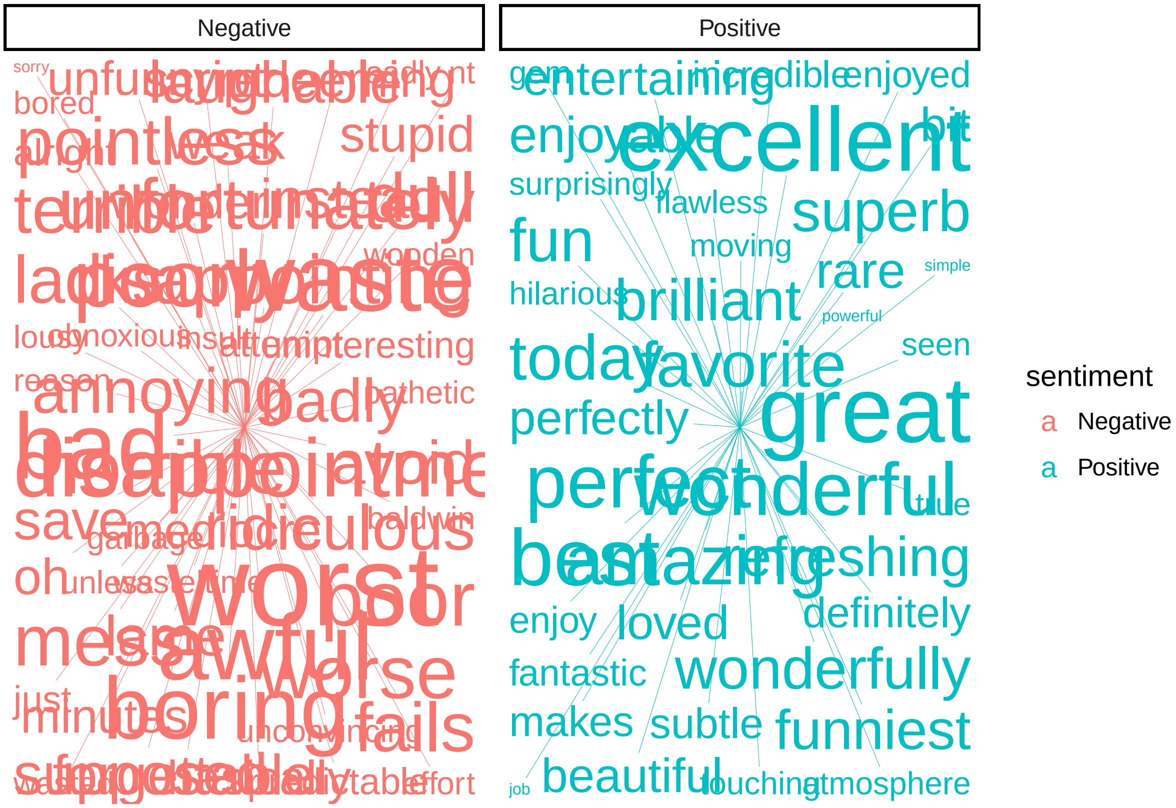
```
## Not adding a normalizer.
## Automatically converting column 'sentiment' into a factor.
## Making per-feature arrays
## Changing data from row-wise to column-wise
## Processed 25000 instances
## Binning and forming Feature objects
## Reserved memory for tree learner: 304827068 bytes
## Starting to train ...
## Not training a calibrator because it is not needed.
```

```
## Elapsed time: 00:01:09.6558035
```

```
##    user  system elapsed
##   0.020   0.068  69.839
```

Now that we have our trained model, we can do some visualizations. For example, for the elastic net, we can visualize the coefficients.

```
logit_cof <- coefficients(logit_model)
coefs <- data.frame(coef = logit_cof, word = names(logit_cof))
coefs <- tbl_df(coefs)

coefs <- coefs %>%
  filter(word != "(Bias)") %>%
  mutate(abs_value = abs(coef),
         sentiment = ifelse(coef > 0, "Positive", "Negative"),
         score = round(abs_value, 0)) %>%
  arrange(desc(abs_value)) %>% slice(1:100)


library(ggplot2)
library(ggrepel)

coefs %>%
  ggplot +
    aes(x = 1, y = 1, colour = sentiment, size = score, label = word) +
    geom_text_repel(segment.size = 0, force = 10) +
    scale_size(range = c(2, 15), guide = FALSE) +
    scale_y_continuous(breaks = NULL) +
    scale_x_continuous(breaks = NULL) +
    labs(x = '', y = '') +
    theme_classic() +
    facet_wrap(~sentiment)
```



Let's try and makea more interactive visual. We'll use `purrr` again to map our coefficients to the beautiful d3wordcloud package

```
coefs %>%
  split(.$sentiment) %>%
  purrr::map( ~ d3wordcloud(.$word, .$score, tooltip = TRUE)) -> d3_graphs

d3_graphs[[1]]
```

```
d3_graphs[[2]]
```

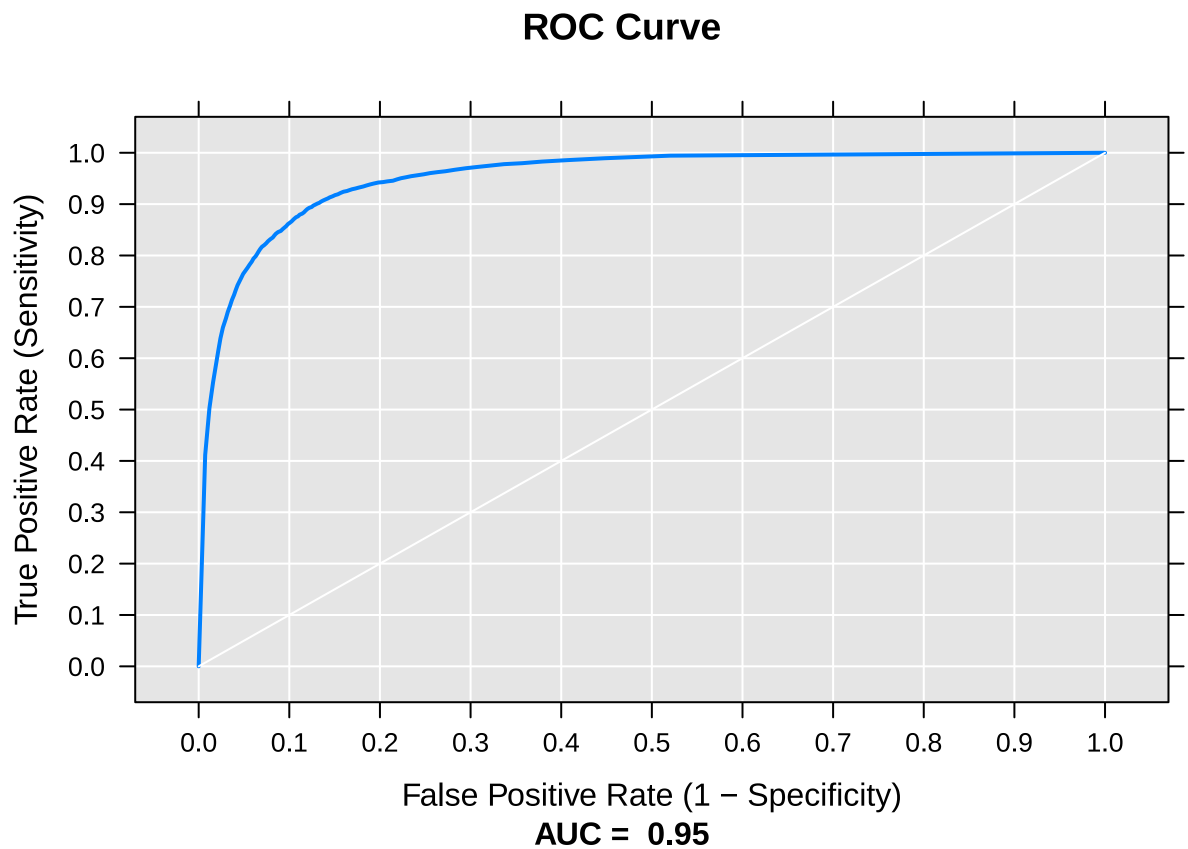
6.1.3 Testing the Logit Model

In order to predict our classifer on test data, we will use the `mxPredict` function from the `MML` package.

```
predictions <- rxPredict(logit_model, data = test_xdf, extraVarsToWrite = "sentiment")
```

```
## Elapsed time: 00:00:26.5864958
```

```
roc_results <- rxRoc(actualVarName = "sentiment", predVarNames = "Probability.1", data = predictions)
roc_results$predVarName <- factor(roc_results$predVarName)
plot(roc_results)
```

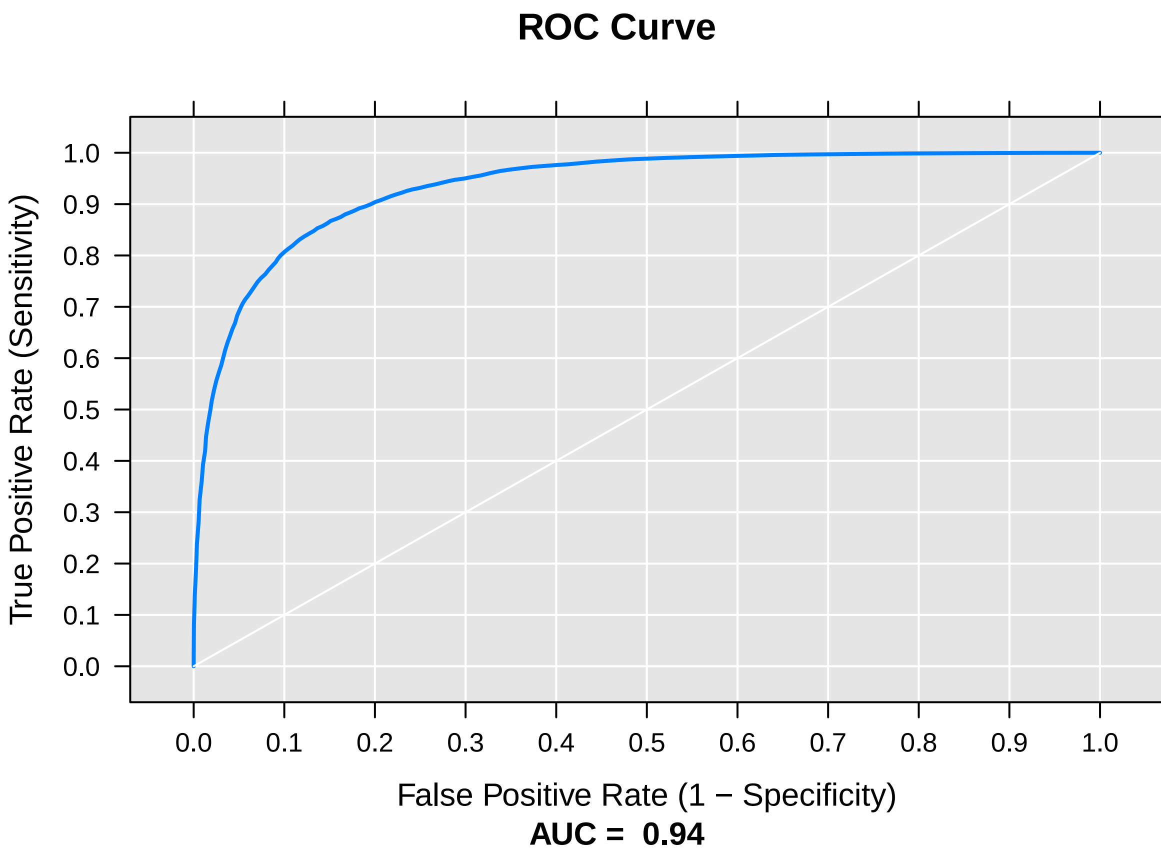


6.1.4 Testing the Fast Trees Model

```
predictions <- rxPredict(fast_trees, data = test_xdf, extraVarsToWrite = "sentiment")
```

```
## Elapsed time: 00:00:29.4280653
```

```
roc_results <- rxRoc(actualVarName = "sentiment", predVarNames = "Probability.1", data = predictions)
roc_results$predVarName <- factor(roc_results$predVarName)
plot(roc_results)
```



6.2 Neural Networks

Let's try to estimate another binary classifier from this dataset, but with a Neural Network architecture rather than a logistic regression model.

In the following chunk, we call our neural network model, and set the optimizer to be a stochastic gradient descent optimizer with a learning rate of 0.2. Furthermore, we use the `type` argument to ensure we are learning a binary classifier. By default our network architecture will have 100 hidden nodes.

```
nn_sentiment <- rxNeuralNet(sentiment ~ reviewTran,
                            data = train_xdf,
                            type = "binary",
                            mlTransforms = list(featurizeText(vars = c(reviewTran = "review"),
                                                             language = "English",
                                                             stopwordsRemover = stopwordsDefault(),
                                                             keepPunctuations = FALSE)),
                          # acceleration = "gpu",
                          miniBatchSize = 4)
```

```
## Not adding a normalizer.
## Automatically converting column 'sentiment' into a factor.
## Using: SSE Math
## Warning: Math acceleration mode not compatible with mini-batches. Setting batch size to 1.
## ***** Net definition *****
##   input Data [74398];
##   hidden H [100] sigmoid { // Depth 1
```

```
##     from Data all;
##   }
##   output Result [1] sigmoid { // Depth 0
##     from H all;
##   }
## ***** End net definition *****
## Input count: 74398
## Output count: 1
## Output Function: Sigmoid
## Loss Function: CrossEntropy
## PreTrainer: NoPreTrainer
## ___________________________________________________________________
## Starting training...
## Learning rate: 0.001000
## Momentum: 0.000000
## InitWtsDiameter: 0.100000
## ___________________________________________________________________
## Initializing 1 Hidden Layers, 7440001 Weights...
## Estimated Pre-training MeanError = 0.704585
## Iter:1/100, MeanErr=0.694443(-1.44%), 125415.57M WeightUpdates/sec
## Iter:2/100, MeanErr=0.694329(-0.02%), 130494.76M WeightUpdates/sec
## Iter:3/100, MeanErr=0.694192(-0.02%), 132241.61M WeightUpdates/sec
## Iter:4/100, MeanErr=0.694030(-0.02%), 131118.71M WeightUpdates/sec
## Iter:5/100, MeanErr=0.694215(0.03%), 128647.66M WeightUpdates/sec
## Iter:6/100, MeanErr=0.693894(-0.05%), 126401.64M WeightUpdates/sec
## Iter:7/100, MeanErr=0.693471(-0.06%), 128879.81M WeightUpdates/sec
## Iter:8/100, MeanErr=0.692859(-0.09%), 134158.70M WeightUpdates/sec
## Iter:9/100, MeanErr=0.692376(-0.07%), 127289.61M WeightUpdates/sec
## Iter:10/100, MeanErr=0.691277(-0.16%), 125183.79M WeightUpdates/sec
## Iter:11/100, MeanErr=0.690757(-0.08%), 124604.89M WeightUpdates/sec
## Iter:12/100, MeanErr=0.689446(-0.19%), 127705.30M WeightUpdates/sec
## Iter:13/100, MeanErr=0.687619(-0.26%), 128392.39M WeightUpdates/sec
## Iter:14/100, MeanErr=0.685640(-0.29%), 130021.94M WeightUpdates/sec
## Iter:15/100, MeanErr=0.683191(-0.36%), 131894.86M WeightUpdates/sec
## Iter:16/100, MeanErr=0.680226(-0.43%), 136040.52M WeightUpdates/sec
## Iter:17/100, MeanErr=0.676299(-0.58%), 134689.88M WeightUpdates/sec
## Iter:18/100, MeanErr=0.671884(-0.65%), 126922.87M WeightUpdates/sec
## Iter:19/100, MeanErr=0.666806(-0.76%), 126259.57M WeightUpdates/sec
## Iter:20/100, MeanErr=0.660784(-0.90%), 128367.22M WeightUpdates/sec
## Iter:21/100, MeanErr=0.653908(-1.04%), 127823.27M WeightUpdates/sec
## Iter:22/100, MeanErr=0.645968(-1.21%), 125449.66M WeightUpdates/sec
## Iter:23/100, MeanErr=0.637465(-1.32%), 128992.18M WeightUpdates/sec
## Iter:24/100, MeanErr=0.628361(-1.43%), 128921.78M WeightUpdates/sec
## Iter:25/100, MeanErr=0.618374(-1.59%), 128686.66M WeightUpdates/sec
## Iter:26/100, MeanErr=0.607630(-1.74%), 125625.10M WeightUpdates/sec
## Iter:27/100, MeanErr=0.596812(-1.78%), 126844.28M WeightUpdates/sec
## Iter:28/100, MeanErr=0.585189(-1.95%), 127762.36M WeightUpdates/sec
## Iter:29/100, MeanErr=0.573419(-2.01%), 136768.58M WeightUpdates/sec
## Iter:30/100, MeanErr=0.561505(-2.08%), 136910.76M WeightUpdates/sec
## Iter:31/100, MeanErr=0.549777(-2.09%), 135253.79M WeightUpdates/sec
## Iter:32/100, MeanErr=0.538191(-2.11%), 138856.24M WeightUpdates/sec
## Iter:33/100, MeanErr=0.526569(-2.16%), 133778.06M WeightUpdates/sec
## Iter:34/100, MeanErr=0.515547(-2.09%), 126691.70M WeightUpdates/sec
## Iter:35/100, MeanErr=0.504761(-2.09%), 130933.52M WeightUpdates/sec
```

```
## Iter:36/100, MeanErr=0.494128(-2.11%), 126065.20M WeightUpdates/sec
## Iter:37/100, MeanErr=0.484511(-1.95%), 127670.71M WeightUpdates/sec
## Iter:38/100, MeanErr=0.474862(-1.99%), 126004.34M WeightUpdates/sec
## Iter:39/100, MeanErr=0.465674(-1.93%), 128223.43M WeightUpdates/sec
## Iter:40/100, MeanErr=0.456896(-1.88%), 129744.36M WeightUpdates/sec
## Iter:41/100, MeanErr=0.448901(-1.75%), 131564.48M WeightUpdates/sec
## Iter:42/100, MeanErr=0.441171(-1.72%), 128833.48M WeightUpdates/sec
## Iter:43/100, MeanErr=0.433480(-1.74%), 130683.41M WeightUpdates/sec
## Iter:44/100, MeanErr=0.426293(-1.66%), 132297.95M WeightUpdates/sec
## Iter:45/100, MeanErr=0.419575(-1.58%), 126329.45M WeightUpdates/sec
## Iter:46/100, MeanErr=0.413401(-1.47%), 127071.73M WeightUpdates/sec
## Iter:47/100, MeanErr=0.406957(-1.56%), 124416.71M WeightUpdates/sec
## Iter:48/100, MeanErr=0.401313(-1.39%), 125157.72M WeightUpdates/sec
## Iter:49/100, MeanErr=0.395543(-1.44%), 125105.73M WeightUpdates/sec
## Iter:50/100, MeanErr=0.390251(-1.34%), 126154.53M WeightUpdates/sec
## Iter:51/100, MeanErr=0.385156(-1.31%), 122496.72M WeightUpdates/sec
## Iter:52/100, MeanErr=0.380155(-1.30%), 134559.53M WeightUpdates/sec
## Iter:53/100, MeanErr=0.375252(-1.29%), 124359.93M WeightUpdates/sec
## Iter:54/100, MeanErr=0.371025(-1.13%), 127304.69M WeightUpdates/sec
## Iter:55/100, MeanErr=0.366597(-1.19%), 125388.90M WeightUpdates/sec
## Iter:56/100, MeanErr=0.362269(-1.18%), 126106.61M WeightUpdates/sec
## Iter:57/100, MeanErr=0.358292(-1.10%), 133691.33M WeightUpdates/sec
## Iter:58/100, MeanErr=0.354345(-1.10%), 132204.62M WeightUpdates/sec
## Iter:59/100, MeanErr=0.350553(-1.07%), 125161.28M WeightUpdates/sec
## Iter:60/100, MeanErr=0.346730(-1.09%), 137173.58M WeightUpdates/sec
## Iter:61/100, MeanErr=0.343272(-1.00%), 126914.94M WeightUpdates/sec
## Iter:62/100, MeanErr=0.339770(-1.02%), 135821.28M WeightUpdates/sec
## Iter:63/100, MeanErr=0.336230(-1.04%), 134923.71M WeightUpdates/sec
## Iter:64/100, MeanErr=0.333191(-0.90%), 126294.75M WeightUpdates/sec
## Iter:65/100, MeanErr=0.330051(-0.94%), 126416.33M WeightUpdates/sec
## Iter:66/100, MeanErr=0.326981(-0.93%), 124905.46M WeightUpdates/sec
## Iter:67/100, MeanErr=0.323995(-0.91%), 124701.09M WeightUpdates/sec
## Iter:68/100, MeanErr=0.321182(-0.87%), 125438.43M WeightUpdates/sec
## Iter:69/100, MeanErr=0.318367(-0.88%), 127085.70M WeightUpdates/sec
## Iter:70/100, MeanErr=0.315463(-0.91%), 122849.20M WeightUpdates/sec
## Iter:71/100, MeanErr=0.313039(-0.77%), 125325.57M WeightUpdates/sec
## Iter:72/100, MeanErr=0.310367(-0.85%), 126297.57M WeightUpdates/sec
## Iter:73/100, MeanErr=0.307799(-0.83%), 125278.45M WeightUpdates/sec
## Iter:74/100, MeanErr=0.305068(-0.89%), 127582.72M WeightUpdates/sec
## Iter:75/100, MeanErr=0.302958(-0.69%), 128067.18M WeightUpdates/sec
## Iter:76/100, MeanErr=0.300405(-0.84%), 127380.58M WeightUpdates/sec
## Iter:77/100, MeanErr=0.297983(-0.81%), 125099.27M WeightUpdates/sec
## Iter:78/100, MeanErr=0.295944(-0.68%), 127625.57M WeightUpdates/sec
## Iter:79/100, MeanErr=0.293907(-0.69%), 125770.09M WeightUpdates/sec
## Iter:80/100, MeanErr=0.291620(-0.78%), 132136.38M WeightUpdates/sec
## Iter:81/100, MeanErr=0.289745(-0.64%), 125433.57M WeightUpdates/sec
## Iter:82/100, MeanErr=0.287662(-0.72%), 127810.74M WeightUpdates/sec
## Iter:83/100, MeanErr=0.285591(-0.72%), 128472.66M WeightUpdates/sec
## Iter:84/100, MeanErr=0.283583(-0.70%), 126612.25M WeightUpdates/sec
## Iter:85/100, MeanErr=0.281815(-0.62%), 127929.90M WeightUpdates/sec
## Iter:86/100, MeanErr=0.279837(-0.70%), 126589.35M WeightUpdates/sec
## Iter:87/100, MeanErr=0.278120(-0.61%), 130552.71M WeightUpdates/sec
## Iter:88/100, MeanErr=0.276277(-0.66%), 127951.54M WeightUpdates/sec
## Iter:89/100, MeanErr=0.274533(-0.63%), 130041.89M WeightUpdates/sec
```

```
## Iter:90/100, MeanErr=0.272938(-0.58%), 127955.84M WeightUpdates/sec
## Iter:91/100, MeanErr=0.270917(-0.74%), 126596.53M WeightUpdates/sec
## Iter:92/100, MeanErr=0.269519(-0.52%), 127407.53M WeightUpdates/sec
## Iter:93/100, MeanErr=0.267711(-0.67%), 128363.02M WeightUpdates/sec
## Iter:94/100, MeanErr=0.266137(-0.59%), 125147.07M WeightUpdates/sec
## Iter:95/100, MeanErr=0.264770(-0.51%), 126902.31M WeightUpdates/sec
## Iter:96/100, MeanErr=0.262845(-0.73%), 128173.66M WeightUpdates/sec
## Iter:97/100, MeanErr=0.261513(-0.51%), 127248.96M WeightUpdates/sec
## Iter:98/100, MeanErr=0.259934(-0.60%), 126299.90M WeightUpdates/sec
## Iter:99/100, MeanErr=0.258394(-0.59%), 126134.74M WeightUpdates/sec
## Iter:100/100, MeanErr=0.257150(-0.48%), 134732.80M WeightUpdates/sec
## Done!
## Estimated Post-training MeanError = 0.256146
## ___________________________________________________________________
## Not training a calibrator because it is not needed.
## Elapsed time: 00:02:34.9539664
```

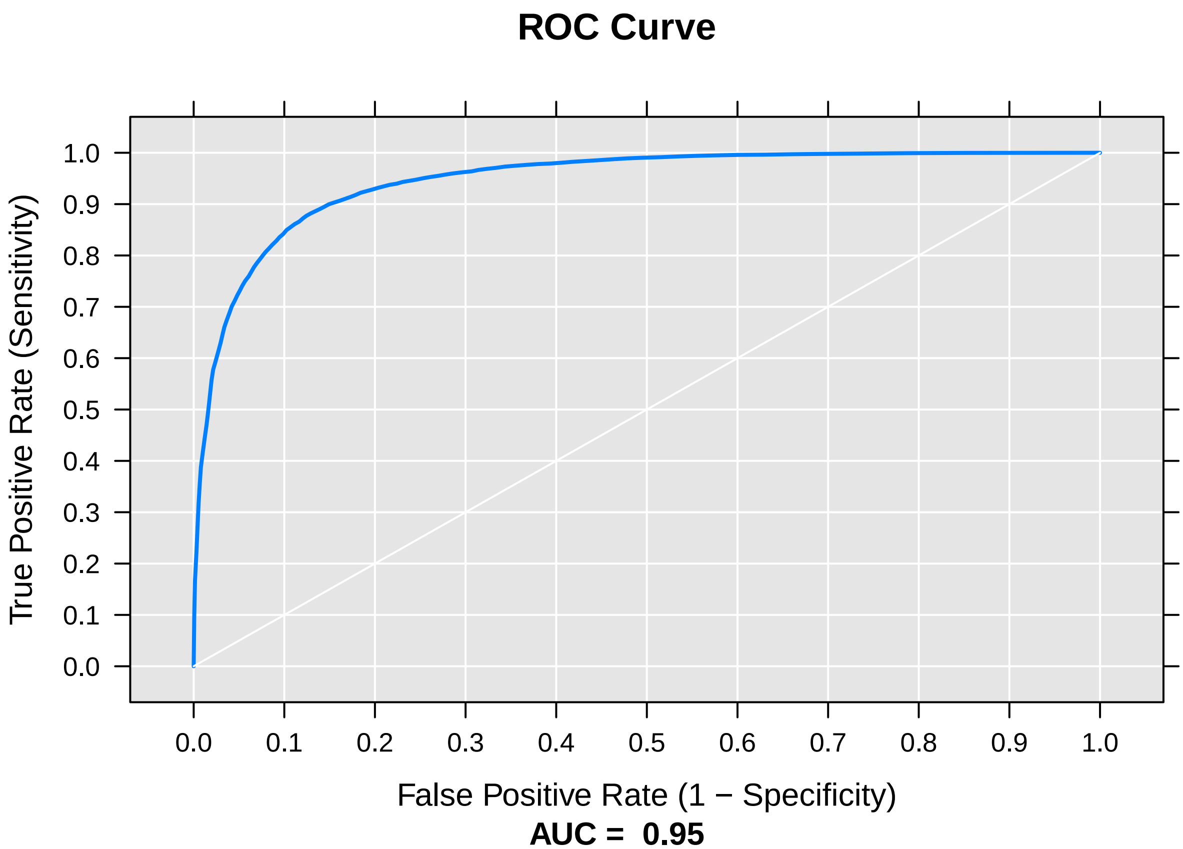
6.2.1 Scoring the Neural Net

We can similary score our results from the neural network model

```
predictions <- rxPredict(nn_sentiment, data = test_xdf, extraVarsToWrite = "sentiment")
```

```
## Elapsed time: 00:00:12.8994661
```

```
roc_results <- rxRoc(actualVarName = "sentiment", predVarNames = "Probability.1", data = predictions)
roc_results$predVarName <- factor(roc_results$predVarName)
plot(roc_results)
```



6.3 Exercises

1. The Rscript Rscripts/9-Other-Sentiment-Datasets.R has two additional datasets with reviews and ratings (binarized). Try the above analysis on the other two datasets.

Chapter 7

Transfer Learning with Pre-Trained Deep Neural Network Architectures – The Shallow End of Deep Learning

7.1 Pre-Trained Models

Transfer learning is a pretty incredible method for learning expressive models without having to train a deep architecture from scratch. In some ways, it's nearly a "free-lunch": take a pre-built model trained for weeks on a large image corpus, and reuse the features from that model for your domain-specific task.

`MicrosoftML` ships with a number of pre-trained DNNs on the ImageNet challenge dataset.

```
AlexNetFeatures <- read.csv(system.file(
  "extdata/ImageAnalyticsTestData/featurizeImage_alexnet_output.csv",
  package = "MicrosoftML"),
  header = FALSE)

ResNet18Features <- read.csv(system.file(
  "extdata/ImageAnalyticsTestData/featurizeImage_resnet18_output.csv",
  package = "MicrosoftML"),
  header = FALSE)

ResNet50Features <- read.csv(system.file(
  "extdata/ImageAnalyticsTestData/featurizeImage_resnet50_output.csv",
  package = "MicrosoftML"),
  header = FALSE)

ResNet101Features <- read.csv(system.file(
  "extdata/ImageAnalyticsTestData/featurizeImage_resnet101_output.csv",
  package = "MicrosoftML"),
  header = FALSE)

lapply(list(AlexNetFeatures, ResNet18Features, ResNet50Features, ResNet101Features),
       dim)
```

```
## [[1]]
## [1]    2 4097
```

```
##
## [[2]]
## [1]   2 513
##
## [[3]]
## [1]    2 2049
##
## [[4]]
## [1]    2 2049
```

7.2 CMU Faces Dataset

For this notebook, we'll use the CMU Faces dataset compiled by Tom Mitchell and his students way back in 1999.

```
# get paths to full-resolution images, regex"[[:alpha:]_]+.pgm"
# see: http://archive.ics.uci.edu/ml/machine-learning-databases/faces-mld/faces.data.html for image res
# prepare training and testing data, extract labels: left VS right
l <- "left"
r <- "right"
imgs_l <- list.files("data/faces",
                     pattern = paste0(l, "[[:alpha:]_]+.pgm"),
                     recursive = TRUE, full.names = TRUE)
imgs_r <- list.files("data/faces",
                     pattern = paste0(r, "[[:alpha:]_]+.pgm"),
                     recursive = TRUE, full.names = TRUE)

l_l <- length(imgs_l)
l_r <- length(imgs_r)

train_l_l <- ceiling(l_l / 2) #get balanced train and test set, split each class by half
train_l_r <- ceiling(l_r / 2)

trainIndex_l <- sample(l_l, train_l_l)
trainIndex_r <- sample(l_r, train_l_r)

train_df <- data.frame(Path = c(imgs_l[trainIndex_l], imgs_r[trainIndex_r]),
                       Label = c(rep(TRUE, train_l_l), rep(FALSE, train_l_r)),
                       stringsAsFactors = FALSE)

test_df <- data.frame(Path = c(imgs_l[-trainIndex_l], imgs_r[-trainIndex_r]),
                      Label = c(rep(TRUE, l_l-train_l_l), rep(FALSE, l_r-train_l_r)),
                      stringsAsFactors = FALSE)

train_df <- train_df[sample(nrow(train_df)),]
test_df <- test_df[sample(nrow(test_df)),]

lapply(list(train_df, test_df), dim)
```

```
## [[1]]
## [1] 157   2
##
## [[2]]
```

```
## [1] 155   2
```

7.3 On-the Fly Featurization

We can develop features on-the-fly and embed them into any of the `MicrosoftML` learners. This is especially useful if we want to train on data that is too large to fit in memory, so instead we work in batches.

```
mlTransform <- list(loadImage(vars = list(Image = "Path")),
                    resizeImage(vars = "Image",
                                width = 224, height = 224,
                                resizingOption = "IsoPad"),
                    extractPixels(vars = list(Pixels = "Image")),
                    featurizeImage(var = "Pixels", outVar = "Feature",
                                   dnnModel = "resnet101"))

model <- rxLogisticRegression(Label ~ Feature,
                              data = train_df,
                              mlTransforms = mlTransform, mlTransformVars = "Path")
```

```
## Automatically adding a MinMax normalization transform, use 'norm=Warn' or 'norm=No' to turn this behavior
## LBFGS multi-threading will attempt to load dataset into memory. In case of out-of-memory issues, turn off
## Warning: Too few instances to use 32 threads, decreasing to 1 thread(s)
## Beginning optimization
## num vars: 2049
## improvement criterion: Mean Improvement
## L1 regularization selected 144 of 2049 weights.
## Not training a calibrator because it is not needed.
## Elapsed time: 00:04:08.9797577
## Elapsed time: 00:00:00.0024762
```

```
summary(model)
```

```
## Call:
## rxLogisticRegression(formula = Label ~ Feature, data = train_df,
##     mlTransforms = mlTransform, mlTransformVars = "Path")
##
## LogisticRegression (BinaryClassifierTrainer) for: Label~Feature
## Data: train_df
##
##
## First 20 of 144 Non-zero Coefficients:
## (Bias): 0.2561404
## f765: 0.9757374
## f1789: -0.8494654
## f1752: -0.7844995
## f396: 0.6964862
## f1837: -0.6718948
## f1496: -0.614819
## f1173: 0.614722
## f530: -0.6073583
## f78: 0.5863273
## f851: 0.5443665
## f39: -0.5386707
## f1961: -0.5299587
```

```
## f1409: -0.4981045
## f529: 0.4914077
## f573: 0.4909203
## f619: -0.4871628
## f2011: 0.4459811
## f779: -0.4267508
## f596: 0.4123578
```

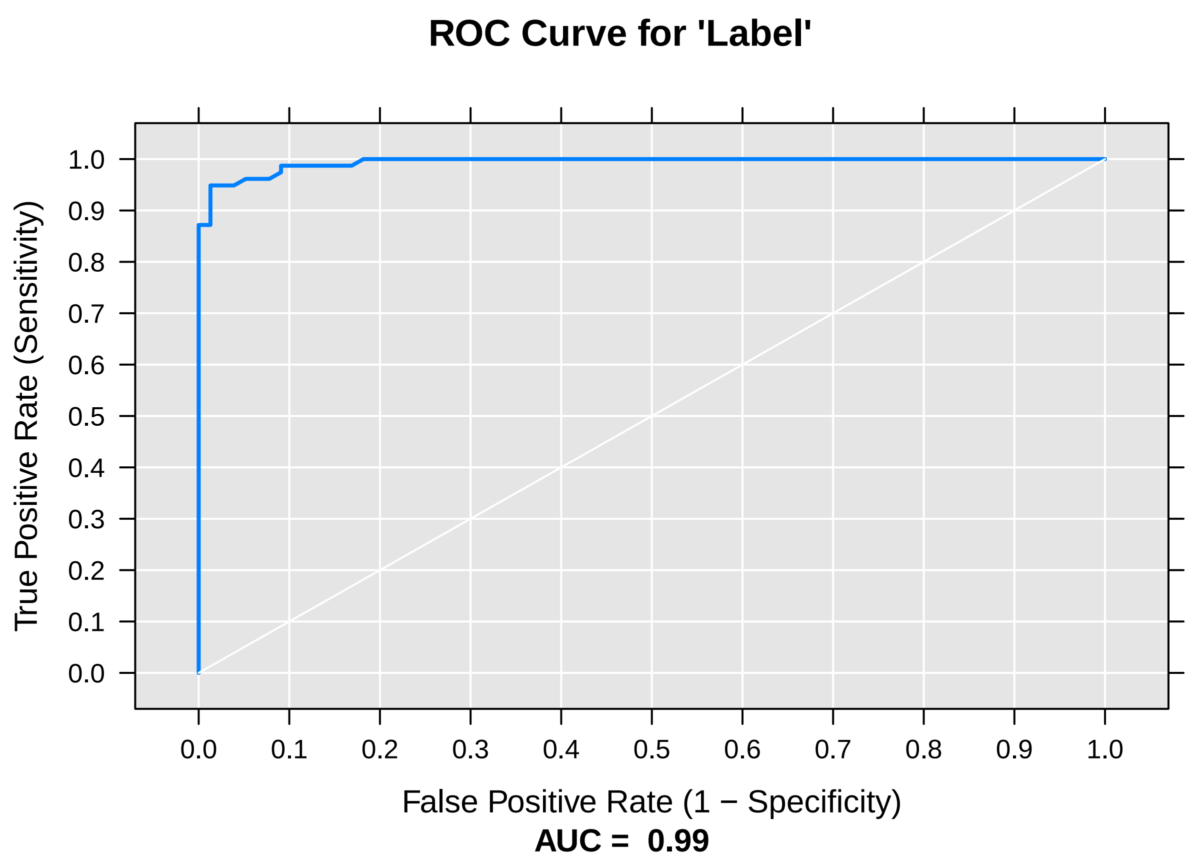
```r
score  <- rxPredict(model, test_df, extraVarsToWrite = "Label")
```

```
## Elapsed time: 00:04:03.4518699
```

```r
sum(score$Label==score$PredictedLabel)/nrow(score)
```

```
## [1] 0.9612903
```

```r
rxRocCurve("Label","Probability", score)
```



7.4 Retaining Features

While the above approach is scalable beyond datasets that can fit in memory, it has the drawback of not being reusable. In paricular, we can't "pull-out" the features we trained on our dataset for later use.

If you would like to retain the features you trained on, you can do so by using the `featurizeImage` function in MicrosoftML directly. It is analogous to the `mlTransforms` argumenet above.

```r
rxFeaturize(data = train_df,
            outData = "data/train.xdf",
```

```r
            overwrite = TRUE,
            mlTransforms = list(loadImage(vars = list(Image = "Path")),
                                resizeImage(vars = "Image",
                                            width = 224, height = 224,
                                            resizingOption = "IsoPad"),
                                extractPixels(vars = list(Pixels = "Image")),
                                featurizeImage(var = "Pixels",
                                               outVar = "Feature",
                                               dnnModel = "resnet18")),
            mlTransformVars = c("Path", "Label")) -> train_xdf
```

```
## Elapsed time: 00:00:22.1955525
```

```r
rxFeaturize(data = test_df, outData = "data/test.xdf", overwrite = TRUE,
            mlTransforms = list(loadImage(vars = list(Image = "Path")),
                                resizeImage(vars = "Image", width = 224, height = 224,
                                            resizingOption = "IsoPad"),
                                extractPixels(vars = list(Pixels = "Image")),
                                featurizeImage(var = "Pixels",
                                               outVar = "Feature",
                                               dnnModel = "resnet18")),
            mlTransformVars = c("Path", "Label")) -> test_xdf
```

```
## Elapsed time: 00:00:22.2479056
```

```r
varInfo <- rxGetVarInfo(train_xdf)

features <- paste("Feature", 0:511, sep=".", collapse = " + ")
form <- as.formula(paste("Label", features, sep="~"))
model <- rxLogisticRegression(formula = form, data = train_xdf)
```

```
## Automatically adding a MinMax normalization transform, use 'norm=Warn' or 'norm=No' to turn this behavior
## LBFGS multi-threading will attempt to load dataset into memory. In case of out-of-memory issues, turn off
## Warning: Too few instances to use 32 threads, decreasing to 1 thread(s)
## Beginning optimization
## num vars: 513
## improvement criterion: Mean Improvement
## L1 regularization selected 97 of 513 weights.
## Not training a calibrator because it is not needed.
## Elapsed time: 00:00:00.6830235
## Elapsed time: 00:00:00.0022631
```

```r
summary(model)
```

```
## Call:
## rxLogisticRegression(formula = form, data = train_xdf)
##
## LogisticRegression (BinaryClassifierTrainer) for: Label~Feature.0+Feature.1+Feature.2+Feature.3+Featur
## Data: train_xdf (RxXdfData Data Source)
## File name: data/train.xdf
##
##
## First 20 of 97 Non-zero Coefficients:
## (Bias): 0.2091156
## Feature.487: -1.327465
## Feature.431: 0.9092103
```

```
## Feature.460: -0.8586729
## Feature.157: 0.7992205
## Feature.47: -0.7986112
## Feature.511: -0.6897881
## Feature.315: -0.6197985
## Feature.270: 0.6049663
## Feature.233: -0.5948368
## Feature.62: 0.5754997
## Feature.456: 0.56534
## Feature.293: 0.5637971
## Feature.269: -0.5151423
## Feature.441: -0.493934
## Feature.88: 0.483808
## Feature.406: -0.4684003
## Feature.93: 0.465048
## Feature.145: -0.4564239
## Feature.224: -0.4489715
```

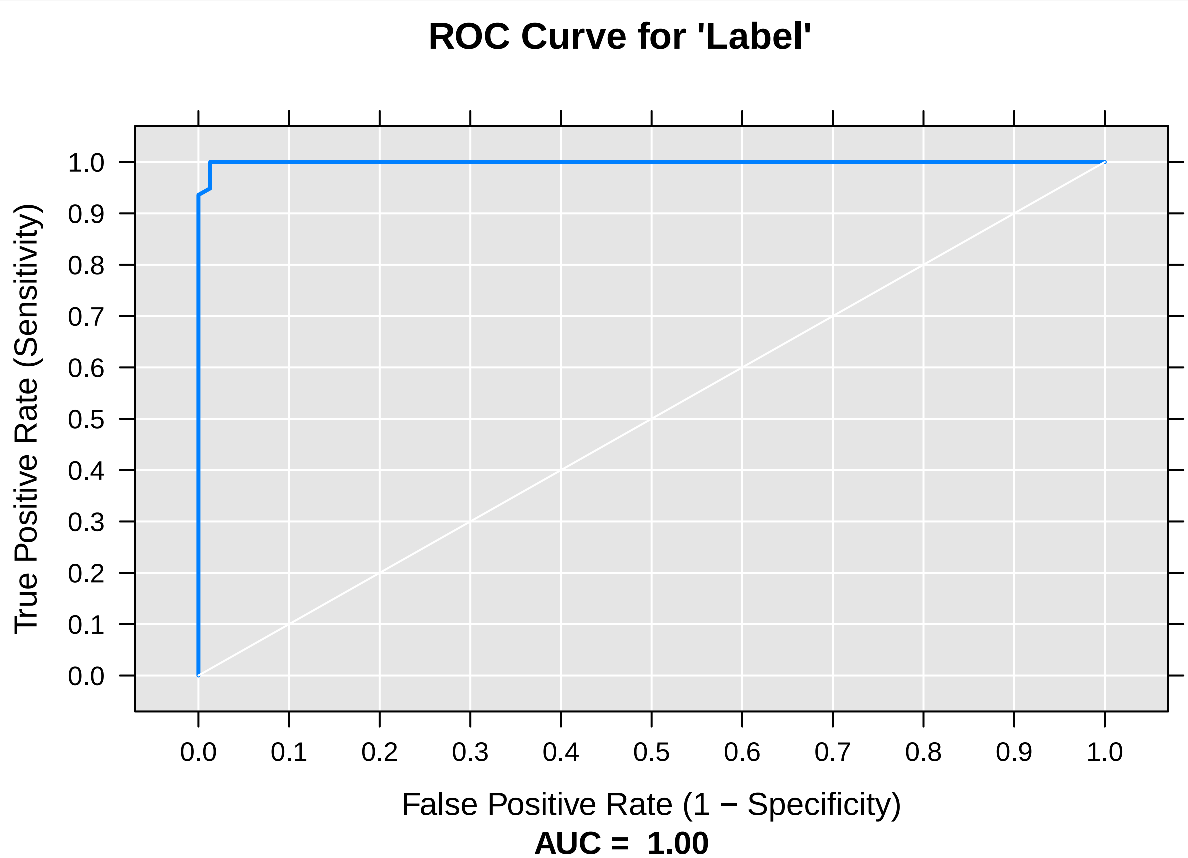
```r
score <-rxPredict(model, test_xdf, extraVarsToWrite = "Label")
```

```
## Elapsed time: 00:00:00.1706593
```

```r
sum(score$Label==score$PredictedLabel)/nrow(score)
```

```
## [1] 0.9870968
```

```r
rxRocCurve("Label","Probability",score)
```



Bibliography