Case Study: MNIST

Thursday, July 8, 2021 10:13 AM

```
library(tidyverse)
library(dslabs)
mnist <- read_mnist()
```

The dataset includes two components, a training set and test set:

```
names(mnist)
#> [1] "train" "test"
```

Each of these components includes a matrix with features in the columns:

```
dim(mnist$train$images)
#> [1] 60000 784
```

and vector with the classes as integers:

```
class(mnist$train$labels)
#> [1] "integer"
table(mnist$train$labels)
#>
#>    0    1    2    3    4    5    6    7    8    9
#> 5923 6742 5958 6131 5842 5421 5918 6265 5851 5949
```

Consider a subset of the dataset and sample 10,000 random rows from the training set and 1,000 random rows from the test set:

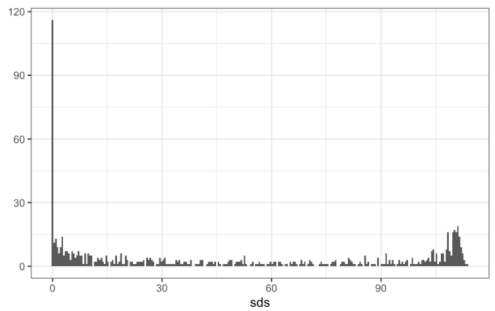
```
index <- sample(nrow(mnist$train$images), 10000)
x <- mnist$train$images[index,]
y <- factor(mnist$train$labels[index])

index <- sample(nrow(mnist$test$images), 1000)
x_test <- mnist$test$images[index,]
y_test <- factor(mnist$test$labels[index])
```

PREPROCESSING

Examples of preprocessing include standardizing the predictors, taking the log transform of some predictors, removing predictors that are highly correlated with others, and removing predictors with very few non-unique values or close to zero variation

Run the `nearZero` function from the **caret** package to see that several features do not vary much from observation to observation
I can see that there is a large number of features with 0 variability:

```
library(matrixStats)
sds <- colSds(x)
qplot(sds, bins = 256)
```



This is expected because there are parts of the image that rarely contain writing (dark pixels).

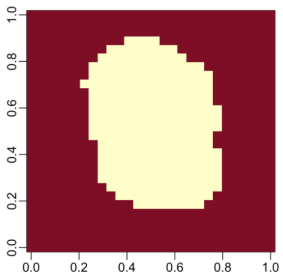
The **caret** packages includes a function that recommends features to be removed due to *near zero variance*:

```
library(caret)
nzv <- nearZeroVar(x)
```

```
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
 [18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34
 [35]  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51
 [52]  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68
 [69]  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
 [86]  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102
[103] 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119
[120] 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136
```

I can see the columns recommended for removal:

```
image(matrix(1:784 %in% nzv, 28, 28))
```



So I end up keeping this number of columns:

```
col_index <- setdiff(1:ncol(x), nzv)
length(col_index)
#> [1] 252
```

Add column names to the feature matrices as these are required by **caret**:

```
colnames(x) <- 1:ncol(mnist$train$images)
colnames(x_test) <- colnames(x)
```

MODEL FITTING

KNN

- The first step is to optimize for *k*
- Keep in mind that when I run the algorithm, I will have to compute a distance between each observation in the test set and each observation in the training set
- There are a lot of computations
- Therefore use k-fold cross validation to improve speed

```
control <- trainControl(method = "cv", number = 10, p = .9)
train_knn <- train(x[ ,col_index], y,
     method = "knn",
     tuneGrid = data.frame(k = c(3,5,7)),
       trControl = control)
```

```
train_knn
```

```
k-Nearest Neighbors

10000 samples
  252 predictor
   10 classes: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 9001, 8999, 8999, 9001, 9000, 8999, ...
Resampling results across tuning parameters:

  k  Accuracy   Kappa
  3  0.9466983  0.9407286
  5  0.9467986  0.9408387
  7  0.9444998  0.9382805

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was k = 5.
```

NOTE: In general, it is a good idea to try a test run with a subset of the data to get an idea of timing before i start running code that might take hours to complete. I can do this as follows:

```
n <- 1000
b <- 2
index <- sample(nrow(x), n)
control <- trainControl(method = "cv", number = b, p = .9)
train_knn <- train(x[index, col_index], y[index], method = "knn", tuneGrid =
data.frame(k = c(3,5,7)), trControl = control)
```

I can then increase n and b and try to establish a pattern of how they affect computing time to get an idea of how long the fitting process will take for larger values of n and b

Once I optimize the algorithm, fit it to the entire dataset:

```
fit_knn <- knn3(x[, col_index], y,  k = 3)
```

```
3-nearest neighbor model
Training set outcome distribution:

   0    1    2    3    4    5    6    7    8    9
 980 1121 1019 1048  961  889  984 1074  902 1022
```

I now achieve a high accuracy:

```
y_hat_knn <- predict(fit_knn, x_test[, col_index], type="class")

cm <- confusionMatrix(y_hat_knn, factor(y_test)) cm$overall["Accuracy"]
#> Accuracy
#> 0.953
```

From the specificity and sensitivity, we also see that 8s are the hardest to detect and the most commonly incorrectly predicted digit is 7

```
cm$byClass[,1:2]
#>          Sensitivity Specificity
#> Class: 0       0.990       0.996
#> Class: 1       1.000       0.993
#> Class: 2       0.965       0.997
#> Class: 3       0.950       0.999
#> Class: 4       0.930       0.997
#> Class: 5       0.921       0.993
#> Class: 6       0.977       0.996
#> Class: 7       0.956       0.989
#> Class: 8       0.887       0.999
#> Class: 9       0.951       0.990
```

Random Forest

- With random forest, computation time is a challenge
- For each forest, I need to build hundreds of trees
- I also have several parameters I can tune

- Because with random forest the fitting is the slowest part of the procedure rather than the predicting (as with kNN), use only five-fold cross validation
- Reduce the number of trees that are fit since I am not yet building my final model
- Finally, to compute on a smaller dataset, take a random sample of the observations when constructing each tree. Change this number with the `nSamp` argument.

```
library(randomForest)

control <- trainControl(method="cv", number = 5)
grid <- data.frame(mtry = c(1, 5, 10, 25, 50, 100))
```

```
train_rf <- train(x[, col_index], y,
     method = "rf",
     ntree = 150,
     trControl = control,
     tuneGrid = grid,
     nSamp = 5000)
```

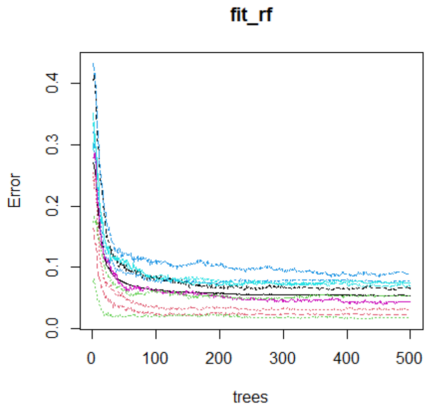
After opimizing fit the final model:

```
fit_rf <- randomForest(x[, col_index], y, minNode = train_rf$bestTune$mtry)
```

```
Call:
 randomForest(x = x[, col_index], y = y)
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 15

        OOB estimate of  error rate: 5.43%
Confusion matrix:
    0    1   2   3   4   5   6    7   8   9 class.error
0 958    0   4   3   1   2   5    0   7   0  0.02244898
1   0 1100   8   1   3   3   2    0   4   0  0.01873327
2   6    7 940   7  13   3   9   18  12   4  0.07752699
3   2    3  19 975   3  21   3    7   8   7  0.06965649
4   1    1   4   0 918   1   6    1   3  26  0.04474506
5   9    5   3  17   3 829  11    2   3   7  0.06749156
6   5    3   4   0   4  13 953    0   2   0  0.03150407
7   0    4  19   0  11   0   0 1017   4  19  0.05307263
8   3   12  11  18   4  10   8    1 821  14  0.08980044
9   6    2   7  16  21   2   0   14   8 946  0.07436399
```

To check that I ran enough trees use the plot function:

```
plot(fit_rf)
```



Predict and check accuracy:

```
y_hat_rf <- predict(fit_rf, x_test[ ,col_index])
```

```
> y_hat_rf
  1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
  1   8   2   1   5   8   6   9   9   0   2   1   3   8   7
 16  17  18  19  20  21  22  23  24  25  26  27  28  29  30
  4   7   1   7   3   6   1   2   5   8   3   0   8   4   1
```

```
cm <- confusionMatrix(y_hat_rf, y_test)
cm$overall["Accuracy"]
#> Accuracy
#> 0.956
```

VARIABLE IMPORTANCE

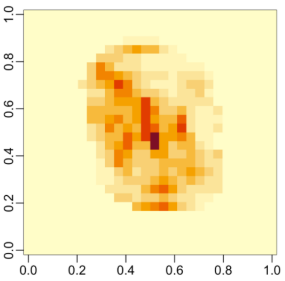
The following function computes the importance of each feature:

```
imp <- importance(fit_rf)
```

```
    MeanDecreaseGini
153         40.87177
154         54.10200
155         59.66756
156         61.78329
157         56.98353
158         32.17608
159         21.58664
```

I can see which features are being used most by plotting an image:

```
mat <- rep(0, ncol(x))
mat[col_index] <- imp
image(matrix(mat, 28, 28))
```



ENSEMBLES

- In machine learning, one can usually greatly improve the final results by combining the results of different algorithms
- Here I compute new class probabilities by taking the average of random forest and kNN
- The accuracy improves to 0.96

```
p_rf <- predict(fit_rf, x_test[,col_index], type = "prob")
```

	0	1	2	3	4	5	6	7	8	9
1	0.000	0.984	0.000	0.004	0.002	0.002	0.002	0.002	0.002	0.002
2	0.048	0.002	0.130	0.148	0.046	0.174	0.044	0.002	0.312	0.094
3	0.016	0.000	0.898	0.004	0.022	0.004	0.022	0.002	0.018	0.014
4	0.000	0.976	0.004	0.006	0.000	0.010	0.000	0.000	0.000	0.004
5	0.012	0.000	0.012	0.102	0.010	0.804	0.012	0.008	0.022	0.018

```
p_rf <- p_rf / rowSums(p_rf)
```

	0	1	2	3	4	5	6	7	8	9
1	0.000	0.984	0.000	0.004	0.002	0.002	0.002	0.002	0.002	0.002
2	0.048	0.002	0.130	0.148	0.046	0.174	0.044	0.002	0.312	0.094
3	0.016	0.000	0.898	0.004	0.022	0.004	0.022	0.002	0.018	0.014
4	0.000	0.976	0.004	0.006	0.000	0.010	0.000	0.000	0.000	0.004
5	0.012	0.000	0.012	0.102	0.010	0.804	0.012	0.008	0.022	0.018

```
p_knn  <- predict(fit_knn, x_test[,col_index])
> head(p_knn)
```

	0	1	2	3	4	5	6	7	8	9
[1,]	0	1	0	0.0000000	0	0.0000000	0	0	0.0000000	0
[2,]	0	0	0	0.3333333	0	0.0000000	0	0	0.6666667	0
[3,]	0	0	1	0.0000000	0	0.0000000	0	0	0.0000000	0
[4,]	0	1	0	0.0000000	0	0.0000000	0	0	0.0000000	0
[5,]	0	0	0	0.3333333	0	0.6666667	0	0	0.0000000	0
[6,]	0	0	0	0.0000000	0	0.0000000	0	0	1.0000000	0

```
p <- (p_rf + p_knn)/2
```

	0	1	2	3	4	5	6	7	8	9
1	0.000	0.992	0.000	0.0020000	0.001	0.0010000	0.001	0.001	0.0010000	0.001
2	0.024	0.001	0.065	0.2406667	0.023	0.0870000	0.022	0.001	0.4893333	0.047
3	0.008	0.000	0.949	0.0020000	0.011	0.0020000	0.011	0.001	0.0090000	0.007
4	0.000	0.988	0.002	0.0030000	0.000	0.0050000	0.000	0.000	0.0000000	0.002

```
y_pred <- factor(apply(p, 1, which.max)-1)
> head(y_pred, 20)
 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
 1  8  2  1  5  8  6  9  9  0  2  1  3  8  7  4  7  1  7  3
Levels: 0 1 2 3 4 5 6 7 8 9
```

```
confusionMatrix(y_pred, y_test)$overall["Accuracy"]
#> Accuracy
#> 0.961
```