MNIST using Random Trees

Samuel Velez Arango

05/05/2019

Introduction

The MNIST (Modified National Institute of Standards and Technology) database is a compilation of handwritten numbers. Each of the components in the data set include a matrix with features in the columns, which we will use to train a couple models to find which one performs best recognizing the handwritten numbers. Each entry is composed of 784 features, which represent each pixel in a 28x28 image, and their values depend on the "intensity" of the pixel, ranging between 0 to 255, 0 being white and 255 black.

The library is free to access through the dslabs package:

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

Because this default data set contains so many entries (more than my computer can handle), I will reduce it by taking 50.000 random samples to train our model and 10.000 to test it.

```
set.seed(1)
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])
```

Column names are added:

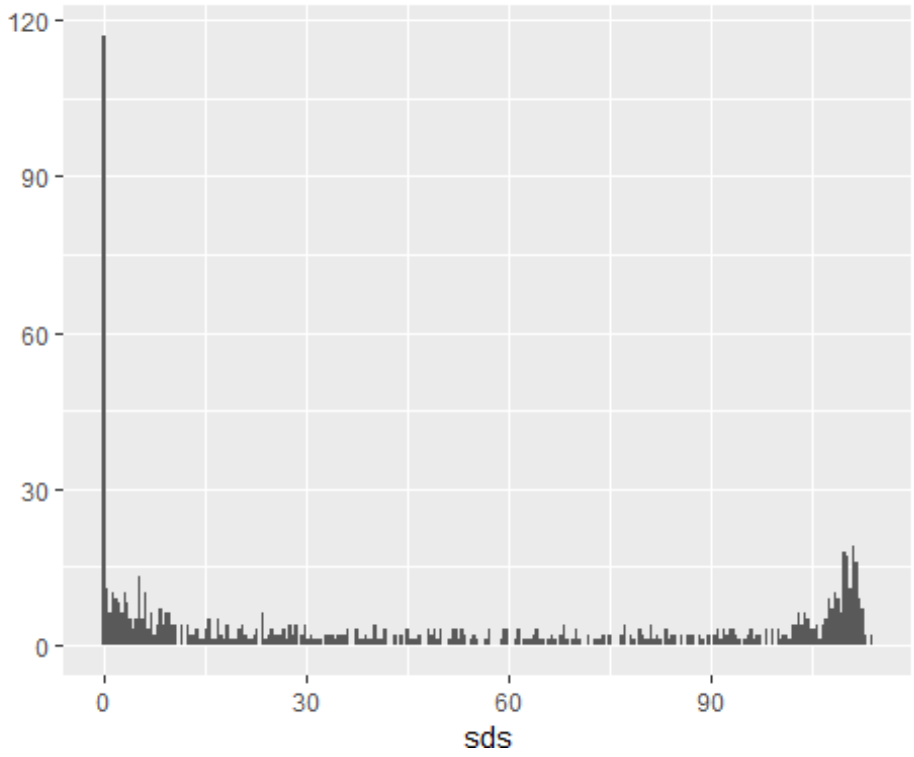
```
colnames(x) <- 1:ncol(mnist$train$images)
colnames(x_test) <- colnames(mnist$test$images)
```

Pre-processing

Since the digits are all centered, some spaces in the 28x28 images may go entirely unused, and these would act as useless features that could be removed. This will be tested by checking their variability, and those with zero or near zero variability will be deleted:

```
library(matrixStats)
library(ggplot2)
```

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

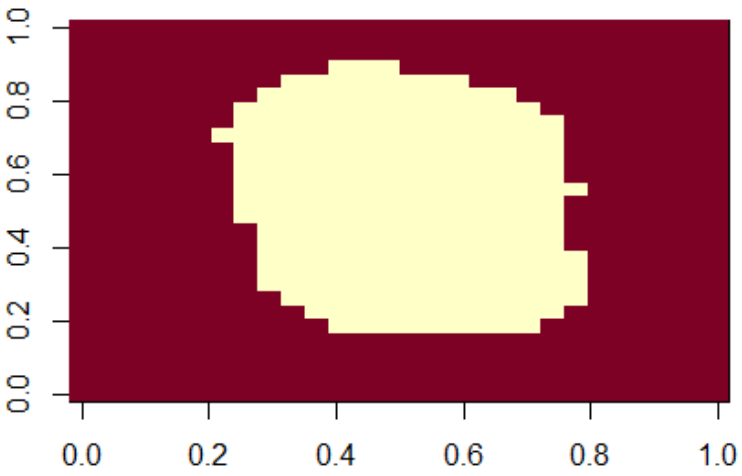


This plot shows that some parts of the image never change. The nearZeroVar function can recommend which features should be deleted, by calculating the frequency of the most common values in each feautre.

```
library(caret)

lowvariance <- nearZeroVar(x)
```

Plotting the features that can be removed (the yellow ones being the ones that will be kept):



This reduces our number of columns from 784 to 250

```
col_ind_x <- setdiff(1:ncol(x),lowvariance)
length(col_ind_x)

## [1] 250
```

Fitting models

The models that will be tested are k-Nearest Neighbors and Random Forest, as they are both relatively simple to fit, yet provide the necessary flexibility.

k-Nearest Neighbors

kNN models work by using the "k" nearest data points to the observed one to estimate its conditional probability. The advantage of applying k Nearest Neighbors is how responsive it is to local structures. The goal with kNN models is to optimize k to achieve maximum accuracy.

The following code will find the k that provides the best accuracy, and we also employ the k-fold cross validation method with the trainControl function, which will fold our data in 10, and use 10% of each to test itself.

```
control_set <- trainControl(method = "cv", number = 10, p = .9)
```

And now we train the model trying out k from 1 to 9 by 2.

```
train_knn <- train(x[,col_ind_x], y,
                   method = "knn",
                   tuneGrid = data.frame(k = seq(1,9,2)),
                   trControl = control_set)
train_knn

## k-Nearest Neighbors
##
## 10000 samples
##   250 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: 9002, 8999, 9002, 8998, 8999, 8999, ...
## Resampling results across tuning parameters:
##
##   k  Accuracy   Kappa
##   1  0.9480011  0.9421929
##   3  0.9487016  0.9429699
##   5  0.9469006  0.9409664
##   7  0.9447010  0.9385199
##   9  0.9427010  0.9362940
```

```
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was k = 3.
```

These results indicate the optimal k, thus it will be inserted to fit the model:

```
model_knn <- knn3(x[, col_ind_x], y,  k = train_knn$bestTune)
```

And now we will apply it to testing data set and get a good conclusion on its performance:

```
y_hat_knn <- predict(model_knn, x_test[, col_ind_x], type="class")
confusionmatrix <- confusionMatrix(y_hat_knn, factor(y_test))
confusionmatrix$overall["Accuracy"]

## Accuracy
##    0.952
```

The optimized number of k-neighbors provides an accuracy of 95.2% in the test subset.

```
confusionmatrix$byClass[,1:2]

##          Sensitivity Specificity
## Class: 0   0.9708738   0.9944259
## Class: 1   1.0000000   0.9943439
## Class: 2   0.9056604   0.9977629
## Class: 3   0.9545455   0.9912281
## Class: 4   0.9292929   1.0000000
## Class: 5   0.9666667   0.9934066
## Class: 6   0.9767442   0.9912473
## Class: 7   0.9611650   0.9910814
## Class: 8   0.8979592   0.9988914
## Class: 9   0.9549550   0.9943757
```

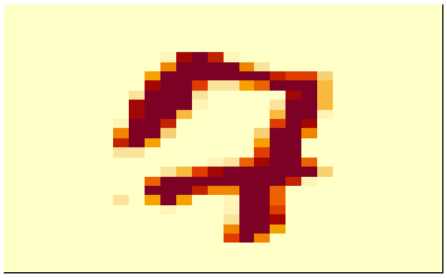
Closer inspection of each class reveals that the number with the lowest sensitivity (the one that is most often guessed incorrectly) is 8 with the lowest true positive rate.

Here are a few examples of our mistakes:

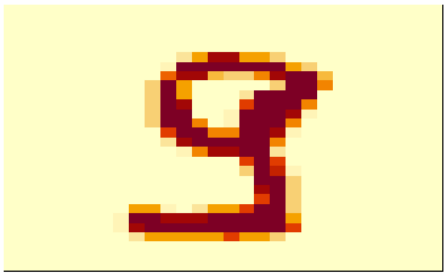
```
p_max <- predict(model_knn, x_test[,col_ind_x])
p_max <- apply(p_max, 1, max)
ind <- which(y_hat_knn != y_test)
ind <- ind[order(p_max[ind], decreasing = TRUE)]
#image(matrix(ind, 28, 28))

for(i in ind[1:6]){
  image(matrix(x_test[i,], 28, 28)[, 28:1],
        main = paste0("We guessed Pr(",y_hat_knn[i],")=",round(p_max[i],
2), " but it is a ",y_test[i]),
        xaxt="n", yaxt="n")
}
```

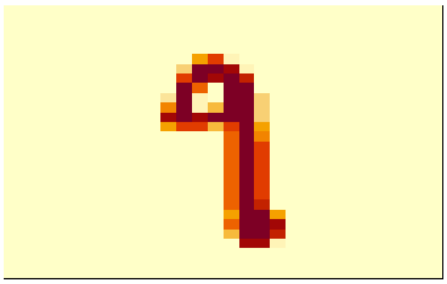
We guessed Pr(9)=1 but it is a 7



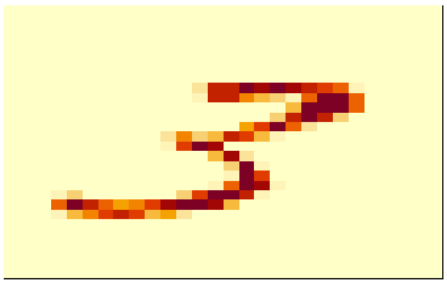
We guessed Pr(3)=1 but it is a 9



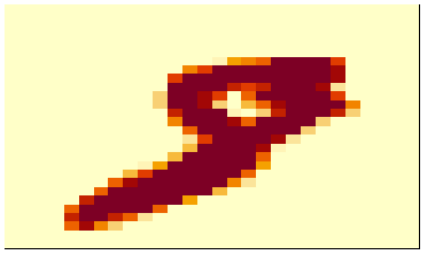
We guessed Pr(1)=1 but it is a 9



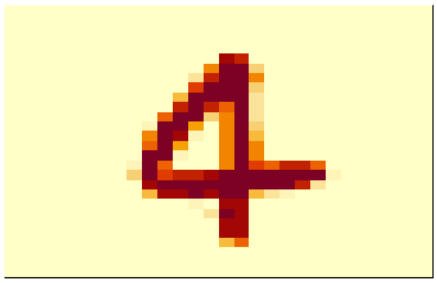
We guessed Pr(5)=1 but it is a 3



We guessed Pr(7)=1 but it is a 8



We guessed Pr(6)=1 but it is a 4



Random Forests

Random Forest models create a certain amount of decision trees and then creates an ensemble of them. Each tree in each forest is subject to several parameters that can be tuned to maximize precision. Using the Rborist package's Random Trees, the parameters that will be optimized are the number of randomly selected predictors (predFixed) to split the "branches" of the trees, and minimal node size (minNode), which defines how many data points have to be in a node to create a "branch". The random number of predictors ensures the trees will not be correlated, increasing precision. We wil test values between 10 and 50 skipping by 5. Meanwhile the minimal node size defines how deep the tree will be. This last parameter will be fixed at one, to avoid calculating an optimal one yet letting the tree run as deep as it needs to. When building each tree, a random subset of observations will be taken using the nSamp function. The cross validation method will be used again, but this time folding it only 5

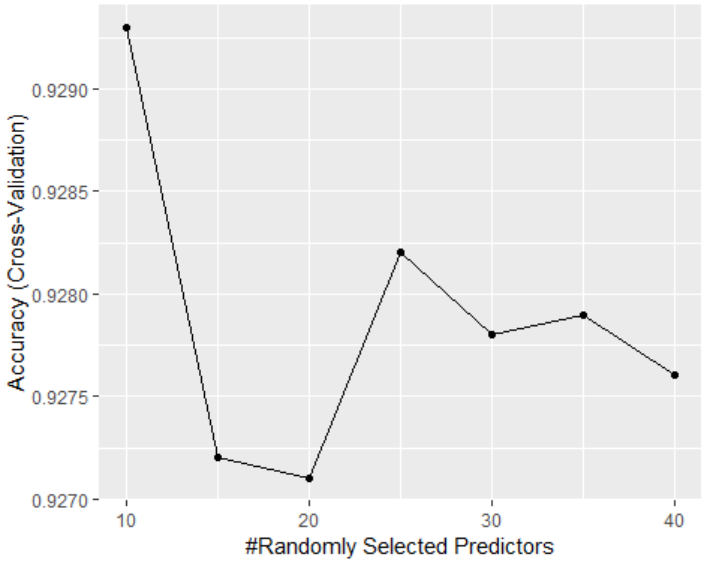
times and with only 50 trees, as processing requirements are more demanding for this model than for kNN.

```
library(Rborist)

control_set <- trainControl(method="cv", number = 5, p = 0.8)
tunegrid <- expand.grid(minNode = c(1) , predFixed = seq(10, 40, 5))

train_randforest <-  train(x[, col_ind_x],
                    y,
                    method = "Rborist",
                    nTree = 50,
                    trControl = control_set,
                    tuneGrid = tunegrid,
                    nSamp = 5000)

ggplot(train_randforest)
```




```
train_randforest$bestTune

##   predFixed minNode
## 1        10       1
```

This shows the optimal number of random predictors while the minimal node size is fixed at 1. Knowing this, the number of trees can be increased to create the actual model.

```
model_randforest <- Rborist(x[ ,col_ind_x], y,
                    nTree = 1000,
                    minNode = train_randforest$bestTune$minNode,
                    predFixed = train_randforest$bestTune$predFixed)

y_hat_rf <- factor(levels(y)[predict(model_randforest, x_test[ ,col_ind_x
])$yPred])
```

```
confusionmatrix <- confusionMatrix(y_hat_rf, y_test)
confusionmatrix$overall["Accuracy"]
```

```
## Accuracy
##    0.946
```

The model provides an accuracy of 0.946, slightly worse than our previous knn result.

```
confusionmatrix$byClass[,1:2]
```

```
##          Sensitivity Specificity
## Class: 0   0.9902913   0.9944259
## Class: 1   1.0000000   0.9977376
## Class: 2   0.8773585   0.9921700
## Class: 3   0.9318182   0.9956140
## Class: 4   0.9292929   0.9944506
## Class: 5   0.9777778   0.9934066
## Class: 6   0.9651163   0.9934354
## Class: 7   0.9126214   0.9955407
## Class: 8   0.9387755   0.9922395
## Class: 9   0.9369369   0.9910011
```

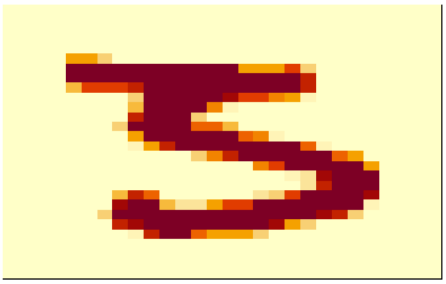
Meanwhile, the most problematic number is now 2. This means that the performance of Random Forests was quite similar to that of kNN. To conclude with these two results, the flexibility in Random Forests should give it an edge in this challenge, but shouldn't be too far from knn, which also had a good performance. I suspect that had its processing not being restricted so much, Random Forest could provide an even better result.

These are some of our mistakes:

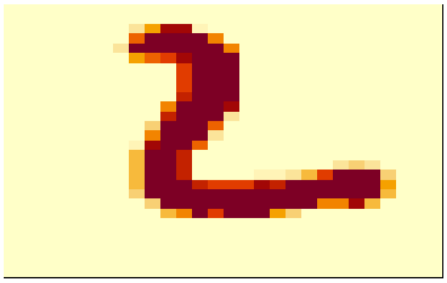
```
p_max <- predict(model_randforest, x_test[,col_ind_x])$census
p_max <- p_max / rowSums(p_max)
p_max <- apply(p_max, 1, max)
ind  <- which(y_hat_rf != y_test)
ind <- ind[order(p_max[ind], decreasing = TRUE)]

for(i in ind[1:6]){
     image(matrix(x_test[i,], 28, 28)[, 28:1],
           main = paste0("We guessed Pr(",y_hat_rf[i],")=",round(p_max[i]
, 2), " but it is a ",y_test[i]),
           xaxt="n", yaxt="n")
}
```

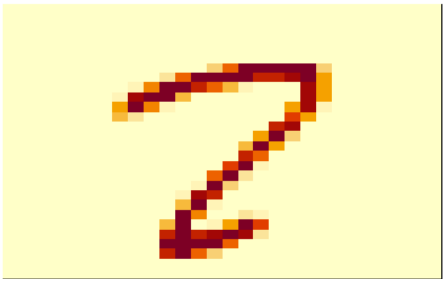
We guessed Pr(5)=0.76 but it is a 3



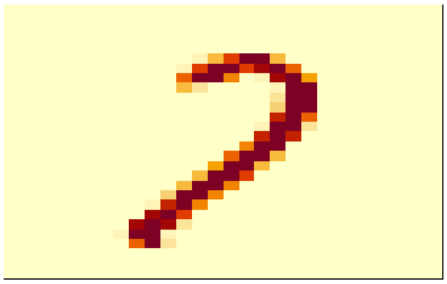
We guessed Pr(6)=0.7 but it is a 2



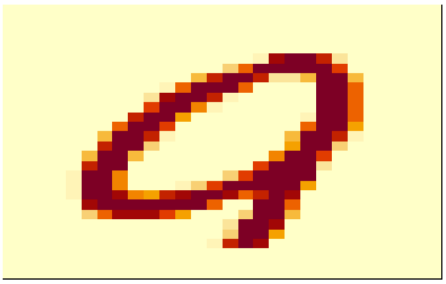
We guessed Pr(7)=0.62 but it is a 2



We guessed Pr(2)=0.61 but it is a 7



We guessed Pr(0)=0.55 but it is a 9



We guessed Pr(6)=0.5 but it is a 8

Ensemble

We can make an attempt to improve on our previous results by combining the models, which can be done by simply taking the average of the class probabilities for each prediction:

```
p_randforest <- predict(model_randforest, x_test[,col_ind_x])$census
p_randforest <- p_randforest / rowSums(p_randforest)
p_knn  <- predict(model_knn, x_test[,col_ind_x])
p <- (p_randforest + p_knn)/2
y_pred <- factor(apply(p, 1, which.max)-1)
```

```r
confusionmatrix <- confusionMatrix(y_pred, y_test)
confusionmatrix$overall["Accuracy"]
```

```
## Accuracy
##    0.956
```

```r
confusionmatrix$byClass[,1:2]
```

```
##          Sensitivity Specificity
## Class: 0   0.9805825   0.9955407
## Class: 1   1.0000000   0.9943439
## Class: 2   0.9056604   0.9977629
## Class: 3   0.9545455   0.9912281
## Class: 4   0.9393939   1.0000000
## Class: 5   0.9666667   0.9945055
## Class: 6   0.9767442   0.9912473
## Class: 7   0.9611650   0.9933110
## Class: 8   0.9081633   0.9988914
## Class: 9   0.9639640   0.9943757
```

Doing this pushes the accuracy into 95.6%, indeed improving our result over the previous ones.

While we attained a relatively high accuracy, the current best approaches hover around 97%, though they use much more advanced methods. Our most erroneous guess were the 5 and 8 digits through all approaches, improving their accuracy will be an interesting challenge.