Data Modeling with Caret

Kyle Joecken

November 14, 2015

Part 1: Machine Learning in R Without `caret`

A simple machine learning example that does not take advantage of `caret`

First, we load the required libraries and read our data into data frames. The following data (and most of the code) come from kaggle's introductory "Digit Recognizer" optical character recognition problem.

```
library(randomForest)
trainData <- read.csv("data/digits/train.csv")
trainData$label <- as.factor(trainData$label)
testData <- read.csv("data/digits/test.csv")
```

Let's verify that the data look properly loaded, and inspect the format.

```
dim(trainData); dim(testData); str(trainData[, 1:6])
```

```
## [1] 42000   785
```

```
## [1] 28000   784
```

```
## 'data.frame':    42000 obs. of  6 variables:
##  $ label : Factor w/ 10 levels "0","1","2","3",..: 2 1 2 5 1 1 8 4 6 4 ...
##  $ pixel0: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel1: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel2: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel3: int  0 0 0 0 0 0 0 0 0 0 ...
##  $ pixel4: int  0 0 0 0 0 0 0 0 0 0 ...
```

We see that the first column (`label`) is the actual digit represented by an image, and the remaining 784 columns are greyscale values for each pixel in the 28x28 images.

After setting the seed to keep our results repeatable, we select a random subsample of our data and pull the predictors from the outcome.

```
set.seed(0)
numTrain <- 10000
rows <- sample(1:nrow(trainData), numTrain)
labels <- as.factor(trainData[rows,1])
subtrain <- trainData[rows,-1]
```

Finally, we build a random forest model on the subset of the `train` data set, computing the predicted outcomes of the test set along the way. The actual `randomForest` function is called with four arguments, though only the first is necessary.

```
randomForest(x, y=NULL, xtest=NULL, ntree=500)
```

There are over a dozen additional potential parameters to pass, including `mtry`, the number of predictors to randomly sample at each break point.

```
numTrees <- 25
rf <- randomForest(subtrain, labels, xtest=testData, ntree=numTrees)
predictions <- data.frame(
    ImageId=1:nrow(testData),
    Label=levels(labels)[rf$test$predicted]
)
head(predictions)
```

```
##   ImageId Label
## 1       1     2
## 2       2     0
## 3       3     9
## 4       4     4
## 5       5     2
## 6       6     7
```

This went rather smoothly. But:

- What if I want to reserve my own data set for validation before predicting on the test set?
- What if I want further details on factor selection done by the model?
- What if I simply want to try a different model?

`caret` helps will all of these things and more.

Part 2: Data and Model Exploration in `caret`

Variable importance and parameter tuning

Let's improve upon kaggle's example model by applying some of `caret`'s functionality. We begin by loading the `caret` package. We will simultaneously load a parallel processing package `doMC` and tell it how many cores we're rocking (the Mac on which I wrote this has four cores with two threads each). For those packages that implement some form of parallelization, `caret` does not interfere. `randomForest` is definitely one of those packages.

See the `caret` documentation for additional information.

```
library(caret)
library(doMC)
registerDoMC(8)
```

`createDataPartition`

The first function we will want to learn is `caret`'s data partitioning function. Here is the function call from the documentation:

```
createDataPartition(
    y,
    times = 1,
    p = 0.5,
    list = TRUE,
    groups = min(5, length(y))
)
```

This function takes `times` samples from your data vector `y` of proportion `p`. If your data are discrete, `createDataPartition` will automatically take a representative sample from each level as best as it can; otherwise, you can use `groups` to help `caret` partition a continuous variable.

The values returned are chosen indices from `y`.

train

This function trains your model. Again from the slimmed down docs:

```
train(
    x,
    y,
    method = "rf",
    ...
)
```

This call returns a `train` object, which is basically a list. The model contained is built applying the given `method` (in this case `"rf"` means random forest) to the predictors in the data frame `x` and with associated vector of outcomes `y`. As `caret` is really just a wrapper for the underlying packages that deploy varous methods, we can pass additional arguments through the ellipses as needed.

Let's have a look at an example. These lines are nearly identical to those from kaggle's "benchmark" code. A few things are different:

- I want to plot soon, so I reduced from a sample of 10,000 to one of about 1,000
- I asked `randomForest` to keep track of importance variables, which it does not do by default

You can see that we pass `list=FALSE` to `createDataPartition`; as we only have one sample, we'd like to have our row numbers in a vector so that we can easily subset our data with it. We also used the formula implementation of the `train` function rather than slice the data frame via `train(naiveData[, -1], naive$label, ...)`.

```
set.seed(0)
inTrain <- createDataPartition(trainData$label, p=1/40, list=FALSE)
naiveData <- trainData[inTrain, ]
naiveModel <- train(
    label ~ .,
    data = naiveData,
    method="rf",
    ntree=25,
    importance=TRUE
)
```

varImp

Since we've asked `randomForest` to keep track of importance, let's have a look at it. The `varImp` function computes importance on a scale from 0 to 100 (by default–set `scale=FALSE` to return the raw score used).

```
varImp(naiveModel)
```

```
## rf variable importance
##
##   variables are sorted by maximum importance across the classes
##   only 20 most important variables shown (out of 784)
##
##              0     1     2     3     4     5      6     7     8     9
## pixel541 61.52 64.81 64.07 13.23 72.30 35.56 100.00 59.12 51.84 66.12
## pixel378 82.61 98.46 84.90 88.62 77.01 58.12  61.93 82.00 74.71 68.00
## pixel318 59.12 86.95 82.19 94.40 62.16 64.72  85.19 76.38 34.27 68.28
## pixel515 59.12 69.80 73.71 76.49 48.60 66.84  92.61 45.11 53.24 78.41
## pixel598 67.47 59.12 62.48 67.49 70.48 59.12  34.97 67.26 74.25 91.30
## pixel430 63.26 72.91 57.07 42.38 89.54 64.20  70.00 79.75 59.12 74.07
## pixel290 58.78 81.45 59.12 89.32 21.37 42.88  59.12 50.41 58.28 74.77
## pixel292 72.61 62.05 86.97 63.86 51.44 40.57  52.25 37.19 50.69 63.69
## pixel181 43.79 59.12 45.78 48.01 86.27 49.53  63.57 79.18 25.34 52.70
## pixel571 67.50 40.24 68.22 51.51 70.20 21.37  86.12 52.71 71.37 56.82
## pixel210 59.12 63.00 13.15 57.05 85.81 67.51  80.77 58.51 43.88 75.57
## pixel239 59.12 66.86 28.99 13.31 73.31 24.20  76.92 70.71 85.81 43.44
## pixel205 59.12 85.22 76.80 71.36 13.00 40.24  47.93 67.17 50.17 65.50
## pixel345 52.69 70.57 59.12 85.15 81.31 67.91  71.98 61.08 41.78 64.93
## pixel346 68.41 59.12 67.19 35.88 84.92 66.33  64.59 61.17 67.00 61.12
## pixel183 41.52 59.12 59.12 59.12 72.69 84.44  21.37 72.09 59.12 62.63
## pixel406 69.33 84.25 62.54 35.23 44.88 52.74  59.12 78.32 71.59 33.57
## pixel403 59.12 84.25 64.01 60.60 22.15 73.38  59.25 61.14 40.91 69.38
## pixel405 82.24 77.73 67.10 57.99 59.61 31.07  21.37 84.19 81.21 53.45
## pixel408 67.90 40.24 38.61 67.83 67.94 84.18  47.64 40.24 21.37 62.63
```

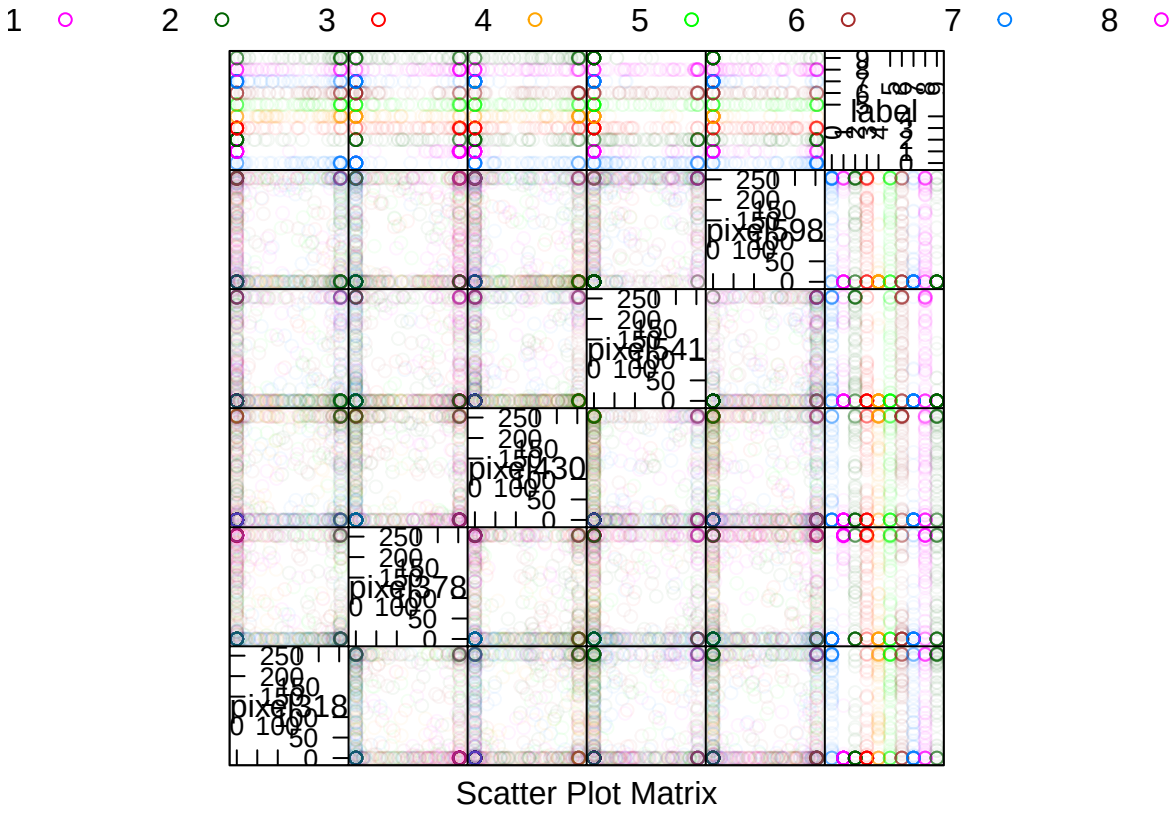
featurePlot

A wrapper for various `lattice` plots. Once more, the call string from documentation:

```
featurePlot(
    x,
    y,
    plot = if(is.factor(y)) "strip" else "scatter",
    ...
)
```

As before, `x` holds the predictor data frame and `y` holds the outcome vector. `plot` is a string corresponding to the type of plot you want (e.g., `"pairs"`). `...` implies that you can add additional arguments to be passed down to the `lattice` plot call.

```
featurePlot(
    x = naiveData[, c(320, 380, 432, 543, 600, 1)],
    y = naiveData$label,
    plot = "pairs",
    alpha = 1/20,
    auto.key = list(columns = 10)
)
```



Scatter Plot Matrix

train(tuneGrid)

As an optional argument to pass to `train`, `tuneGrid` allows you to pass in various combinations of hyperparameters to your model in an effort to optimize them. The `caret` documentation has a nice example that demonstrates how you make a simple matrix of hyperparameter combinations, save it as a named matrix, and pass that in as the `tuneGrid` argument.

If you want to know what hyperparameters a particular method takes, simply call the `modelLookup` function (e.g., `modelLookup("rf")`). What returns will be a printout of each hyperparameter by name, description, and some indicators of its intended use. For additional details, you'll need to check the documentation of the underlying package.

Note: You must name your tuning grid! `caret` will get angry if you try to pass in a call to `expand.grid`.

For `randomForest` (`method="rf"`), there is only one hyperparameter: `mtry`. This tells `randomForest` how many of the predictors to try and split on at each node. By default, `randomForest` takes a random sample of the square root of the total and tries to split on those. In our case, 28 x 28 = 784 pixels means the default is 28 pixels chosen at each split. But what if that isn't best?

```
set.seed(12345)
inTrain <- createDataPartition(trainData$label, p=0.5, list=FALSE)
fitGrid <- expand.grid(
    mtry = (1:8) * 10 - 2
)
rfModel <- train(
    label ~ .,
    data = trainData[inTrain, ],
    method="rf",
```

```
    tuneGrid=fitGrid,
    ntree=25
)
```

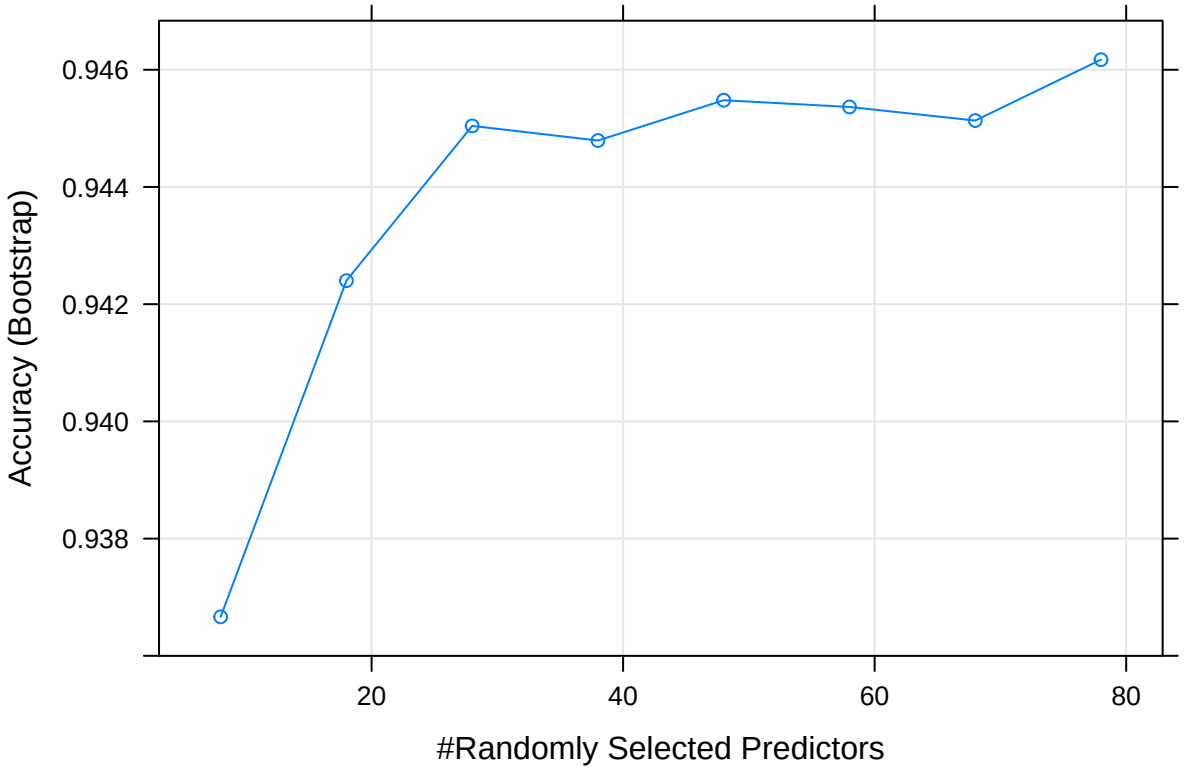
If you ask `R` to print the `train` object, it outputs a nice summary that includes (within reason) a list of the parameter combinations and the resulting 'quality' metrics (these can be changed).

```
print(rfModel)
```

```
## Random Forest
##
## 21003 samples
##   784 predictor
##    10 classes: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 21003, 21003, 21003, 21003, 21003, 21003, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa      Accuracy SD  Kappa SD
##    8    0.9366648  0.9295969  0.003284314  0.003650292
##   18    0.9424012  0.9359757  0.002832008  0.003145199
##   28    0.9450417  0.9389111  0.002242495  0.002492187
##   38    0.9447924  0.9386345  0.002373526  0.002635873
##   48    0.9454799  0.9393983  0.002690703  0.002990869
##   58    0.9453652  0.9392713  0.002634015  0.002927135
##   68    0.9451333  0.9390135  0.002898686  0.003219813
##   78    0.9461718  0.9401678  0.002299291  0.002553780
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was mtry = 78.
```

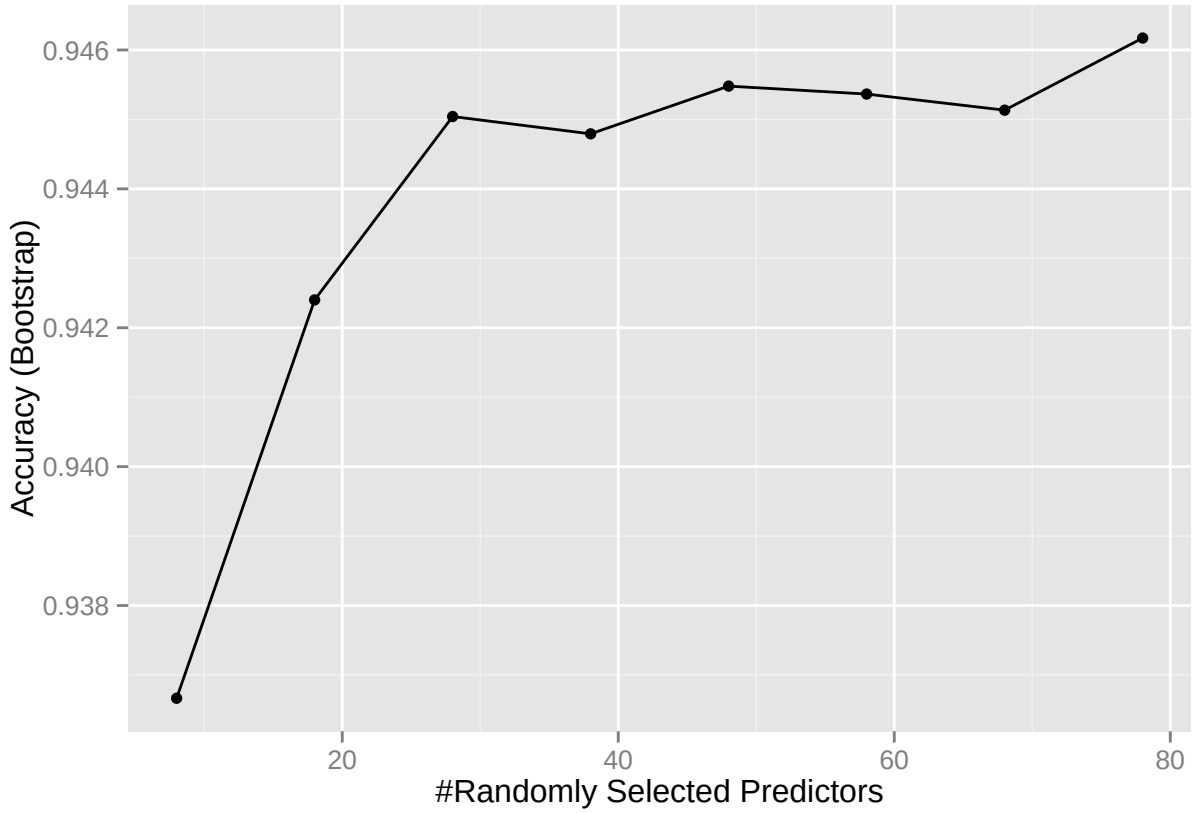
Similarly, if you plot a `train` object, you get a graph of your metric against your hyperparameter(s).

```
plot(rfModel)
```



Do you like `ggplot2`? So does `caret`!

```
ggplot(rfModel)
```



Part 3: Model Validation

Tuning and performance

In this final section, we'd like to look at a few tools that can help validate and analyze your models. For a classification task, the most obvious such tool is the confusion matrix. Perhaps unsurprisingly, `caret` has an aptly-named helper function.

confusionMatrix.train

`caret`'s `confusionMatrix` function has two iterations, and the first applies to a `train` object. Assuming that the outcomes of the method call were explicitly discrete, calling `confusionMatrix(myModel)` will return a simple diagram that shows how frequently each level was guessed correctly or confused for a different level. This is simply a finer level of detail on the accuracy score we've already been shown by printing the `train` object directly.

```
confusionMatrix(rfModel)
```

```
## Bootstrapped (25 reps) Confusion Matrix
##
## (entries are percentages of table totals)
##
##           Reference
## Prediction    0    1    2    3    4    5    6    7    8    9
##          0  9.6  0.0  0.1  0.0  0.0  0.1  0.1  0.0  0.0  0.1
##          1  0.0 10.9  0.0  0.0  0.0  0.0  0.0  0.1  0.1  0.0
##          2  0.0  0.1  9.4  0.2  0.0  0.0  0.0  0.1  0.1  0.0
##          3  0.0  0.0  0.1  9.5  0.0  0.2  0.0  0.0  0.2  0.1
##          4  0.0  0.0  0.1  0.0  9.2  0.0  0.0  0.1  0.1  0.2
##          5  0.0  0.0  0.0  0.2  0.0  8.4  0.1  0.0  0.1  0.1
##          6  0.1  0.0  0.1  0.0  0.1  0.1  9.7  0.0  0.1  0.0
##          7  0.0  0.0  0.1  0.1  0.0  0.0  0.0  9.9  0.0  0.1
##          8  0.1  0.0  0.1  0.1  0.1  0.1  0.1  0.0  8.9  0.1
##          9  0.0  0.0  0.0  0.1  0.3  0.1  0.0  0.2  0.2  9.2
```

predict

What if we want to know how the model performs on data it wasn't trained on? We'll need to apply it to other data we've been holding back (the point of `createDataPartition`), and compare that to truth values for those data. Once again to the docs:

```
predict(
    object,
    newdata = NULL,
    ...
)
```

Here, `object` is the `train` object we're using to predict, and `newdata` is a data frame containing the withheld data. As with other `caret` functions, this is essentially a wrapper for the prediction functions of the various packages, so additional arguments are ocassionally necessary and can be passed through the ellipsis.

```
rfValidData <- predict(rfModel, trainData[-inTrain, ])
```

confusionMatrix

Now that we have used our model to predict the outcomes for new data, we'll want to compare that to the known truth values. This is the other (perhaps more useful) version of `confuseMatrix`. As usual, the docs:

```
confusionMatrix(
    data,
    reference,
    ...
)
```

Here, `data` is a vector of newly predicted data and `reference` are the truth values.

```
confusionMatrix(rfValidData, trainData[-inTrain, "label"])
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1    2    3    4    5    6    7    8    9
##          0 2020    0   11   12    2    8   15    1    5   12
##          1    0 2295    4    4    4    6    3   10   23    6
##          2    3   14 1987   33    5    6    2   43    5    6
##          3    3    7   11 2009    2   42    0    4   24   33
##          4    1    3   11    3 1949    4    5   17    8   42
##          5    7    4    4   45    4 1782   21    2   21   14
##          6   12    3    8    3    6   18 2002    0   13    4
##          7    3    7   22   18    2    4    1 2092    5   20
##          8   15    6   25   35    9   18   18    4 1902   13
##          9    2    3    5   13   53    9    1   27   25 1944
##
## Overall Statistics
##
##                Accuracy : 0.9517
##                  95% CI : (0.9487, 0.9545)
##     No Information Rate : 0.1115
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9463
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity           0.97773   0.9799  0.95163  0.92368  0.95727  0.93938
## Specificity           0.99651   0.9968  0.99381  0.99331  0.99504  0.99361
## Pos Pred Value        0.96836   0.9745  0.94439  0.94098  0.95399  0.93592
## Neg Pred Value        0.99757   0.9975  0.99465  0.99120  0.99541  0.99398
## Prevalence            0.09840   0.1115  0.09944  0.10359  0.09697  0.09035
## Detection Rate        0.09620   0.1093  0.09463  0.09568  0.09282  0.08487
```

```
## Detection Prevalence  0.09935   0.1122  0.10020  0.10168  0.09730  0.09068
## Balanced Accuracy     0.98712   0.9884  0.97272  0.95849  0.97616  0.96650
##                      Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity           0.96809  0.95091  0.93648  0.92837
## Specificity           0.99646  0.99564  0.99246  0.99270
## Pos Pred Value        0.96762  0.96228  0.93007  0.93372
## Neg Pred Value        0.99651  0.99426  0.99319  0.99207
## Prevalence            0.09849  0.10478  0.09673  0.09973
## Detection Rate        0.09535  0.09963  0.09058  0.09258
## Detection Prevalence  0.09854  0.10354  0.09739  0.09916
## Balanced Accuracy     0.98227  0.97327  0.96447  0.96053
```

trainControl

This is going rather well, but we have not yet considered how the model is validating itself as it trains. By default, `caret` uses boostrap resampling; this can be changed, however. Much like `tuneGrid`, there is a `trControl` argument to the `train` function that takes the output of the `trainControl` function (as documented here):

```
trainControl(
    method = "boot",
    number = ifelse(method %in% c("cv", "repeatedcv"), 10, 25),
    repeats = ifelse(method %in% c("cv", "repeatedcv"), 1, number,
    ...
)
```

You can set `method` to be a string like `"repeatedcv"` to change the resampling method, and pass additional parameters that suit your method. I've mentioned the ones that have to do with repeated cross-validation, but there are many others in the docs if you are interested.

```
set.seed(2967)
inTrain <- createDataPartition(trainData$label, p = 0.5, list = FALSE)
fitControl <- trainControl(
    method = "repeatedcv",
    number = 5,
    repeats = 3
)
# hyperparameters must be passed through the tuneGrid argument, even if constant
fitGrid <- expand.grid(mtry = 58)
finalModel <- train(
    label ~ .,
    data = trainData[inTrain, ],
    method = "rf",
    trControl = fitControl,
    tuneGrid = fitGrid
)
```

How did we do this time?

```
print(finalModel)
```

```
## Random Forest
```

```
##
## 21003 samples
##   784 predictor
##    10 classes: '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
##
## No pre-processing
## Resampling: Cross-Validated (5 fold, repeated 3 times)
## Summary of sample sizes: 16803, 16803, 16802, 16803, 16801, 16802, ...
## Resampling results
##
##   Accuracy   Kappa      Accuracy SD  Kappa SD
##   0.9587674  0.9541714  0.003369295  0.003744758
##
## Tuning parameter 'mtry' was held constant at a value of 58
##
```

```
confusionMatrix(finalModel)
```

```
## Cross-Validated (5 fold, repeated 3 times) Confusion Matrix
##
## (entries are percentages of table totals)
##
##           Reference
## Prediction    0    1    2    3    4    5    6    7    8    9
##          0  9.7  0.0  0.1  0.0  0.0  0.0  0.1  0.0  0.0  0.1
##          1  0.0 10.9  0.0  0.0  0.0  0.0  0.0  0.1  0.1  0.0
##          2  0.0  0.1  9.5  0.2  0.0  0.0  0.0  0.1  0.0  0.0
##          3  0.0  0.0  0.1  9.7  0.0  0.1  0.0  0.0  0.1  0.2
##          4  0.0  0.0  0.1  0.0  9.3  0.0  0.0  0.0  0.1  0.2
##          5  0.0  0.0  0.0  0.2  0.0  8.6  0.1  0.0  0.1  0.0
##          6  0.0  0.0  0.1  0.0  0.1  0.1  9.6  0.0  0.0  0.0
##          7  0.0  0.0  0.1  0.1  0.0  0.0  0.0 10.0  0.0  0.1
##          8  0.1  0.0  0.1  0.1  0.0  0.1  0.0  0.0  9.1  0.1
##          9  0.0  0.0  0.0  0.1  0.2  0.0  0.0  0.1  0.1  9.3
```

Is that just overfit? How about on the other, reserve half of the data?

```
validData <- predict(finalModel, trainData[-inTrain, ])
confusionMatrix(validData, trainData[-inTrain, "label"])
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction    0    1    2    3    4    5    6    7    8    9
##          0 2029    0    9    4    3   10    9    1    2    8
##          1    0 2309    5    8    3    1    3    7   19    5
##          2    2   13 2012   38    5    3    2   28    9    9
##          3    1    7   15 2035    1   20    0    3   16   28
##          4    3    4    9    3 1957    4    5   14    8   36
##          5    3    2    1   26    0 1818   16    0   15    9
##          6   15    1    6    3   12   15 2025    0   12    2
##          7    0    4   16   14    0    1    0 2110    2   21
##          8   12    2   13   31    5   10    8    7 1926   10
```

```
##          9    1    0    2   13   50   15    0   30   22 1966
##
## Overall Statistics
##
##                Accuracy : 0.9614
##                  95% CI : (0.9587, 0.964)
##     No Information Rate : 0.1115
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9571
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity           0.98209   0.9859  0.96360  0.93563  0.96120  0.95836
## Specificity           0.99757   0.9973  0.99424  0.99517  0.99546  0.99623
## Pos Pred Value        0.97783   0.9784  0.94861  0.95720  0.95791  0.96190
## Neg Pred Value        0.99804   0.9982  0.99597  0.99258  0.99583  0.99587
## Prevalence            0.09840   0.1115  0.09944  0.10359  0.09697  0.09035
## Detection Rate        0.09663   0.1100  0.09582  0.09692  0.09320  0.08658
## Detection Prevalence  0.09882   0.1124  0.10101  0.10125  0.09730  0.09001
## Balanced Accuracy     0.98983   0.9916  0.97892  0.96540  0.97833  0.97729
##                      Class: 6 Class: 7 Class: 8 Class: 9
## Sensitivity           0.97921   0.9591  0.94830  0.93887
## Specificity           0.99651   0.9969  0.99483  0.99296
## Pos Pred Value        0.96844   0.9732  0.95158  0.93664
## Neg Pred Value        0.99773   0.9952  0.99447  0.99323
## Prevalence            0.09849   0.1048  0.09673  0.09973
## Detection Rate        0.09644   0.1005  0.09173  0.09363
## Detection Prevalence  0.09959   0.1033  0.09639  0.09997
## Balanced Accuracy     0.98786   0.9780  0.97157  0.96592
```

Part 4: Other Features

Additional Models

- Is a random forest not appropriate for your modeling task? There are over 200 other models `caret` can handle.
- Don't see what you want? Well, you'll get no help from me, but `caret` is capable of handling custom models.

Additional additions

- Instead of `createDataPartition` using the outcome, you can split on the predictors using (for example) maximum dissimilarity.
- You can also affect class subsampling by having `caret` up- or down-sample so that underrepresented classes carry more weight in model training.
- `caret` can help pre-process your data, often from right inside the `train` function.