A basic tutorial of caret: the machine learning package in R

R has a wide number of packages for machine learning (ML), which is great, but also quite frustrating since each package was designed independently and has very different syntax, inputs and outputs. Caret unifies these packages into a single package with constant syntax, saving everyone a lot of frustration and time!

R MACHINE LEARNING

AUTHOR
Rebecca Barter

PUBLISHED
November 17, 2017

Note: If you're new to caret, I suggest learning tidymodels instead http://www.rebeccabarter.com/blog/2020-03-25_machine_learning/. Tidymodels is essentially caret's successor. Don't worry though, your caret code will still work!

Older note: This tutorial was based on an older version of the abalone data that had a binary `old` varibale rather than a numeric `age` variable. It has been modified lightly so that it uses a manual `old` variable (is the abalone older than 10 or not) and ignores the numeric `age` variable.

Materials prepared by Rebecca Barter. Package developed by Max Kuhn.

An interactive Jupyter Notebook version of this tutorial can be found at https://github.com/rlbarter/STAT-215A-Fall-2017/tree/master/week11. Feel free to download it and use for your own learning or teaching adventures!

R has a wide number of packages for machine learning (ML), which is great, but also quite frustrating since each package was designed independently and has very different syntax, inputs and outputs.

This means that if you want to do machine learning in R, you have to learn a large number of separate methods.

Recognizing this, Max Kuhn (at the time working in drug discovery at Pfizer, now at RStudio) put together a single package for performing any machine learning method you like. This package is called `caret`. Caret stands for **C**lassification **A**nd **Re**gression **T**raining. Apparently caret has little to do with our orange friend, the carrot.

Not only does caret allow you to run a plethora of ML methods, it also provides tools for auxiliary techniques such as:

- Data preparation (imputation, centering/scaling data, removing correlated predictors, reducing skewness)
- Data splitting
- Variable selection
- Model evaluation

An extensive vignette for caret can be found here: https://topepo.github.io/caret/index.html

A simple view of caret: the default `train` function

To implement your machine learning model of choice using caret you will use the `train` function. The types of modeling options available are many and are listed here: https://topepo.github.io/caret/available-models.html. In the example below, we will use the ranger implementation of random forest to predict whether abalone are "old" or not based on a bunch of physical properties of the abalone (sex, height, weight, diameter, etc). The abalone data came from the UCI Machine Learning repository (we split the data into a training and test set).

First we load the data into R:

```
# load in packages
library(caret)
library(ranger)
library(tidyverse)
library(e1071)
# load in abalone dataset
abalone_data <- read.table("data/abalone.data", sep = ",")
# load in column names
colnames(abalone_data) <- c("sex", "length", "diameter", "heigh
```

```
                            "whole.weight", "shucked.weight",
                            "viscera.weight", "shell.weight", "a
# add a logical variable for "old" (age > 10)
abalone_data <- abalone_data %>%
  mutate(old = age > 10) %>%
  # remove the "age" variable
  select(-age)
# split into training and testing
set.seed(23489)
train_index <- sample(1:nrow(abalone_data), 0.9 * nrow(abalone_d
abalone_train <- abalone_data[train_index, ]
abalone_test <- abalone_data[-train_index, ]
# remove the original dataset
rm(abalone_data)
# view the first 6 rows of the training data
head(abalone_train)
```

```
     sex length diameter height whole.weight shucked.weight
viscera.weight
232    M  0.565    0.440  0.175       0.9025         0.3100
0.1930
3906   M  0.380    0.270  0.095       0.2190         0.0835
0.0515
1179   F  0.650    0.500  0.190       1.4640         0.6415
0.3390
2296   F  0.520    0.415  0.145       0.8045         0.3325
0.1725
1513   F  0.650    0.500  0.160       1.3825         0.7020
0.3040
1023   F  0.640    0.500  0.170       1.5175         0.6930
0.3260
     shell.weight   old
232        0.3250  TRUE
3906       0.0700 FALSE
1179       0.4245 FALSE
2296       0.2850 FALSE
1513       0.3195 FALSE
1023       0.4090  TRUE
```

It looks like we have 3,759 abalone:

```
dim(abalone_train)
```

```
[1] 3759    9
```

Time to fit a random forest model using caret. Anytime we want to fit a model using `train` we tell it which model to fit by providing a formula for the first argument (`as.factor(old) ~ .` means that we want to model `old` as a function of all of the other variables). Then we need to

provide a method (we specify `"ranger"` to implement randomForest).

```
# fit a random forest model (using ranger)
rf_fit <- train(as.factor(old) ~ .,
                data = abalone_train,
                method = "ranger")
```

By default, the `train` function without any arguments re-runs the model over 25 bootstrap samples and across 3 options of the tuning parameter (the tuning parameter for `ranger` is `mtry`; the number of randomly selected predictors at each cut in the tree).

```
rf_fit
```

```
No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 3759, 3759, 3759, 3759, 3759, 3759,
...
Resampling results across tuning parameters:

  mtry  splitrule   Accuracy   Kappa
  2     gini        0.7794339  0.4982012
  2     extratrees  0.7788261  0.4867672
  5     gini        0.7722038  0.4853445
  5     extratrees  0.7784925  0.4974177
  9     gini        0.7665692  0.4738511
  9     extratrees  0.7759596  0.4933252

Tuning parameter 'min.node.size' was held constant at a value
of 1
Accuracy was used to select the optimal model using the largest
value.
The final values used for the model were mtry = 2, splitrule =
gini
 and min.node.size = 1.
```

To test the data on an independent test set is equally as simple using the inbuilt `predict` function.

```
# predict the outcome on a test set
abalone_rf_pred <- predict(rf_fit, abalone_test)
# compare predicted outcome and true outcome
confusionMatrix(abalone_rf_pred, as.factor(abalone_test$old))
```

```
Confusion Matrix and Statistics

          Reference
Prediction FALSE TRUE
```

```
     FALSE   229   60
     TRUE     33   96

               Accuracy : 0.7775
                 95% CI : (0.7346, 0.8165)
    No Information Rate : 0.6268
    P-Value [Acc > NIR] : 2.672e-11

                  Kappa : 0.5072

 Mcnemar's Test P-Value : 0.007016

            Sensitivity : 0.8740
            Specificity : 0.6154
         Pos Pred Value : 0.7924
         Neg Pred Value : 0.7442
             Prevalence : 0.6268
```

Getting a little fancier with caret

We have now seen how to fit a model along with the default resampling implementation (bootstrapping) and parameter selection. While this is great, there are many more things we could do with caret.

Pre-processing (`preProcess`)

There are a number of pre-processing steps that are easily implemented by caret. Several stand-alone functions from caret target specific issues that might arise when setting up the model. These include

- `dummyVars` : creating dummy variables from categorical variables with multiple categories
- `nearZeroVar` : identifying zero- and near zero-variance predictors (these may cause issues when subsampling)
- `findCorrelation` : identifying correlated predictors
- `findLinearCombos` : identify linear dependencies between predictors

In addition to these individual functions, there also exists the **`preProcess`** function which can be used to perform more common tasks such as centering and scaling, imputation and transformation. `preProcess` takes in a data frame to be processed and a method which can be any of “BoxCox”, “YeoJohnson”, “expoTrans”, “center”, “scale”, “range”, “knnImpute”, “bagImpute”, “medianImpute”, “pca”, “ica”,

"spatialSign", "corr", "zv", "nzv", and "conditionalX".

```
# center, scale and perform a YeoJohnson transformation
# identify and remove variables with near zero variance
# perform pca
abalone_no_nzv_pca <- preProcess(select(abalone_train, - old),
                         method = c("center", "scale", "nzv", "pc
abalone_no_nzv_pca
```

```
Created from 3759 samples and 8 variables

Pre-processing:
  - centered (7)
  - ignored (1)
  - principal component signal extraction (7)
  - scaled (7)

PCA needed 3 components to capture 95 percent of the variance
```

```
# identify which variables were ignored, centered, scaled, e
abalone_no_nzv_pca$method
```

```
$center
[1] "length"         "diameter"       "height"
"whole.weight"
[5] "shucked.weight" "viscera.weight" "shell.weight"

$scale
[1] "length"         "diameter"       "height"
"whole.weight"
[5] "shucked.weight" "viscera.weight" "shell.weight"

$pca
[1] "length"         "diameter"       "height"
"whole.weight"
[5] "shucked.weight" "viscera.weight" "shell.weight"

$ignore
[1] "sex"
```

```
# identify the principal components
abalone_no_nzv_pca$rotation
```

```
                    PC1         PC2        PC3
length        -0.3835950  0.01308476 -0.5915192
diameter      -0.3838966  0.03978406 -0.5874657
```

```
height         -0.3458509  0.88289420  0.2793599
whole.weight   -0.3910710 -0.22191114  0.2394200
shucked.weight -0.3784382 -0.33048177  0.2601988
viscera.weight -0.3819522 -0.23798574  0.2841819
shell.weight   -0.3792439 -0.06036456  0.1454731
```

Data splitting (`createDataPartition` and `groupKFold`)

Generating subsets of the data is easy with the `createDataPartition` function. While this function can be used to simply generate training and testing sets, it can also be used to subset the data while respecting important groupings that exist within the data.

First, we show an example of performing general sample splitting to generate 10 different 80% subsamples.

```
# identify the indices of 10 80% subsamples of the iris data
train_index <- createDataPartition(iris$Species,
                                   p = 0.8,
                                   list = FALSE,
                                   times = 10)
```

```
# look at the first 6 indices of each subsample
head(train_index)
```

```
     Resample01 Resample02 Resample03 Resample04 Resample05
Resample06
[1,]          3          3          1          1          1
2
[2,]          4          4          2          2          2
3
[3,]          5          5          3          3          3
4
[4,]          6          6          5          4          4
5
[5,]          7          9          6          5          6
6
[6,]          8         10         10          6          7
7
     Resample07 Resample08 Resample09 Resample10
[1,]          2          2          1          2
[2,]          4          3          3          5
[3,]          5          4          4          6
[4,]          6          5          5          7
```

```
[5,]          8          6          8          9
[6,]          9          7          9         11
```

While the above is incredibly useful, it is also very easy to do using a for loop. Not so exciting.

Something that IS more exciting is the ability to do K-fold cross validation which respects groupings in the data. The `groupKFold` function does just that!

As an example, let's consider the following made-up abalone groups so that each sequential set of 5 abalone that appear in the dataset together are in the same group. For simplicity we will only consider the first 50 abalone.

```
# add a madeup grouping variable that groupes each subsequen  5
# filter to the first 50 abalone for simplicity
abalone_grouped <- cbind(abalone_train[1:50, ], group = rep(1:1(
head(abalone_grouped, 10)
```

```
     sex length diameter height whole.weight shucked.weight
viscera.weight
232    M  0.565    0.440  0.175       0.9025         0.3100
0.1930
3906   M  0.380    0.270  0.095       0.2190         0.0835
0.0515
1179   F  0.650    0.500  0.190       1.4640         0.6415
0.3390
2296   F  0.520    0.415  0.145       0.8045         0.3325
0.1725
1513   F  0.650    0.500  0.160       1.3825         0.7020
0.3040
1023   F  0.640    0.500  0.170       1.5175         0.6930
0.3260
2390   M  0.420    0.340  0.125       0.4495         0.1650
0.1125
856    F  0.575    0.465  0.140       0.9580         0.4420
0.1815
2462   F  0.500    0.385  0.130       0.7680         0.2625
0.0950
2756   F  0.525    0.415  0.150       0.7055         0.3290
```

The following code performs 10-fold cross-validation while respecting the groups in the abalone data. That is, each group of abalone must always appear in the same group together.

```
# perform grouped K means
group_folds <- groupKFold(abalone_grouped$group, k = 10)
group_folds
```

```
$Fold1
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 21 22 23 24
25 26 27 28 29 30
[26] 31 32 33 34 35 41 42 43 44 45 46 47 48 49 50

$Fold2
 [1]  1  2  3  4  5  6  7  8  9 10 16 17 18 19 20 21 22 23 24
25 26 27 28 29 30
[26] 36 37 38 39 40 41 42 43 44 45

$Fold3
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 46 47 48 49
50

$Fold4
 [1]  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 26 27 28 29
30 31 32 33 34 35
[26] 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

Resampling options (`trainControl`)

One of the most important part of training ML models is tuning parameters. You can use the **`trainControl`** function to specify a number of parameters (including sampling parameters) in your model. The object that is outputted from `trainControl` will be provided as an argument for `train`.

```
set.seed(998)
# create a testing and training set
in_training <- createDataPartition(abalone_train$old, p = .75,
training <- abalone_train[ in_training,]
testing  <- abalone_train[-in_training,]
```

```
# specify that the resampling method is
fit_control <- trainControl(## 10-fold CV
                            method = "cv",
                            number = 10)
```

```
# run a random forest model
set.seed(825)
rf_fit <- train(as.factor(old) ~ .,
                data = abalone_train,
                method = "ranger",
```

```
                trControl = fit_control)
rf_fit
```

```
Random Forest

3759 samples
   8 predictor
   2 classes: 'FALSE', 'TRUE'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 3384, 3383, 3383, 3382, 3383, 3383,
...
Resampling results across tuning parameters:

  mtry  splitrule   Accuracy   Kappa
  2     gini        0.7826656  0.5054371
  2     extratrees  0.7853266  0.5032091
  5     gini        0.7765528  0.4953944
  5     extratrees  0.7850614  0.5120121
  9     gini        0.7683032  0.4787823
  9     extratrees  0.7810713  0.5057059

Tuning parameter 'min.node.size' was held constant at a value
```

We could instead use our **grouped folds** (rather than random CV folds) by assigning the `index` argument of `trainControl` to be `grouped_folds`.

```
# specify that the resampling method is
group_fit_control <- trainControl(## use grouped CV folds
                                  index = group_folds,
                                  method = "cv")
set.seed(825)
rf_fit <- train(as.factor(old) ~ .,
                data = select(abalone_grouped, - group),
                method = "ranger",
                trControl = group_fit_control)
```

```
rf_fit
```

```
Random Forest

50 samples
 8 predictor
 2 classes: 'FALSE', 'TRUE'

No pre-processing
Resampling: Cross-Validated (10 fold)
```

```
Summary of sample sizes: 40, 35, 45, 40, 45, 45, ...
Resampling results across tuning parameters:

  mtry  splitrule   Accuracy   Kappa
  2     gini        0.5222222  0.03968254
  2     extratrees  0.5111111  0.03784970
  5     gini        0.5444444  0.01758658
  5     extratrees  0.5333333  0.08743687
  9     gini        0.5777778  0.08071789
  9     extratrees  0.5555556  0.13952020

Tuning parameter 'min.node.size' was held constant at a value
```

You can also pass functions to `trainControl` that would have otherwise been passed to `preProcess`.

Model parameter tuning options (`tuneGrid =`)

You could specify your own tuning grid for model parameters using the `tuneGrid` argument of the `train` function. For example, you can define a grid of parameter combinations.

```
# define a grid of parameter options to try
rf_grid <- expand.grid(mtry = c(2, 3, 4, 5),
                       splitrule = c("gini", "extratrees"),
                       min.node.size = c(1, 3, 5))
rf_grid
```

```
   mtry  splitrule min.node.size
1     2       gini             1
2     3       gini             1
3     4       gini             1
4     5       gini             1
5     2 extratrees             1
6     3 extratrees             1
7     4 extratrees             1
8     5 extratrees             1
9     2       gini             3
10    3       gini             3
11    4       gini             3
12    5       gini             3
13    2 extratrees             3
14    3 extratrees             3
15    4 extratrees             3
16    5 extratrees             3
17    2       gini             5
```

```
18    3       gini            5
19    4       gini            5
20    5       gini            5
```

```
# re-fit the model with the parameter grid
rf_fit <- train(as.factor(old) ~ .,
                data = select(abalone_grouped, -group),
                method = "ranger",
                trControl = group_fit_control,
                # provide a grid of parameters
                tuneGrid = rf_grid)
rf_fit
```

```
Random Forest

50 samples
 8 predictor
 2 classes: 'FALSE', 'TRUE'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 40, 35, 45, 40, 45, 45, ...
Resampling results across tuning parameters:

  mtry  splitrule   min.node.size  Accuracy   Kappa
  2     gini        1              0.5722222   0.083698830
  2     gini        3              0.4944444  -0.009825701
  2     gini        5              0.5388889   0.012270259
  2     extratrees  1              0.5111111   0.037849695
  2     extratrees  3              0.5277778   0.085035842
  2     extratrees  5              0.5277778   0.085035842
  3     gini        1              0.5555556   0.111111111
  3     gini        3              0.5888889   0.111111111
  3     gini        5              0.5722222   0.066856453
```

Advanced topics

This tutorial has only scratched the surface of all of the options in the caret package. To find out more, see the extensive vignette https://topepo.github.io/caret/index.html.