INTRODUCTION TO CARET (A CLASSIFICATION LENS)

Fiona Lodge
RLadies Meetup

4/22/2019

MOTIVATION

This presentation will go through main functions of the `caret` package, which includes the following: `CreateDataPartition`, `PreProcess`, `trainControl` and `train`. We will focus on aspects of classifications problems for model training and tuning, and the predictors in these problems will be only numeric. The focus on this presentation is to demonstrate the processes of `caret`, not to talk about the mathematics of models.

I was motivated to learn caret because I am working through the book, *Applied Predictive Modeling* (Kuhn, 2016) to improve my statistical modeling skills. This book contains modeling problems and examples done in R, along with statistical explanations. Although the book does not exclusively rely on `caret`, you could work through the problems with `caret`.

I was also motivated to learn `caret` because my work flow tends to switch between R to Python. I find the EDA process to be more suited to using R and the modeling (sci-kit learn) to more suited to using Python. In terms of ease, uniformity, and the ability to run multiple models easily, the `caret` package seems comparable.

MY BACKGROUND

- Data science internship at Securian Financial. Seeking employment for June or thereafter.
- Received a master's (P.S.M) in Applied and Industrial Mathematics from the University of Stout.
- Former life includes seven years as a high school math teacher, and still currently a high school softball coach.
- I enjoy mathematics for its depth, there are always 'new things to learn', or 'new ways to learn old things'!
- What I've learned from `caret` has come from about a 1-2 months of practice, so I hoping to get some questions I can't answer today! This will mostly get your toes wet with and I recommend reading the vignette and the *Applied Predictive Modeling* for more practice. Kuhn also lists more recommendations in the introduction of the `caret` vignette.
- https://www.linkedin.com/in/fionarlodge/

DESCRIPTION OF CARET

`Caret` is a predictive modeling package created by Max Kuhn and is an acronym for **C**lassification **A**nd **RE**gression **T**raining. The functions in the `caret` package focus on the following main aspects of model development:

- data splitting
- pre-processing
- feature selection
- model tuning using resampling
- variable importance estimation

(Kuhn, 2019)

WHY CARET

Model functions in R vary in syntax, resampling methods, and tuning parameters. Compare the differences between some of the available inputs of a `randomForest` and `gbm` model, which are both tree-based models that can be used for both classification and regression.

	randomForest	gbm
form	`x , y`	`y ~ x`
resampling	`oob`	`cv.folds`
sample tuning parameter	`ntrees`	`n.trees`

HOW CARET STREAMLINES MODELING

Having to work through all of these varying syntaxes (and also functions) can significantly slow down the modeling process.

The `caret` package streamlines this in many ways:

- Pre-Processing, resampling, and tuning all have their own specific functions. This allows you to quickly work and change paths if required.
- The Pre-Processing functions are built to aid in specific development of your training set.
- There are a lot of models tags available, and you should be able to run at least a few different models for any given problem. There are also tools available for choosing the best model between many models.
- The functionality in this package is very deep, and there are a variety of ways to tune the models that you build.
- Lastly, uniform interface for any model!

OTHER BENEFITS TO CARET

- I have found the error messages are helpful. I actually learned a few things about the modeling process from them.
- The package will ask you to install package dependencies, instead of stopping its processes and asking you to start over. This also means that you only need to load `caret` to be able to access a many different modeling libraries!
- Kuhn has included many options for customization, including with preprocessing and models not included in the tagged list.
- It is easy to find tuning parameters (as opposed to sorting through a long list of model inputs).
- The package contains some datasets to practice on.
- I found it refreshing to work through a book written by the author of the package. I got a better understanding of the package structure, functions, and motivation.

THE GENERATED DATASET

- For most of the examples today, I generated a random dataset in Python using the `datasets.make_classification` command in `sci-kit learn`.

```
X, y = make_classification(500, n_features= 5, n_informative = 3,
n_redundant = 0, n_classes = 3, weights = [0.6, 0.3, 0.1])
```

- I also manually added a few missing values. Below is the head of the dataset, and in the code, the dataset is (creatively) referred to as `dat`.

	X0	X1	X2	X3	X4	y
1	-1.4588104	-0.08047330	-0.6332306	2.4122099	1.2415181	B
2	0.6027192	0.36260468	1.4755119	-1.6016240	1.2016910	A
3	-2.7398292	0.09769184	1.5173355	-0.5655078	-2.0879155	B
4	-1.3368162	0.69000199	2.2298045	-1.1350876	-1.8826160	B
5	0.3215398	-0.08808579	0.9228054	0.7519209	0.6560173	A
6	-0.2997237	-0.40722466	-1.4362178	-1.8426319	0.7878204	B

DATA SPLITTING (TRAIN/TEST)

The data splitting function, `CreateDataPartition`, creates a list (or matrix) of indices for your training data. Below is a sample command for the dataset that use 75% of its data for training.

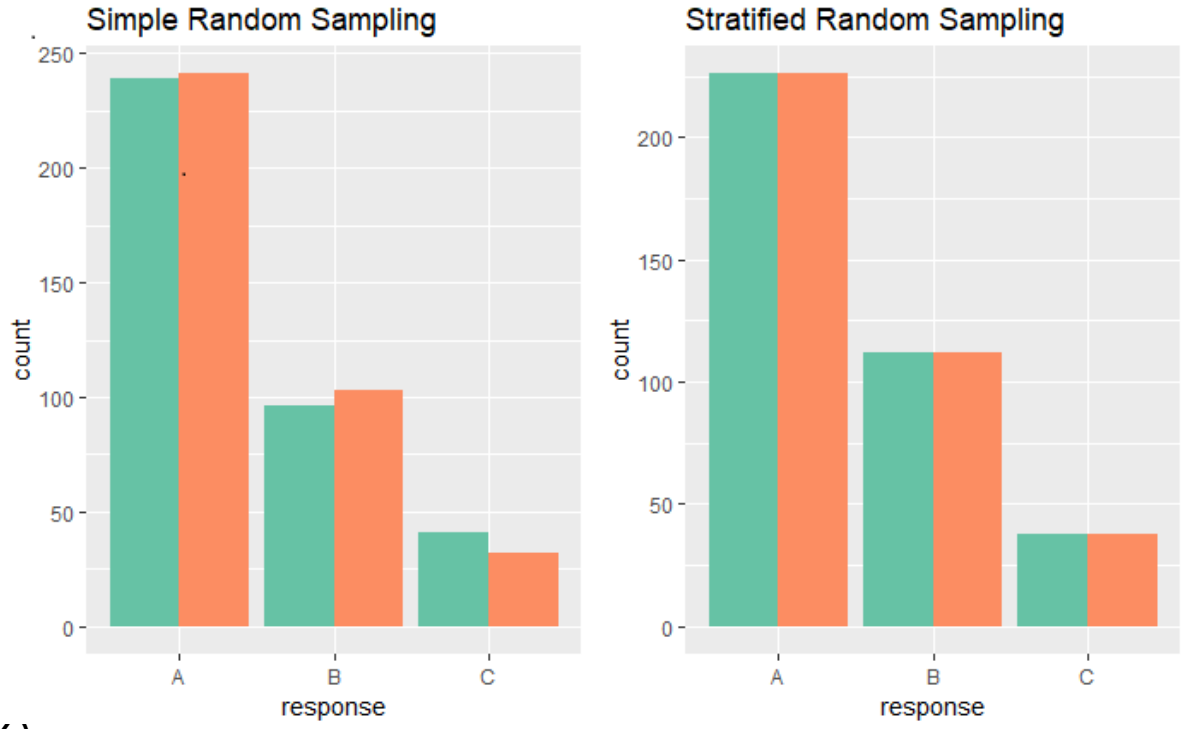
```
set.seed(809)
trainindex <-
  createDataPartition(dat$y, # response always goes here
                      p = 0.75, # fraction to put into training set
                      list = FALSE)
```

Some imputation methods require the format of `x = x` and `y = y`, so that is what I have chosen here, but this may not always be necessary.

```
train.x <- dat[trainindex, -which(names(dat) %in% c('y'))]
train.y <- dat[trainindex, 'y']
test.x <- dat[-trainindex, -which(names(dat) %in% c('y'))]
test.y <- dat[-trainindex, 'y']
```

DATA SPLITTING 2

The `createDataPartition` command uses stratified (proportional) random sampling to find its splits. The plots below compare this method to using the `sample` method in R. Stratified random sampling is preferred for classification problems.



(Kuhn, 2016)

PRE-PROCESSING

The `preProcess` function has many abilities. This includes data transformation, imputation, and even predictor removal. The function itself is descriptive in nature, while `predict.preProcess` will produce the new dataframe. The function, `preProcess` is also available in the `train` function, which will be covered later.

```
preproc <- preProcess(train.x, c('knnImpute'))
preproc

transformed <- predict(preproc, train.x) # This creates a new dataframe
```

```
## Created from 369 samples and 5 variables

## Pre-processing:

##   - centered (5)

##   - ignored (0)

##   - 5 nearest neighbor imputation (5)

##   - scaled (5)
```

PRE-PROCESSING: LINEAR DEPENDENCIES

`caret` has a variety of functions available to assess the quality of your data. For example, if a dataset contains linear combinations of predictors, this can cause misleading results (in short, repetitive information). See the quick example below, where city and zip are often represented by the same vector. Imagine if you were to only have these data points, it would be hard to pick out a functional signal for your model!

```
##          city    zip beds baths sqft        type price latitude longitude
## 1 SACRAMENTO z95838    2     1  836 Residential 59222 38.63191 -121.4349
## 2 SACRAMENTO z95823    3     1 1167 Residential 68212 38.47890 -121.4310
## 3 SACRAMENTO z95815    2     1  796 Residential 68880 38.61830 -121.4438
## 4 SACRAMENTO z95815    2     1  852 Residential 69307 38.61684 -121.4391
## 5 SACRAMENTO z95824    2     1  797 Residential 81900 38.51947 -121.4358
## 6 SACRAMENTO z95841    3     1 1122       Condo 89921 38.66260 -121.3278
```

PRE-PROCESSING: LINEAR DEPENDENCIES

The `findLinearCombos` function will provide you with a recommendation of predictor columns to remove.

```
data("Sacramento")
head(Sacramento)
lincombs <- findLinearCombos(model.matrix(price ~., data = Sacramento))
lincombs$remove # recommended columns to remove
```

```
##  [1]  38  40  41  42  43  45  46  47  48
49  50  51  52  53  54  55  57
## [18]  58  59  61  64  65  66  67  68  69
70  71  72  73  75  76  77 103
## [35] 104
```

YOUR TURN: EXPLORE THE PREPROCESSING TECHNIQUE OF NEARZEROVAR

1. Install and load `caret`. Look up `?nearZeroVar` to get an understanding of what the function does.
2. Load the dataset German credit with `data("GermanCredit")`. The response variable is `Class`.
3. Split the data into a train/test split with `createDataPartition`.
4. Explore the proportional distribution of the predictors in the training set using the `nearZeroVar` command. Note that this could also be done with the `preProcess` command with `method = "nzv"`.

(Kuhn, 2019)

(Hoffman)

OTHER PREPROCESSING TECHNIQUES

There are quite a few other preprocessing techniques available:

- The `dummyVars` function creates dummy variables. R seems to have a few different formulas to create model matrix type data, and I recommend reading before you go!
- The plethora of available methods in `preProcess`:
  - `BoxCox`, `YeoJohnson`,
  - `expoTrans`, `center`, `scale`, `range`,
  - `knnImpute`, `bagImpute`, `medianImpute`,
  - `pca`, `ica`, `spatialSign`, `corr`,
  - `zv`, `nzv`, `conditionalX`

TRAINING PARAMETERS

Training parameters and resampling procedures are set up in the `trainControl` function. Below are some of the more well known methods, none, k-fold, repeated k-fold, and out-of-bag.

```
ctrl.none <- trainControl(method = 'none') # fits model to training set
without resampling
ctrl.cv <- trainControl(method = 'cv', number = 10) # 10-fold cross
validation
ctrl.rpcv <- trainControl(method = 'repeatedcv', number = 10, repeats =
10) #  10-pete of 10-fold cv
ctrl.oob <- trainControl(method = 'oob') # Out of bag sample
```

For `dat`, we will use `ctrl.rpcv`.

THE GBM MODEL AND ITS TUNING PARAMETERS

The `gbm` model is a stochastic gradient boosting model that minimizes a loss function to a certain distribution. You do not need to understand it for the purposes of this presentation, but you should know that `interaction.depth`, `n.trees`, and `shrinkage` have the most affect on the accuracy of your model. Note that choosing a small shrinkage should positively affect results, but also significantly slow done processing time.

```
modelLookup('gbm')[, 1:3] # Nice way to extract tuning parameters
```

```
##   model          parameter                   label
## 1   gbm            n.trees  # Boosting Iterations
## 2   gbm  interaction.depth         Max Tree Depth
## 3   gbm          shrinkage               Shrinkage
## 4   gbm n.minobsinnode Min. Terminal Node Size
```

A FIRST LOOK AT TRAIN

The `train` function is the heart of `caret`, and this is where everything comes together (pre-processing, training parameters, and tuning). This is the function where your model gets trained.

```
gbm.fit <- train(x = train.x,
                 y = train.y,
                 trControl = ctrl.rpcv, #our repeated cv method
                 method = 'gbm',
                 preProcess = c('knnImpute'),
                 verbose = FALSE,
                 # This will keep gbm from printing a tome
                 metric = 'Accuracy')
```

RESULTS OF TRAIN

```
Stochastic Gradient Boosting

376 samples
  5 predictor
  3 classes: 'A', 'B', 'C'

Pre-processing: nearest neighbor imputation (5), centered
 (5), scaled (5)
Resampling: Cross-Validated (10 fold, repeated 10 times)
Summary of sample sizes: 338, 338, 340, 337, 338, 339, ...
Resampling results across tuning parameters:

  interaction.depth  n.trees  Accuracy   Kappa
  1                   50      0.7758597  0.5642986
  1                  100      0.7738887  0.5636193
  1                  150      0.7748071  0.5666376
  2                   50      0.7985457  0.6136391
  2                  100      0.7971910  0.6138592
  2                  150      0.8018799  0.6232320
  3                   50      0.8095675  0.6360303
  3                  100      0.8096054  0.6372956
  3                  150      0.8053504  0.6287952

Tuning parameter 'shrinkage' was held constant at a value of
 0.1
Tuning parameter 'n.minobsinnode' was held constant at a
 value of 10
Accuracy was used to select the optimal model using the
 largest value.
The final values used for the model were n.trees =
 100, interaction.depth = 3, shrinkage = 0.1 and n.minobsinnode = 10.
```

YOUR TURN WITH TRAIN

We will run a random forest model on the popular Iris (Fisher, 1936) dataset. The `method = "rf"` is actually the default in `train`. The only tuning parameter in the `randomForest` tagged model is `mtry`, which is the number of randomly selected predictors.

1. Below is a reminder of how to set up a train/test set, but feel free to do your own!

```
data("iris")

idx <- createDataPartition(iris$Species, p = 0.8, list = FALSE)

trn.x <- iris[idx, -which(names(iris) %in% c("Species"))]
trn.y = iris[idx, c("Species")]
tst.x <- iris[-idx, -which(names(iris) %in% c("Species"))]
tst.y <- iris[-idx, c("Species")]
```

2. Set up a 5-fold cross validation method, center and scale the data, and train a `randomForest` model.

TRAINING WITH CUSTOMIZED GRID

You can set up a customized grid that sets up your own tuning parameters in the `tuneGrid` section of the `train` command.

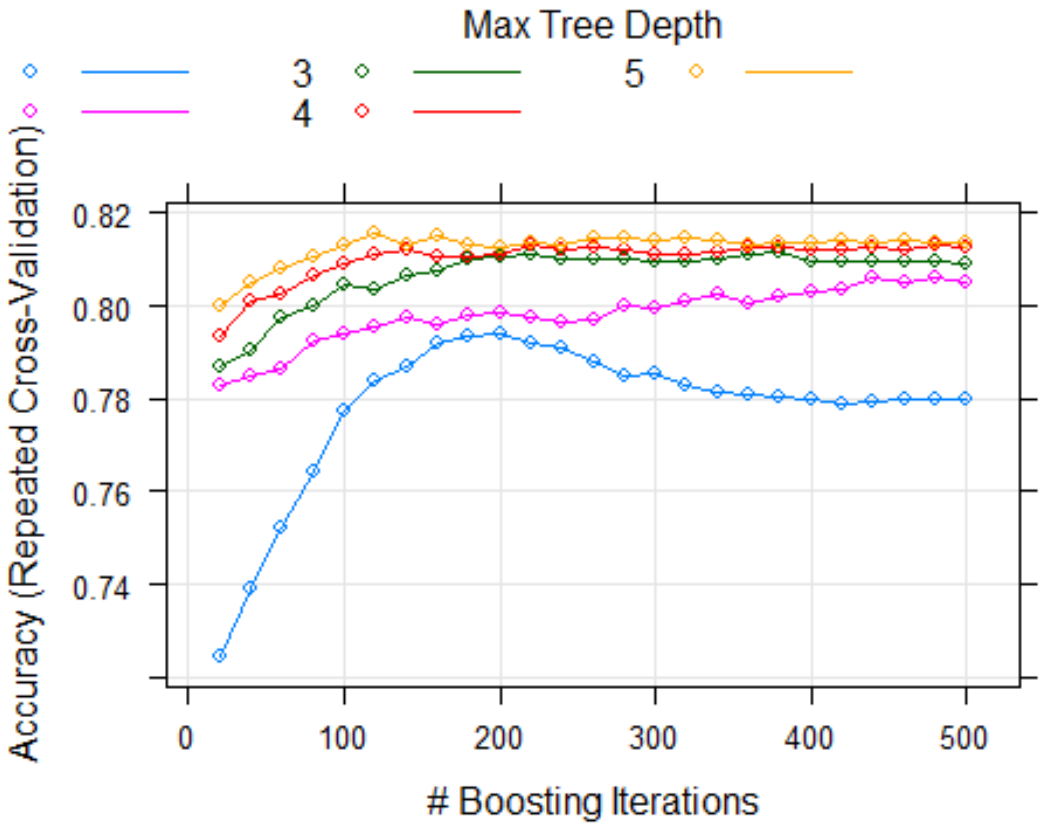
```
gbmGrid <-  expand.grid(interaction.depth = c(1:5),
                        n.trees = (1:25)*20,
                        shrinkage = 0.01,
                        n.minobsinnode = 10)

gbm.fit.grid <- train(x = train.x,
                      y =  train.y,
                       trControl = ctrl.rpcv,
                       method = 'gbm',
                       preProcess = c('knnImpute'),
                       verbose = FALSE,
                       tuneGrid = gbmGrid) # Include your grid here.
```

Note that there is one more way to train – by using a random search, `search = "random"`, and setting the `tuneLength`.

TOOLS FOR WORKING WITH THE TRAIN RESULTS

The results across tuning parameters are available for plotting with the `plot.train` function.



TOOLS FOR WORKING WITH THE TRAIN RESULTS

The default is for `caret` is to determine the best model by the `metric` given and there are a variety of ways to view this optimized model.

```
gbm.fit.grid$bestTune
# Gives optimimzed tuning parameters
gbm.fit.grid$finalModel
# Final model object, note this will not preprocess the test data if
used for predictions
gbm.fit.grid$results
# This dataframe contains all results and is useful for choosing a
model based on a different statistical measure.
```

GETTING PREDICTIONS FROM YOUR MODEL

The function, `predict.train` can be used to get predictions from the optimized model.

```
preds <- predict(gbm.fit.grid, test.x)
table(preds, test.y)
```

```
##       test.y
## preds  A  B  C
##     A 67 12  0
##     B  7 23  3
##     C  1  2  9
```

An additional function, `extractPrediction` which will output a dataframe full of information concerning your predictions (`obs, pred, model, object, dataType`). Note that this function can also be applied to a list of different models.

CONFUSION MATRIX

Accuracy is not the only metric to analyze a classification model with. The `confusionMatrix` command provides several other associated statistics, below is a sample of the output.

```
confusionMatrix(data = preds, reference = test.y)
```

```
Statistics by Class:

                     Class: A Class: B Class: C
Sensitivity            0.8933   0.6216  0.75000
Specificity            0.7551   0.8851  0.97321
Pos Pred Value         0.8481   0.6970  0.75000
Neg Pred Value         0.8222   0.8462  0.97321
Prevalence             0.6048   0.2984  0.09677
Detection Rate         0.5403   0.1855  0.07258
Detection Prevalence   0.6371   0.2661  0.09677
Balanced Accuracy      0.8242   0.7533  0.86161
```

YOUR TURN: PRACTICE WITH THE IRIS MODEL

1. Plot the results of the random forest model. If you want to try a different tuning method, feel free! (Hint: you will have to adjust the mtry parameter).
2. Build predictions on the test set.
3. Print a confusion matrix.

SAMPLING METHODS TO DEAL WITH CLASS IMBALANCE

The distribution of the response class in my dataset is unbalanced, which can affect the results of model training.

The distribution of the response class in my dataset is unbalanced, which can affect the results of model training.

```
## train.y
##   A   B   C
## 226 112  38
```

There are a variety of rebalancing methods in `caret` to handle this, and we will try `upSample`. In essence, this randomly samples from the unbalanced classes until we reach equal proportions.

```
up_train <- upSample(x = train.x,
                     y = train.y)
```

```
##   Class     n
## 1 A       226
## 2 B       226
## 3 C       226
```

TRAINING ON THE UPSAMPLED DATA

This might (or not) improve a fit, but let's compare it to our original `gbm.fit` object. Both results are similiar, but this example also shows us how we can use `ExtractPrediction` to compare models.

```
#Note, you will want to rename these in your list
model.list <- list('original.fit' = gbm.fit,
                   'upsampled.fit' = train.up)

preds.compare <- extractPrediction(model.list,
                                   testX = test.x,
                                   testY = test.y)
preds.compare %>%
  filter(dataType == 'Test')%>%
  group_by(object) %>%
  mutate(accuracy = if_else(obs == pred, 1, 0)) %>%
  summarise(per.accurate = sum(accuracy)/n())
```

```
##    object         per.accurate
## 1 original.fit          0.831
## 2 upsampled.fit         0.806
```

YOUR TURN

Fatty acids were measured in different oil types (coded as A), sunflower (B), peanut (C), olive (D), soybean (E), rapeseed (F) and corn (G). The variable `FattyAcids` contains the predictors, whilst the variable, `oilType` contains the response.

1. By plot or count, view the distribution of the response variable and start to think about how this might affect the model results.

2. Split the data into a train/test split.

3. Train a classification model of your choice. There shouldn't be much pre-processing required.

4. Your choice from here! Run predictions on your test set, go back and tune your model…

(Brodnjak-Vonina et al. (2005))

(Kuhn, 2016)

GOING FURTHER WITH CARET

- Several feature selection tools available. For the `gbm` model we could have used recursive feature selection.
- Tools available for analysis of classification problems with two classes.
- A variety of visualization tools available, specifically geared towards analysis of EDA required for the modeling processes.
- The ability to create recipes in train for customized pre-processing methods.
- Parallel processing methods.
- And….SO much more…we only made it through chapter 5 and a bit more of the `caret` vignette.

BIBLIOGRAPHY

- Kuhn, M., & Johnson, K. (2016). *Applied predictive modeling*. New York: Springer.
- Kuhn, M. (2019, March 27). The caret Package. Retrieved from http://topepo.github.io/caret/
- Datasets
  - Fisher, R. A. (1936) The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, **7**, Part II, 179–188
  - Sacramento Data. (n.d.). Retrieved from https://support.spatialkey.com/spatialkey-sample-csv-data/
  - Hofman, D. (n.d.). Retrieved from https://archive.ics.uci.edu/ml/datasets/statlog (german credit data)
  - Brodnjak-Voncina et al. (2005). Multivariate data analysis in classification of vegetable oils characterized by the content of fatty acids, *Chemometrics and Intelligent Laboratory Systems*, Vol. 75:31-45