

Predictive Modeling with The R CARET Package

Matthew A. Lanham, CAP (lanham@vt.edu)

Doctoral Candidate

Department of Business Information Technology

MatthewALanham.com



VirginiaTech.

Pamplin College of Business

R

R is a software environment for data analysis, computing, and graphics.

Brief History

- 1991, R was developed by New Zealand Professors Robert Gentleman and Ross Ihaka who wanted a better statistical software platform for their students to use in their Macintosh teaching laboratory ([Ihaka & Gentleman, 1996](#)). They decided to model the R language after John Chamber's **statistical S language**.
- 1993, R was first announced to the public.
- 1995, R was made "free" under the GNU General Public License
- 2000, R version 1.0.0 was released and the rest is history.

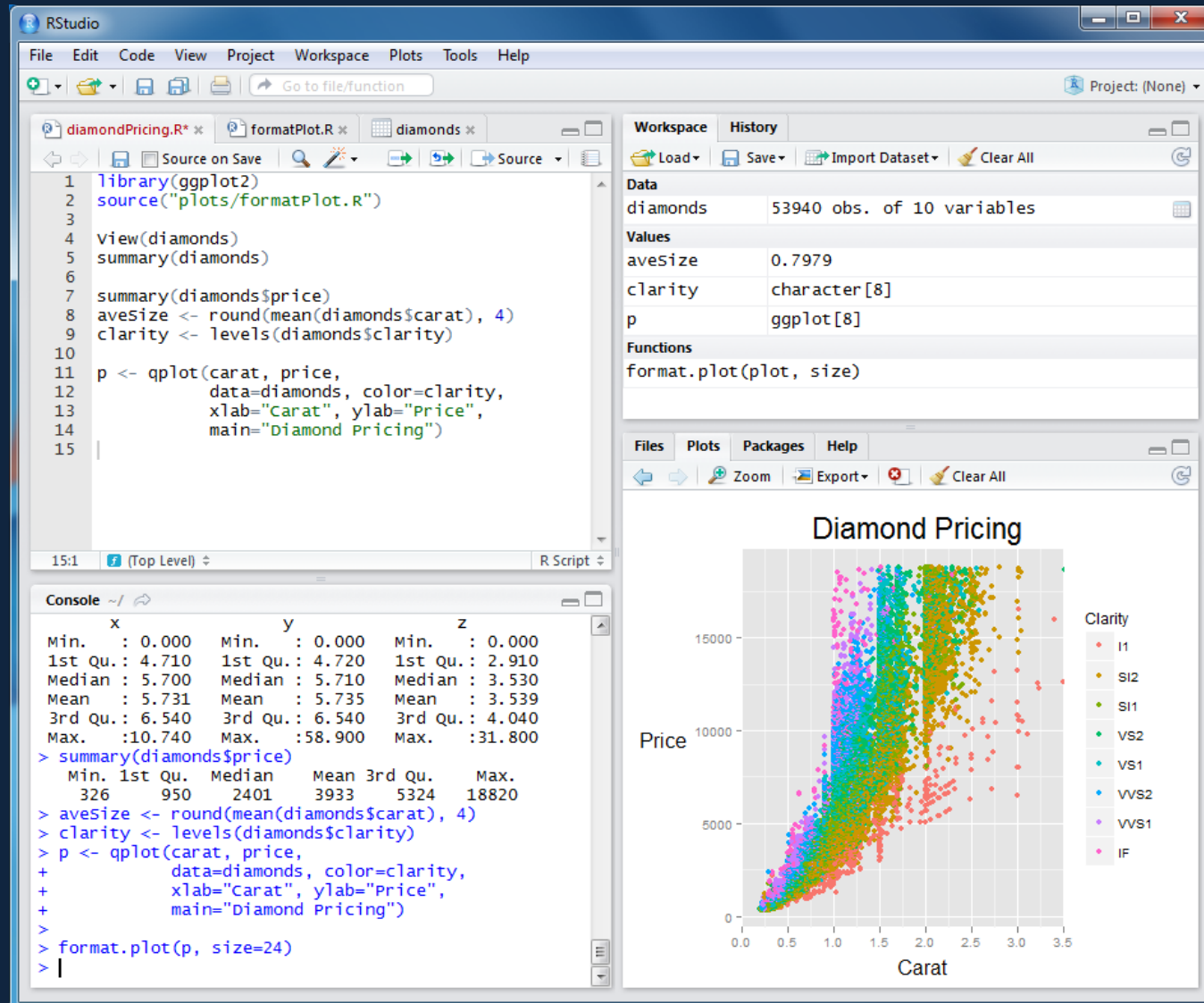


About

- R is an open-source and freely accessible software language under the GNU *General Public License, version 2*
- R works with Windows, Macintosh, Unix, and Linux operating systems.
- R is highly extensible by allowing researchers and practitioners to create their own custom packages and functions.
- It has a nice balance of object-oriented and functional programming constructs ([Hornick & Plunkett, 2013](#)).
- As of 2014 there were 5800 available user-developed packages
- **Consistently ranked as one of the top tools used by analytics professionals today**

R Studio

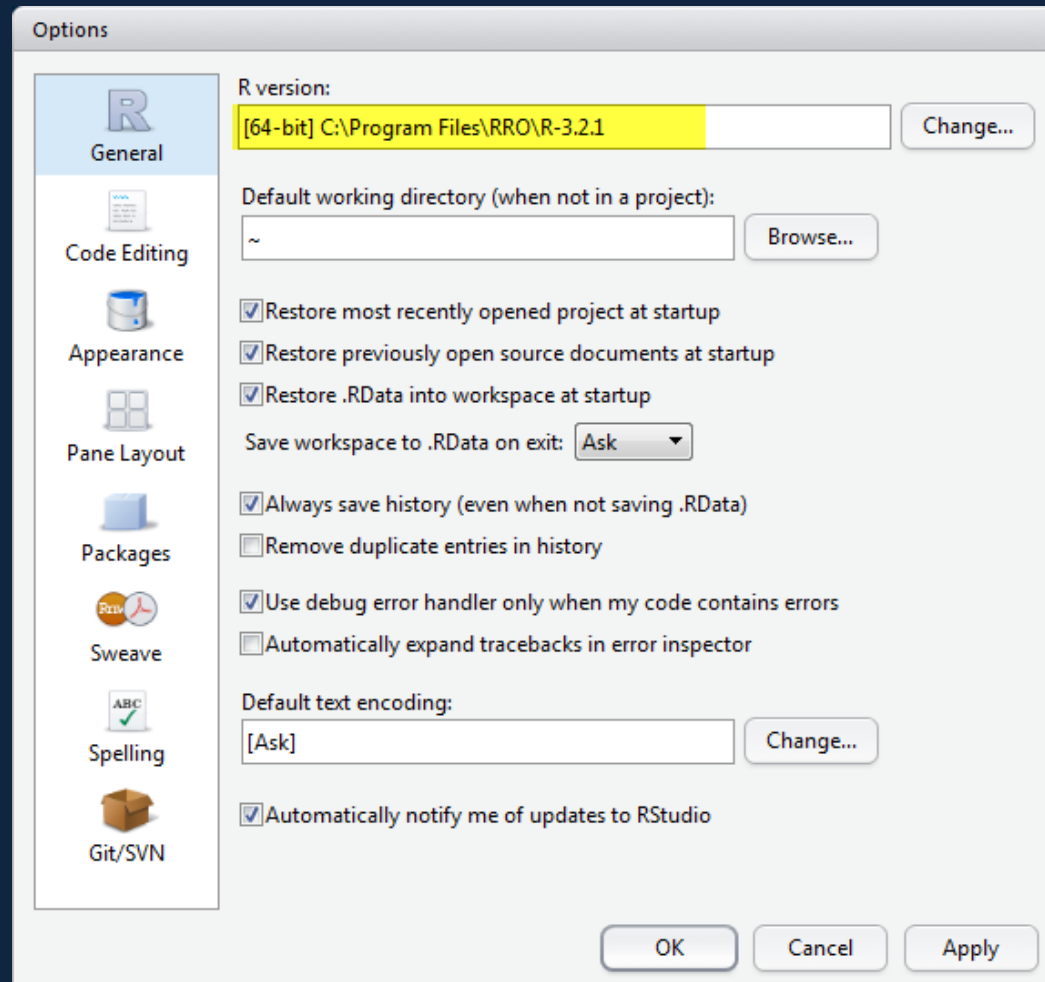
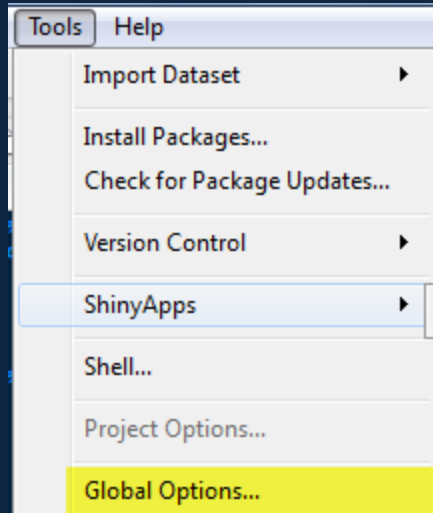
R Studio is an Integrated Development Environment (IDE) for R



Revolution Analytics


Revolution Analytics offers their own free version of R which runs faster

- Revolution R Open (RRO)
- <http://www.revolutionanalytics.com/revolution-r-open>



Revolution Analytics

If you choose to use RRO, the only thing you'll really notice aside from increased speed is a slightly different startup at the console

Console ~/ 

```
R version 3.2.1 (2015-06-18) -- "world-Famous Astronaut"  
Copyright (C) 2015 The R Foundation for Statistical Computing  
Platform: x86_64-w64-mingw32/x64 (64-bit)
```

```
R is free software and comes with ABSOLUTELY NO WARRANTY.  
You are welcome to redistribute it under certain conditions.  
Type 'license()' or 'licence()' for distribution details.
```

```
R is a collaborative project with many contributors.  
Type 'contributors()' for more information and  
'citation()' on how to cite R or R packages in publications.
```

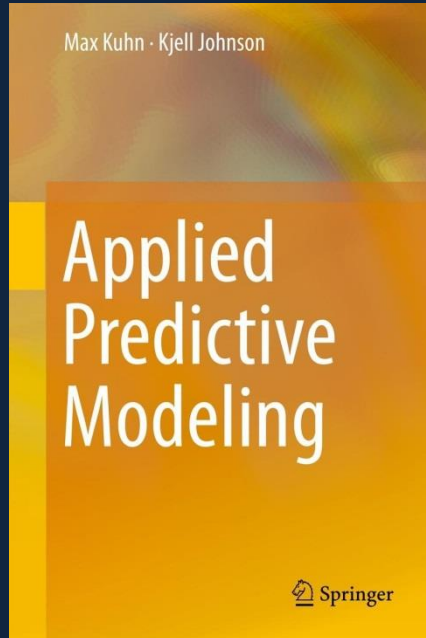
```
Type 'demo()' for some demos, 'help()' for on-line help, or  
'help.start()' for an HTML browser interface to help.  
Type 'q()' to quit R.
```

```
Revolution R Open 3.2.1  
Default CRAN mirror snapshot taken on 2015-07-01  
The enhanced R distribution from Revolution Analytics  
Visit mran.revolutionanalytics.com/open for information  
about additional features.
```

```
Multithreaded BLAS/LAPACK libraries detected. Using 4 cores for math algorithms.  
[workspace loaded from ~/.RData]
```

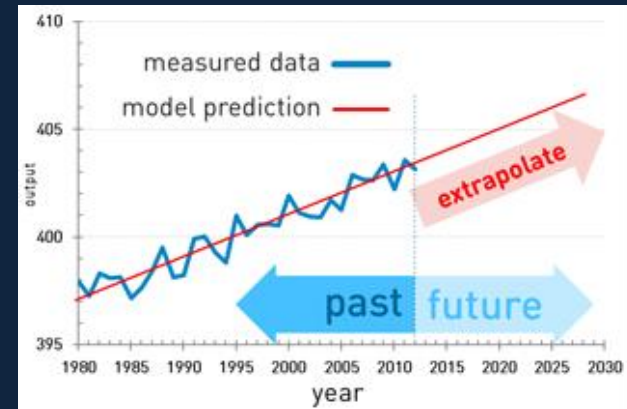
Predictive Modeling

Predictive analytics is the process of building a model that predicts some output or estimates some unknown parameter(s). Predictive analytics is synonymous with predictive modeling, which has associations with machine learning, pattern recognition, as well as data mining ([M. Kuhn & Johnson, 2013](#)).



Predictive Analytics

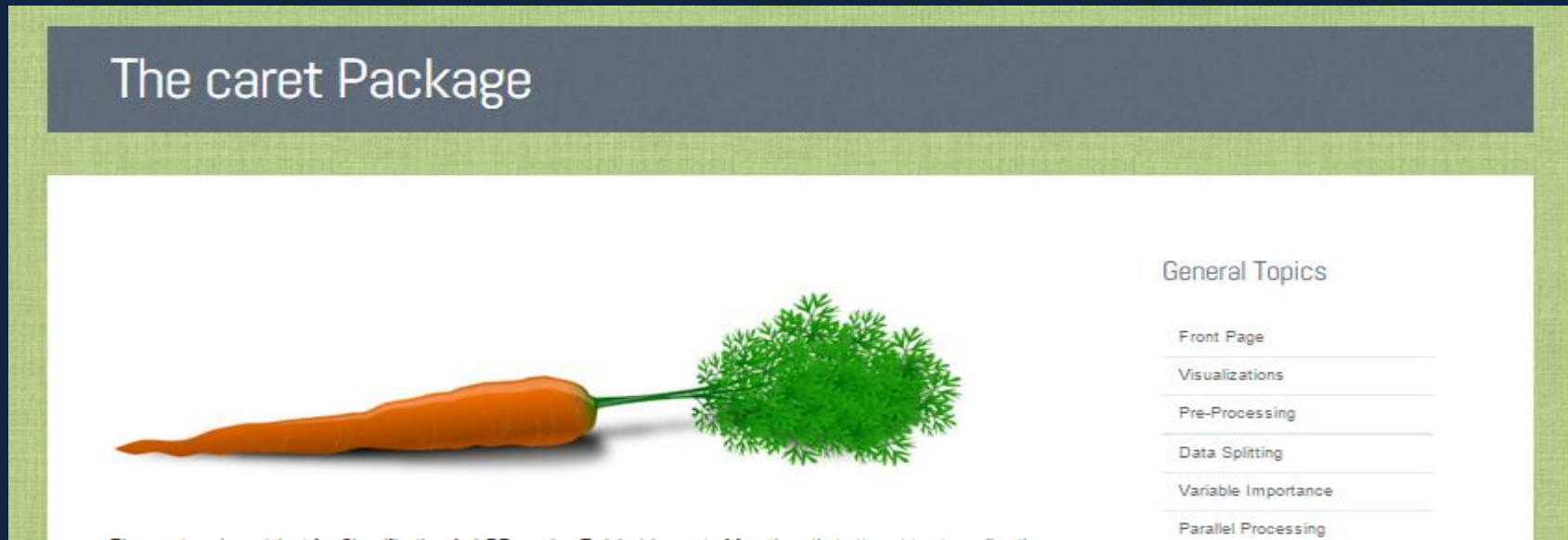
What will happen?



While predictive analytics may have the terminal goal of estimating parameters as accurately as possible, it is more a model development process than specific techniques used ([M. Kuhn & Johnson, 2013](#)).

The CARET Package

<http://topepo.github.io/caret/>



*“The **caret** package (short for **C**lassification **A**nd **R**egression **T**raining) is a set of functions that attempt to **streamline the process for creating predictive models [in R]**.”*

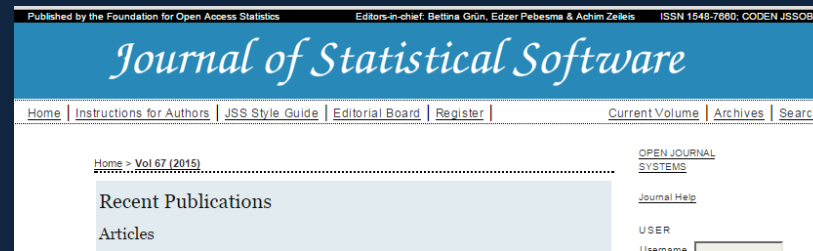
The package contains tools for:

- data splitting/partitioning
- pre-processing
- feature selection
- model tuning using resampling (**what I’m talking about today**)
- variable importance estimation

The CARET Package

There are many packages for predictive modeling. New algorithms are researched, peer-reviewed, and published every year. Often those researchers will develop a working package of their model for use by others on CRAN R.

- Check out [JSS](#) for the state-of-the-art



- New packages get updated often, can check here [CRAN R](#) to see latest contributions

The screenshot shows a web browser displaying the CRAN website. The URL is https://cran.r-project.org/web/packages/available_packages_by_date.html. The page title is "Available CRAN Packages By Date of Publication". Below the title is a table with three columns: Date, Package, and Title. The table lists the most recent packages published on CRAN.

Date	Package	Title
2015-10-09	nhanesA	NHANES Data Retrieval
2015-10-08	assertive	Readable Check Functions to Ensure Code Integrity
2015-10-08	assertive.base	A Lightweight Core of the 'assertive' Package
2015-10-08	BIFIESurvey	Tools for Survey Statistics in Educational Assessment

- R is already viewed as having a flat learning curve (i.e. challenging), so those attempting to teach R using several different modeling methodologies would probably have a bad experience – **Using the CARET package would be ideal**

Many Models with Different Syntax

In addition the functionality provided by CARET, the package is essentially a wrapper for most of the individual predictive modeling packages .

Currently there are **192** different modeling packages that it includes!! Meaning you don't have to use all the different syntax structures for each model you want to try.

This wrapper functionality is exactly what R Users/Teachers need because many of the modeling packages are written by different people. There is no pre-specified design pattern so there are inconsistencies in how one sets functional inputs and what is returned in the output.

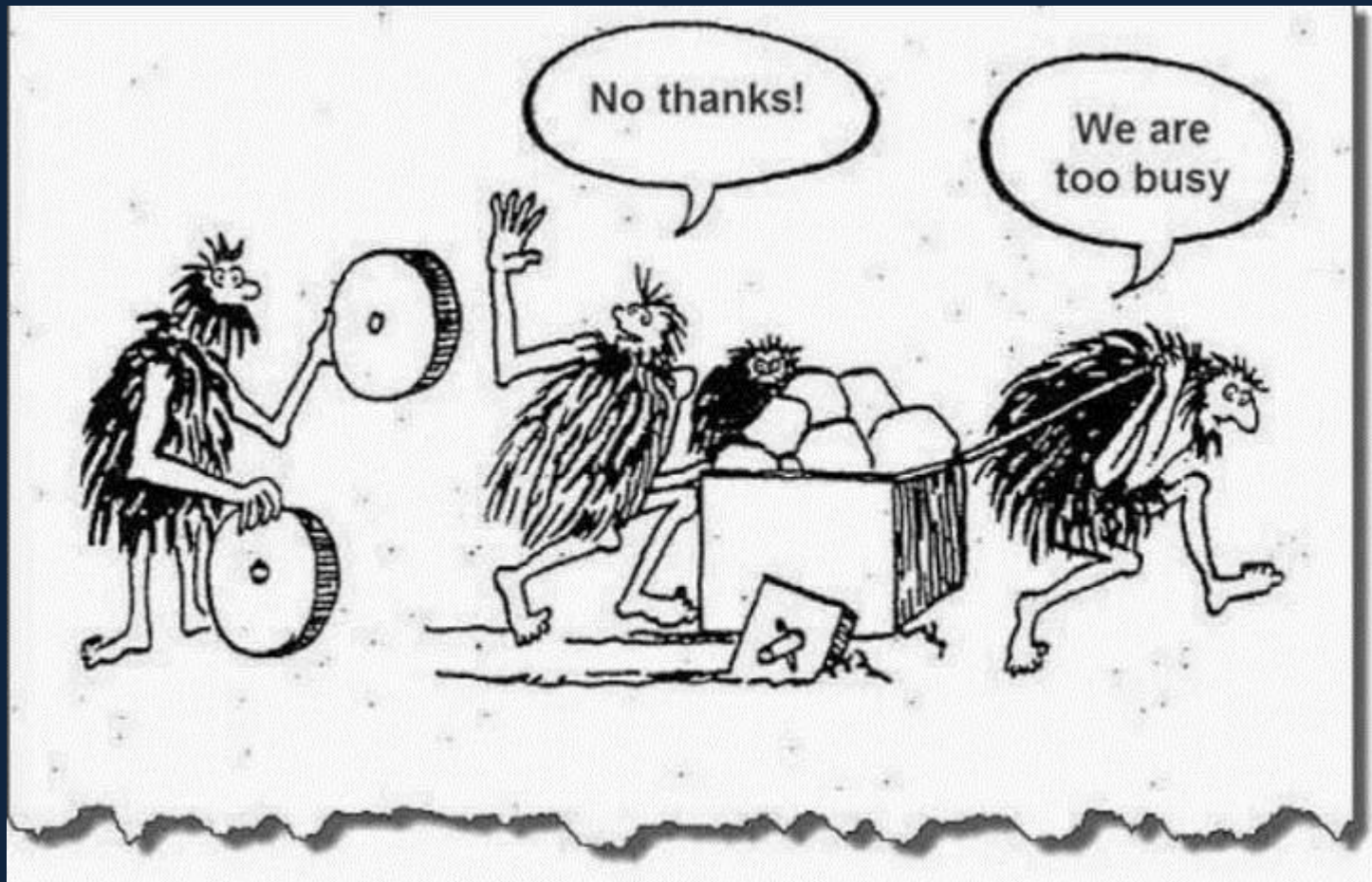
Syntax Examples

obj Class	Package	predict Function Syntax
lda	MASS	<code>predict(obj)</code> (no options needed)
glm	stats	<code>predict(obj, type = "response")</code>
gbm	gbm	<code>predict(obj, type = "response", n.trees)</code>
mda	mda	<code>predict(obj, type = "posterior")</code>
rpart	rpart	<code>predict(obj, type = "prob")</code>
Weka	RWeka	<code>predict(obj, type = "probability")</code>
LogitBoost	caTools	<code>predict(obj, type = "raw", nIter)</code>

Streamlined Modeling

With the nice wrapper functionality, this allows a more streamlined predictive modeling process to evaluate many modeling methodologies.

It will also be much efficient way to tune each model's parameters.



Example

Predicting customer churn for a telecom service based on account information.

```
library(C50)
data(churn)
str(churnTrain)
```

```
'data.frame': 3751 obs. of 20 variables:
 $ state          : Factor w/ 51 levels "AK","AL","AR",...: 36 32 36 37 2 20 25 19 50 16 ...
 $ account_length : int 107 137 84 75 118 121 147 117 141 65 ...
 $ area_code      : Factor w/ 3 levels "area_code_408",...: 2 2 1 2 3 3 2 1 2 2 ...
 $ international_plan : Factor w/ 2 levels "no","yes": 1 1 2 2 2 1 2 1 2 1 ...
 $ voice_mail_plan  : Factor w/ 2 levels "no","yes": 2 1 1 1 1 2 1 1 2 1 ...
 $ number_vmail_messages : int 26 0 0 0 0 24 0 0 37 0 ...
 $ total_day_minutes : num 162 243 299 167 223 ...
 $ total_day_calls   : int 123 114 71 113 98 88 79 97 84 137 ...
 $ total_day_charge  : num 27.5 41.4 50.9 28.3 38 ...
 $ total_eve_minutes : num 195.5 121.2 61.9 148.3 220.6 ...
 $ total_eve_calls   : int 103 110 88 122 101 108 94 80 111 83 ...
 $ total_eve_charge  : num 16.62 10.3 5.26 12.61 18.75 ...
 $ total_night_minutes : num 254 163 197 187 204 ...
 $ total_night_calls  : int 103 104 89 121 118 118 96 90 97 111 ...
 $ total_night_charge : num 11.45 7.32 8.86 8.41 9.18 ...
 $ total_intl_minutes : num 13.7 12.2 6.6 10.1 6.3 7.5 7.1 8.7 11.2 12.7 ...
 $ total_intl_calls   : int 3 5 7 3 6 7 6 4 5 6 ...
 $ total_intl_charge  : num 3.7 3.29 1.78 2.73 1.7 2.03 1.92 2.35 3.02 3.43 ...
 $ number_customer_service_calls : int 1 0 2 3 0 3 0 1 0 4 ...
 $ churn             : Factor w/ 2 levels "yes","no": 2 2 2 2 2 2 2 2 2 1 ...
```

The Predictors and The Response

A list of predictor names we might need later

```
predictors = names(churnTrain)[names(churnTrain) != "churn"]
```

Caret treats the first/leading factor as the factor to be modeled. Some packages treat the second factor as the leading factor, and this is accounted for within caret.

In our case, the nominal probabilities are predicting the occurrence of the “yes events” or churns

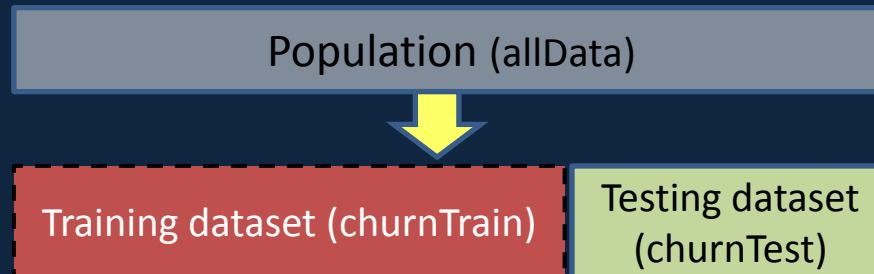
Data Partitioning

Here, we are going to perform a stratified random split

```
allData = rbind(churnTrain, churnTest)

set.seed(1)
inTrainingSet = createDataPartition(allData$churn, p = .75, list = FALSE)
churnTrain = allData[ inTrainingSet,]
churnTest = allData[-inTrainingSet,]
```

What is going on here is the data that is both the training and test sets will both have approximately the same class imbalance



```
table(allData$churn)["yes"]/(table(allData$churn)["yes"]+table(allData$churn)["no"])
# yes
# 0.1414
table(allData$churn)["no"]/(table(allData$churn)["yes"]+table(allData$churn)["no"])
# no
# 0.8586
```

Data Pre-Processing

preProcess() calculates values that can be used to apply to any data set (e.g. training set, testing set, etc.)

Typical Methods (list continues to grow):

- Centering
- Scaling
- Spatial sign transformation
- PCA
- Imputation
- Box-Cox transformation
-

```
numerics = c("account_length", "total_day_calls", "total_night_calls")
head(churnTrain[,numerics])
## Determine means and standard deviations
procValues = preProcess(churnTrain[,numerics],
                        method = c("center", "scale", "YeoJohnson"))

## Use the predict methods to do the adjustments
trainScaled = predict(procValues, churnTrain[,numerics])
testScaled = predict(procValues, churnTest[,numerics])
```

Data Pre-Processing

procValues is a preProcess object

```
> procValues  
  
Call:  
preProcess.default(x = churnTrain[, numerics], method = c("center", "scale", "YeoJohnson"))  
  
Created from 3751 samples and 3 variables  
Pre-processing: centered, scaled, Yeo-Johnson transformation  
  
Lambda estimates for Yeo-Johnson transformation:  
0.89, 1.18, 1.08
```

The preProcess function can also be called within other functions for each resampling iteration (upcoming..)

Boosted Trees – Classic adaBoost Algorithm

Boosting

Boosting is a method to “boost” weak learning algorithms (e.g. lone classification tree) into strong learning algorithms.

Idea

Boosted trees try to improve the model fit over different trees by considering past fits (similar to iterative reweighted least squares)

Tree Boosting Algorithm

```
Initialize equal weights per sample;  
for  $j = 1 \dots M$  iterations do  
  Fit a classification tree using sample weights (denote the model  
  equation as  $f_j(x)$ );  
  forall the misclassified samples do  
    | increase sample weight  
  end  
  Save a “stage-weight” ( $\beta_j$ ) based on the performance of the current  
  model;  
end
```

Boosted Trees – Classic adaBoost Algorithm

Formulation

Here, the categorical response y_i is coded as $\{-1, 1\}$ and the model $f_j(x)$ produces values of $\{-1, 1\}$

Final Prediction

The final prediction is obtained by first predicting using all M trees, then weighting each prediction, so

$$f(x) = \frac{1}{M} \sum_{j=1}^M \beta_j f_j(x)$$

where f_j is the j th tree fit and β_j is the stage weight for that tree

Thus, the final class is determined by the sign of the model prediction.

Lay terms

The final prediction generated is a weighted average of each tree's respective prediction, and the weights are set based on the quality (i.e. accuracy) of each tree

Boosted Tree Parameters

Tuning parameters

Most implementations of boosting have three tuning parameters (could be more)

1. # of boosting iterations (i.e. # of trees)
2. Tree complexity (i.e. # of tree splits)
3. Shrinkage (i.e. the learning rate or how quickly the algorithm adapts)

Boosting functions in R Examples:

- `gbm()` in `gbm` package
- `ada()` in `ada` package
- `blackboost()` in `mboost` package

How do you know?

<http://topepo.github.io/caret/modelList.html>

<http://topepo.github.io/caret/Boosting.html>

Stochastic Gradient Boosting

```
method = 'gbm'
```

Type: Regression, Classification

Tuning Parameters: `n.trees` (# Boosting Iterations), `interaction.depth` (Max Tree Depth), `shrinkage` (Shrinkage), `n.minobsinnode` (Min. Terminal Node Size)

Using the gbm Package

Tuning parameters

The `gbm()` in the `gbm` package can be used to fit the model if you like, then you can use `predict.gbm` or other functions to predict and evaluate the model

Example:

```
> library(gbm)
> # The gbm function does not accept factor response values so we
> # will make a copy and modify the outcome variable
> forGBM <- churnTrain
> forGBM$churn <- ifelse(forGBM$churn == "yes", 1, 0)
>
> gbmFit <- gbm(formula = churn ~ .,          # Use all predictors
+               distribution = "bernoulli",  # For classification
+               data = forGBM,
+               n.trees = 2000,              # 2000 boosting iterations
+               interaction.depth = 7,       # How many splits in each tree
+               shrinkage = 0.01,           # learning rate
+               verbose = FALSE)            # Do not print the details
```

Tuning a gbm Model

Tuning parameters

The previous code assumed that we know appropriate values for the tuning parameters. Do we?

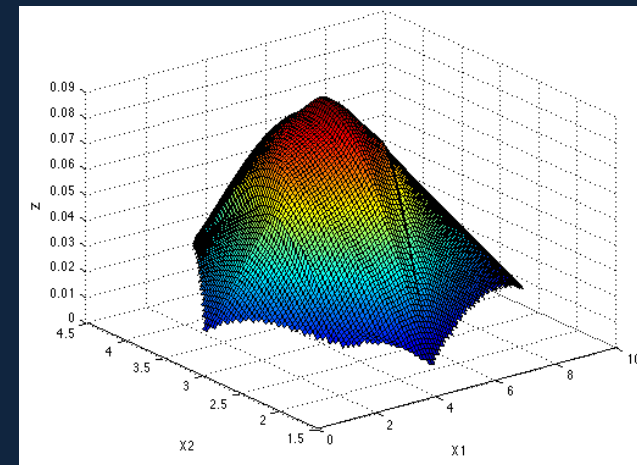
One nice approach for model tuning is **resampling**, which we'll do soon.

We can fit models with different values of the tuning parameters to many resampled versions of the training set and estimate the performance based on the hold-out samples.

From this, a profile of performance can be obtained across different gbm models, and then an “optimal” set of tuning parameters can be chosen.

Idea

- Goal is to get the resampled estimate of performance
- For 3 tuning parameters, can set up a 3-D grid and look at the performance profile across the grid to derive what should be used as the final tuning parameters



Tuning a gbm Model

Tuning

- This is basically a modified cross-validation approach

```
foreach resampled data set do  
  Hold-out samples ;  
  foreach combination of tree depth, learning rate and number of trees  
  do  
    Fit the model on the resampled data set;  
    Predict the hold-outs and save results;  
  end  
  Calculate the average AUC ROC across the hold-out sets of predictions  
end  
Determine tuning parameters based on the highest resampled ROC AUC;
```

The CARET Way - Using train()

Training

- Using the train(), we can specify the response, predictor set, and the modeling technique we want (e.g. boosting via the gbm package)

```
> gbmTune <- train(x = churnTrain[,predictors],  
+                  y= churnTrain$churn,  
+                  method = "gbm")  
>  
> # or, using the formula interface  
> gbmTune <- train(churn ~ ., data = churnTrain, method = "gbm")
```

- Since we have binary outcomes, train() models the probability of the first factor level. This is “yes” in our case for the churn data set.
 - gbm knows we are doing classification because we have a binary factor
 - If we have a numeric response, it would assume we were doing regression type modeling, which is possible using gbm

Stochastic Gradient Boosting

method = 'gbm'

Type: Regression, Classification

Tuning Parameters: n.trees (# Boosting Iterations), interaction.depth (Max Tree Depth), shrinkage (Shrinkage), n.minobsinnode (Min. Terminal Node Size)

Passing Parameters

We can pass options through train to gbm using the three dots...

A good idea to use verbose = FALSE within train() for the gbm method because there is a lot of logging that will be returned

```
gbmTune <- train(churn ~ ., data = churnTrain,  
                 method = "gbm",  
                 verbose = FALSE)
```

To Note:

- **gbm()** is the underlying function, **train()** is the top level function
- Verbose=T is not an argument for the train(), but rather used for the underlying gbm() so the output isn't printed out every time

Changing the Resampling Method

By Default, `train()` uses the bootstrap for resampling.

Repeated cross-validation “repeatedcv” has nice bias-variance properties, so we’ll use 5 repeats of 10-fold cross-validation there are 50 resampled data sets being evaluated.

`trControl` allows us to easily set parameters, then add `trControl` as an argument to the `train()`

```
ctrl <- trainControl(method = "repeatedcv",  
                     repeats = 5)  
  
gbmTune <- train(churn ~ ., data = churnTrain,  
                 method = "gbm",  
                 verbose = FALSE,  
                 trControl = ctrl)
```

Different Performance Metrics

What is going to happen next is:

1. A sequence of different gbm models will be fit
2. Performance is estimated using resampling
3. The best gbm model is chosen based on best performance

What do you define as best performance?

The default metrics for classification problems are accuracy and Cohen's Kappa. Since the data is unbalanced, sensitivity might not be the best statistic to use for evaluation.

However, suppose we wanted to estimate sensitivity, specificity, and area under the ROC curve (thus select best model using greatest AUC)

We'll need to specify that within `train()` so that class probabilities are produced, then estimate these statistics (i.e. AUC) to get a rank of the models.

Different Performance Metrics

The **twoClassSummary()** defined in caret calculates the sensitivity, specificity, and AUC. Custom functions can also be used (check out ?train to learn more)

```
ctrl <- trainControl(method = "repeatedcv", repeats = 5,  
                     classProbs = TRUE,  
                     summaryFunction = twoClassSummary)  
  
gbmTune <- train(churn ~ ., data = churnTrain,  
                 method = "gbm",  
                 metric = "ROC",  
                 verbose = FALSE,  
                 trControl = ctrl)
```

In **trainControl()** can specify if you want class probabilities – not all models will provide these, so this is really useful

The **train()** will want to know which one do you want to optimize for?? Thus, you'll need to specify the metric. In this churn case, metric = "ROC". However, If you were building regression models, you could use metric = "RMSE"

Specifying the Search Grid

By default, `train()` uses a minimal search grid: 3 values per tuning parameter

We can easily expand the scope of this if you like, so our models can test more combinations of tuning parameters

Here we'll look at:

1. Tree depth from 1 to 7
2. Boosting iterations from 100 to 1,000
3. Learning rates: 0.01 (slow), 0.10 (fast)
4. Minimum number of nodes: 10

When we run our experiment, we'll save our results in a `data.frame`

Specifying the Search Grid

Adding in `tuneGrid` to your `train()` will allow you to specify all the combinations of tuning parameters you want to profile.

Note that if you didn't specify your `tuneGrid`, it would only look at 3 different levels for each of the tuning parameters by DEFAULT

```
ctrl <- trainControl(method = "repeatedcv", repeats = 5,  
                     classProbs = TRUE,  
                     summaryFunction = twoClassSummary)  
  
grid <- expand.grid(interaction.depth = seq(1, 7, by = 2),  
                  n.trees = seq(100, 1000, by = 50),  
                  shrinkage = c(0.01, 0.1))  
  
gbmTune <- train(churn ~ ., data = churnTrain,  
                 method = "gbm",  
                 metric = "ROC",  
                 tuneGrid = grid,  
                 verbose = FALSE,  
                 trControl = ctrl)
```

Running our Model

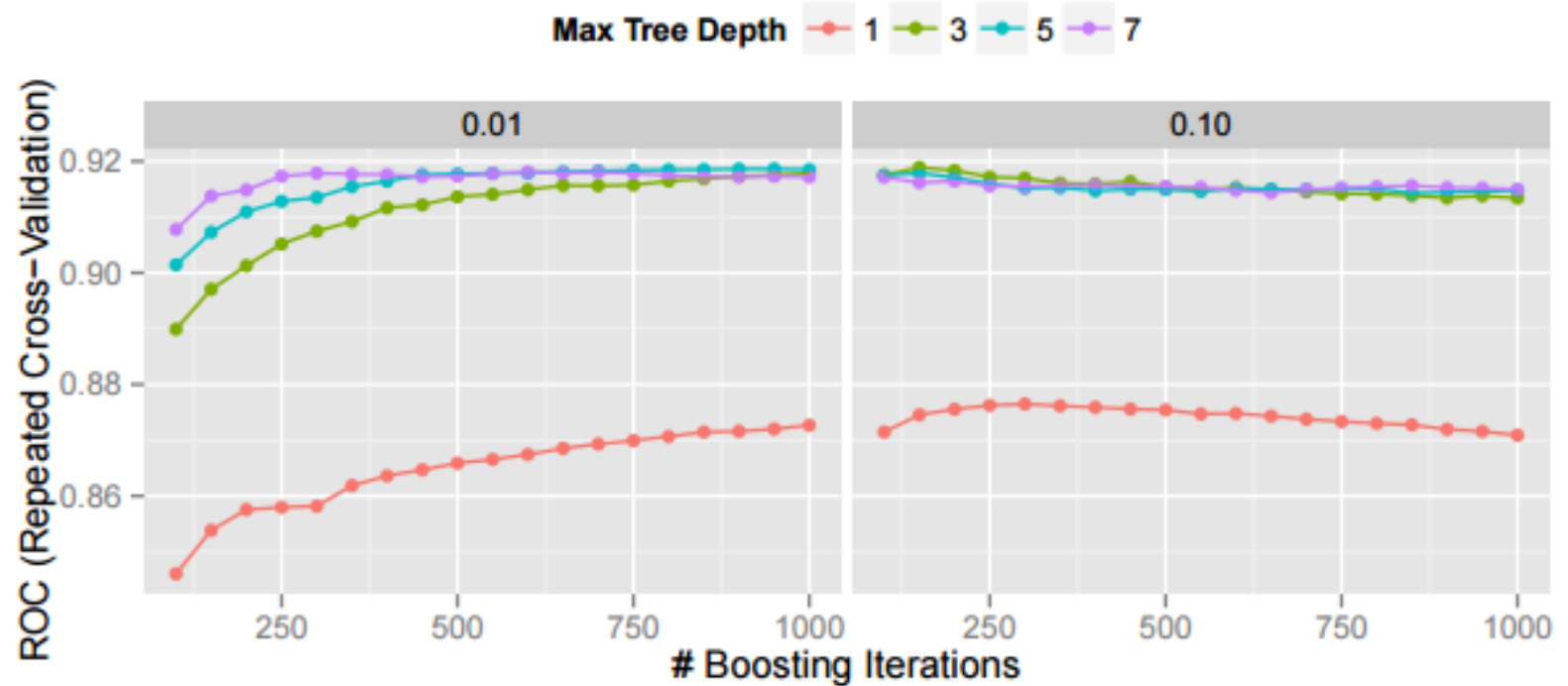
```
> grid <- expand.grid(interaction.depth = seq(1, 7, by = 2),  
+                     n.trees = seq(100, 1000, by = 50),  
+                     shrinkage = c(0.01, 0.1))  
>  
> ctrl <- trainControl(method = "repeatedcv", repeats = 5,  
+                      summaryFunction = twoClassSummary,  
+                      classProbs = TRUE)  
>  
> set.seed(1)  
> gbmTune <- train(churn ~ ., data = churnTrain,  
+                  method = "gbm",  
+                  metric = "ROC",  
+                  tuneGrid = grid,  
+                  verbose = FALSE,  
+                  trControl = ctrl)
```

Once the model completes, we can print the object's output

For each tuning parameter it gives the averaged AUC. If you don't tell it what you want it to do, it will give you the "winner" among the set (i.e. whatever the max AUC is), and that model's corresponding tuning parameters

Evaluating our Results

```
> ggplot(gbmTune) + theme(legend.position = "top")
```



Prediction and Performance Assessment

The `predict()` can be used to get results for other data sets

Predicted Classes

```
> gbmPred <- predict(gbmTune, churnTest)
> str(gbmPred)

Factor w/ 2 levels "yes","no": 2 2 2 2 2 2 2 2 2 2 ...
```

Predicted Probabilities

```
> gbmProbs <- predict(gbmTune, churnTest, type = "prob")
> str(gbmProbs)

'data.frame': 1667 obs. of 2 variables:
 $ yes: num  0.0316 0.092 0.3086 0.0307 0.022 ...
 $ no : num  0.968 0.908 0.691 0.969 0.978 ...
```

Prediction and Performance Assessment

confusionMatrix()

```
> confusionMatrix(gbmPred, churnTest$churn)
```

Confusion Matrix and Statistics

	Reference	
Prediction	yes	no
yes	151	8
no	73	1435

Accuracy : 0.951

95% CI : (0.94, 0.961)

No Information Rate : 0.866

P-Value [Acc > NIR] : < 2e-16

Kappa : 0.762

Mcnemar's Test P-Value : 1.15e-12

Sensitivity : 0.6741

Specificity : 0.9945

Pos Pred Value : 0.9497

Neg Pred Value : 0.9516

Prevalence : 0.1344

Detection Rate : 0.0906

Detection Prevalence : 0.0954

Test Set ROC (using pROC package here)

```
> rocCurve <- roc(response = churnTest$churn,  
+                 predictor = gbmProbs[, "yes"],  
+                 levels = rev(levels(churnTest$churn)))  
> rocCurve
```

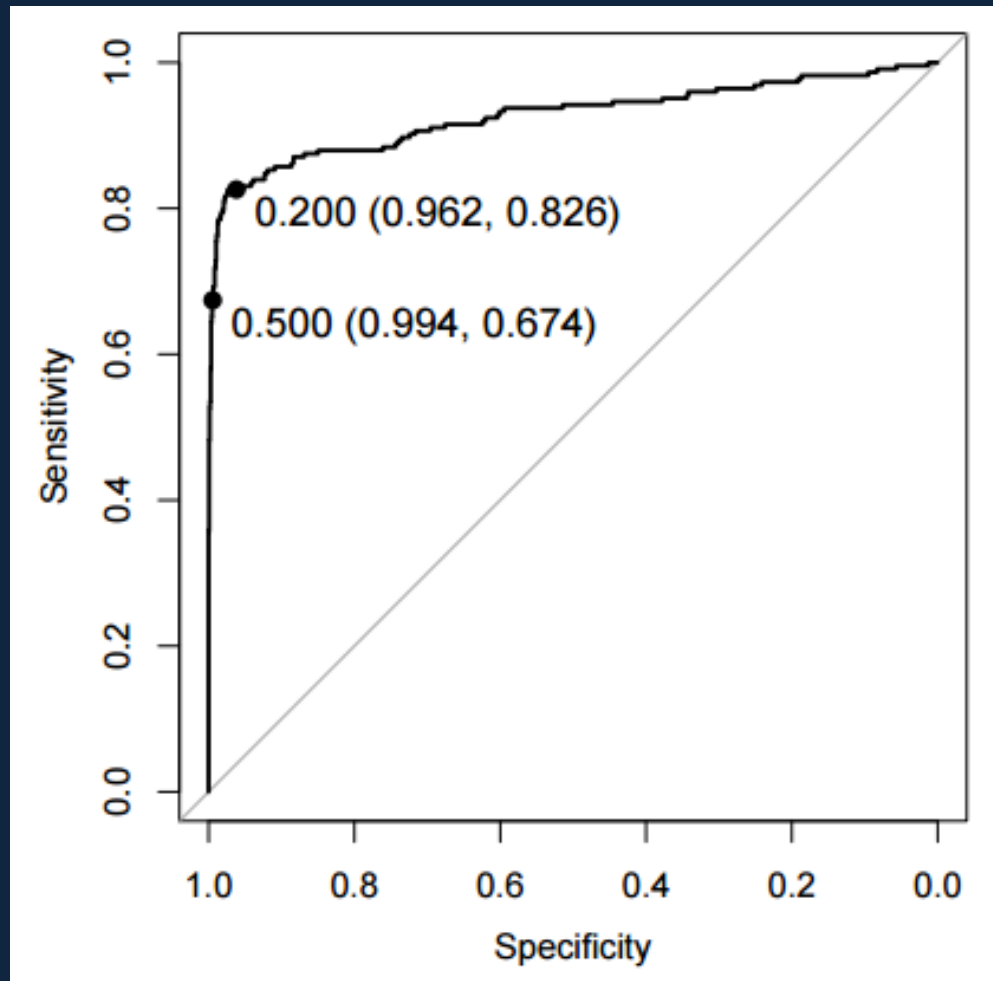
Call:

```
roc.default(response = churnTest$churn, predictor = gbmProbs[,
```

Data: gbmProbs[, "yes"] in 1443 controls (churnTest\$churn no) < 224

Area under the curve: 0.927

Test Set ROC (using pROC package here)



That's it!

If you're still interested, check out for much more information

<http://topepo.github.io/caret/>

