R - General Purpose Machine Learning Scripts

Manuel Amunategui

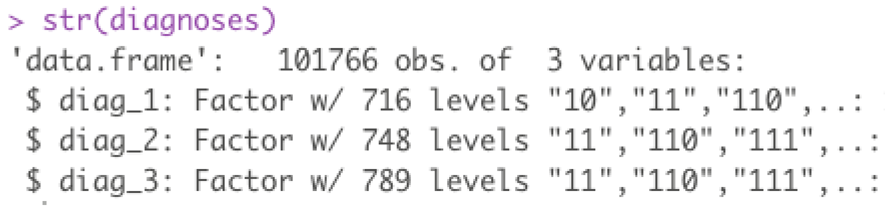
Portland, OR

March 2015

Contents

Feature Hashing (a.k.a. The Hashing Trick) With R

February 21, 2015 *Tags:* modeling



```
> str(diagnoses)
'data.frame':   101766 obs. of  3 variables:
 $ diag_1: Factor w/ 716 levels "10","11","110",..:
 $ diag_2: Factor w/ 748 levels "11","110","111",..:
 $ diag_3: Factor w/ 789 levels "11","110","111",..:
```

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{FeatureHashing}** - Creates a Model Matrix via Feature Hashing With a Formula Interface
- **{RCurl}** - General network (HTTP/FTP/...) client interface for R
- **{caret}** - Classification and Regression Training
- **{glmnet}** - Lasso and elastic-net regularized generalized linear models

Feature hashing is a clever way of modeling data sets containing large amounts of factor and character data. It uses less memory and requires little pre-processing. In this walkthrough, we model a large healthcare data set by first using **dummy variables** and then **feature hashing**.

What's the big deal? Well, normally, one would:

- **Dummify all factor, text, and unordered categorical data**: This creates a new column for each unique value and tags a binary value whether or not an observation contains that particular value. For large data sets, this can drastically increase the dimensional space (adding many more columns).
- **Drop levels**: For example, taking the the top x% most popular levels and neutralizing the rest, grouping levels by theme using **string distance**, or simply ignoring factors too large for a machine's memory.
- **Use a sparse matrix**: This can mitigate the size of these dummied data sets by dropping zeros.

But a more complete solution, especially when there are tens of thousands of unique values, is the **‘hashing trick’**.

Wush Wu created the FeatureHashing package available on CRAN. According to the package’s introduction on CRAN:

"Feature hashing, also called the hashing trick, is a method to transform features to vector. Without looking up the indices in an associative array, it applies a hash function to the features and uses their hash values as indices directly. The method of feature hashing in this package was proposed in Weinberger et. al. (2009). The hashing algorithm is the murmurhash3 from the digest package. Please see the README.md for more information.”

Feature hashing has numerous advantages in modeling and machine learning. It works with address locations instead of actual data, this allows it to process data only when needed. So, the first feature found is really a column of data containing only one level (one value), when it encounters a different value, then it becomes a feature with 2 levels, and so on. It also requires no pre-processing of factor data; you just feed it your factor in its raw state. This approach takes a lot less memory than a fully scanned and processed data set. Plenty oftheory out there for those who want a deeper understanding.

Some of its disadvantages include causing models to run slower and a certain obfuscation of the data.

Let’s Code!

We'll be using a great healthcare data set on historical readmissions of patients with diabetes - Diabetes 130-US hospitals for years 1999-2008 Data Set. Readmissions is a big deal for hospitals in the US as Medicare/medicaid will scrutinize those bills and, in some cases, only reimburse a percentage of them. We’ll use code to automate the download and unzipping of the data directly from the UC Irvine Machine Learning Repository.

```
require(RCurl)
binData <- getBinaryURL("https://archive.ics.uci.edu/ml/machine-learning-databases/00296/dataset_diabetes.zip",
                    ssl.verifypeer=FALSE)

conObj <- file("dataset_diabetes.zip", open = "wb")
writeBin(binData, conObj)
# don't fortet to close it
close(conObj)
```

```
# open diabetes file

files <- unzip("dataset_diabetes.zip")

diabetes <- read.csv(files[1], stringsAsFactors = FALSE)
```

Let's take a quick look at the data and then clean it up.

```
str(diabetes)

## 'data.frame':    101766 obs. of  50 variables:

##  $ encounter_id            : int  2278392 149190 64410 500364 16680 35754 55842 63768 12522
 15738 ...

##  $ patient_nbr             : int  8222157 55629189 86047875 82442376 42519267 82637451 8425
9809 114882984 48330783 63555939 ...

##  $ race                    : chr  "Caucasian" "Caucasian" "AfricanAmerican" "Caucasian" ...

##  $ gender                  : chr  "Female" "Female" "Female" "Male" ...

##  $ age                     : chr  "[0-10)" "[10-20)" "[20-30)" "[30-40)" ...

##  $ weight                  : chr  "?" "?" "?" "?" ...

##  $ admission_type_id       : int  6 1 1 1 1 2 3 1 2 3 ...

##  $ discharge_disposition_id: int  25 1 1 1 1 1 1 1 1 3 ...

##  $ admission_source_id     : int  1 7 7 7 7 2 2 7 4 4 ...

##  $ time_in_hospital        : int  1 3 2 2 1 3 4 5 13 12 ...

##  $ payer_code              : chr  "?" "?" "?" "?" ...

##  $ medical_specialty       : chr  "Pediatrics-Endocrinology" "?" "?" "?" ...

##  $ num_lab_procedures      : int  41 59 11 44 51 31 70 73 68 33 ...

##  $ num_procedures          : int  0 0 5 1 0 6 1 0 2 3 ...

##  $ num_medications         : int  1 18 13 16 8 16 21 12 28 18 ...

##  $ number_outpatient       : int  0 0 2 0 0 0 0 0 0 0 ...

##  $ number_emergency        : int  0 0 0 0 0 0 0 0 0 0 ...

##  $ number_inpatient        : int  0 0 1 0 0 0 0 0 0 0 ...

##  $ diag_1                  : chr  "250.83" "276" "648" "8" ...

##  $ diag_2                  : chr  "?" "250.01" "250" "250.43" ...
##  $ diag_3                  : chr  "?" "255" "V27" "403" ...
##  $ number_diagnoses        : int  1 9 6 7 5 9 7 8 8 8 ...
##  $ max_glu_serum           : chr  "None" "None" "None" "None" ...
##  $ A1Cresult               : chr  "None" "None" "None" "None" ...
##  $ metformin               : chr  "No" "No" "No" "No" ...
##  $ repaglinide             : chr  "No" "No" "No" "No" ...
##  $ nateglinide             : chr  "No" "No" "No" "No" ...
##  $ chlorpropamide          : chr  "No" "No" "No" "No" ...
##  $ glimepiride             : chr  "No" "No" "No" "No" ...
##  $ acetohexamide           : chr  "No" "No" "No" "No" ...
##  $ glipizide               : chr  "No" "No" "Steady" "No" ...
##  $ glyburide               : chr  "No" "No" "No" "No" ...
##  $ tolbutamide             : chr  "No" "No" "No" "No" ...
##  $ pioglitazone            : chr  "No" "No" "No" "No" ...
##  $ rosiglitazone           : chr  "No" "No" "No" "No" ...
##  $ acarbose                : chr  "No" "No" "No" "No" ...
##  $ miglitol                : chr  "No" "No" "No" "No" ...
##  $ troglitazone            : chr  "No" "No" "No" "No" ...
##  $ tolazamide              : chr  "No" "No" "No" "No" ...
##  $ examide                 : chr  "No" "No" "No" "No" ...
##  $ citoglipton             : chr  "No" "No" "No" "No" ...
##  $ insulin                 : chr  "No" "Up" "No" "Up" ...
##  $ glyburide.metformin     : chr  "No" "No" "No" "No" ...
##  $ glipizide.metformin     : chr  "No" "No" "No" "No" ...
##  $ glimepiride.pioglitazone: chr  "No" "No" "No" "No" ...
##  $ metformin.rosiglitazone : chr  "No" "No" "No" "No" ...
##  $ metformin.pioglitazone  : chr  "No" "No" "No" "No" ...
##  $ change                  : chr  "No" "Ch" "No" "Ch" ...
##  $ diabetesMed             : chr  "No" "Yes" "Yes" "Yes" ...
##  $ readmitted              : chr  "NO" ">30" "NO" "NO" ...
```

101766 obs. of 50 variables

Of interest are 3 fields: `diag_1`, `diag_2`, `diag_3`. These 3 features are numerical representations of patient diagnoses. Each patient can have up to 3 diagnoses recorded. If we sum all the unique levels for all 3 sets, we end up with a lot of levels. Remember, even though these are numerical representations of diagnoses, they are still diagnoses and cannot be model as numerical data as the distance between two diagnoses doesn't mean anything. These all need to be dummied into there own column.

```
length(unique(diabetes$diag_1))

## [1] 717

length(unique(diabetes$diag_2))

## [1] 749

length(unique(diabetes$diag_3))

## [1] 790
```

The sum of all unique diagnoses is 2256. This means, in order to break out each diagnosis into its own column, we need an additional 2256 new columns on top of our original 50.

OK, let's finish cleaning up the data. We're going to drop non-predictive features, replace interrogation marks with `NA`s and fix the outcome variable to a binary value.

```
# drop useless variables

diabetes <- subset(diabetes,select=-c(encounter_id, patient_nbr))



# transform all "?" to 0s

diabetes[diabetes == "?"] <- NA



# remove zero variance - ty James http://stackoverflow.com/questions/8805298/quickly-remove-zero-variance-variables-from-a-data-frame

diabetes <- diabetes[sapply(diabetes, function(x) length(levels(factor(x,exclude=NULL)))>1)]



# prep outcome variable to those readmitted under 30 days

diabetes$readmitted <- ifelse(diabetes$readmitted == "<30",1,0)
```

```
outcomeName <- 'readmitted'
```

Now, let's prepare the data two ways: a common approach using **dummy variables** for our factors, and another using **feature hashing**.

Using Dummy Variables

Here we use caret's `dummyVars` function to make our dummy column (see my walkthrough for more details on this great function). **Be warned**, you will need at least 2 gigabytes of free live memory on your system for this to work!

```
diabetes_dummy <- diabetes
# warning will need 2GB at least free memory
require(caret)
dmy <- dummyVars(" ~ .", data = diabetes_dummy)
diabetes_dummy <- data.frame(predict(dmy, newdata = diabetes_dummy))
```

After all this preparation work, we end up with a fairly wide data space with 2462 features/columns:

```
dim(diabetes_dummy)
## [1] 101766   2462
```

We'll use cv.glmnet to model our data as it supports sparse matrices (this also works great with XGBoost).

```
# change all NAs to 0
diabetes_dummy[is.na(diabetes_dummy)] <- 0
# split the data into training and testing data sets
set.seed(1234)
split <- sample(nrow(diabetes_dummy), floor(0.5*nrow(diabetes_dummy)))
objTrain <-diabetes_dummy[split,]
objTest <- diabetes_dummy[-split,]

predictorNames <- setdiff(names(diabetes_dummy),outcomeName)
```

```
# cv.glmnet expects a matrix
library(glmnet)
# straight matrix model not recommended - works but very slow, go with a sparse matrix
# glmnetModel <- cv.glmnet(model.matrix(~., data=objTrain[,predictorNames]), objTrain[,outcomeName],
#              family = "binomial", type.measure = "auc")

glmnetModel <- cv.glmnet(sparse.model.matrix(~., data=objTrain[,predictorNames]), objTrain[,outcomeName],
                         family = "binomial", type.measure = "auc")
glmnetPredict <- predict(glmnetModel,sparse.model.matrix(~., data=objTest[,predictorNames]), s="lambda.min")
```

Let's look at the AUC (Area under the curve) so we can compare this approach with the feature-hashed one:

```
auc(objTest[,outcomeName], glmnetPredict)
## [1] 0.6480986
```

Using Feature Hashing

Now for the fun part, remember that wide data set we just modeled? Well, by using feature hashing, we don't have to do any of that work; we just feed the data set with its factor and character features directly into the model. The code is based on both a kaggle competition and Wush Wu's package readme on GitHub.com.

The `hashed.model.matrix` function takes a `hash_size` value. This is a critical piece. Depending on the size of your data you may need to adjust this value. I set it here to 2^{12}, but if you try a larger value, it will handle more variables (i.e. unique values). On the other hand, if you try a smaller value, you risk having memory collisions and loss of data. It is something you have to experiment with.

```
diabetes_hash <- diabetes
predictorNames <- setdiff(names(diabetes_hash),outcomeName)
```

```
# change all NAs to 0
diabetes_hash[is.na(diabetes_hash)] <- 0

set.seed(1234)
split <- sample(nrow(diabetes_hash), floor(0.5*nrow(diabetes_hash)))
objTrain <-diabetes_hash[split,]
objTest <- diabetes_hash[-split,]

library(FeatureHashing)
objTrain_hashed = hashed.model.matrix(~., data=objTrain[,predictorNames], hash_size=2^12, transpose=FALSE, keep.hashing_mapping=TRUE)
objTrain_hashed = as(objTrain_hashed, "dgCMatrix")
objTest_hashed = hashed.model.matrix(~., data=objTest[,predictorNames], hash_size=2^12, transpose=FALSE, keep.hashing_mapping=TRUE)
objTest_hashed = as(objTest_hashed, "dgCMatrix")

library(glmnet)
glmnetModel <- cv.glmnet(objTrain_hashed, objTrain[,outcomeName],
                     family = "binomial", type.measure = "auc")
```

Let's see how this version scored:

```
glmnetPredict <- predict(glmnetModel, objTest_hashed, s="lambda.min")
auc(objTest[,outcomeName], glmnetPredict)
## [1] 0.6475847
```

Practically the same score as prepping the data yourself but with half the work and a much smaller memory footprint.

Note: This subject and code was inspired by a Kaggle.com competition: Avazu - Click-Through Rate Prediction. More precisely by the following thread. I have used feature hashing in**Python** via sklearn's `preprocessing.OneHotEncoder` and `feature_extraction.FeatureHasher` but these approaches aren't as common in **R**. I hope this is changing with the arrival of great packages such as Wu's. Many thanks to Wu and all Kagglers!!

Full source code (also on GitHub):

```
# get data ----------------------------------------------------------------
# UCI Diabetes 130-US hospitals for years 1999-2008 Data Set
# https://archive.ics.uci.edu/ml/machine-learning-databases/00296/
require(RCurl)
binData <- getBinaryURL("https://archive.ics.uci.edu/ml/machine-learning-databases/00296/dataset_diabetes.
zip",
                    ssl.verifypeer=FALSE)

conObj <- file("dataset_diabetes.zip", open = "wb")
writeBin(binData, conObj)
# don't fortet to close it
close(conObj)

# open diabetes file
files <- unzip("dataset_diabetes.zip")
diabetes <- read.csv(files[1], stringsAsFactors = FALSE)

# quick look at the data
str(diabetes)

# drop useless variables
diabetes <- subset(diabetes,select=-c(encounter_id, patient_nbr))

# transform all "?" to 0s
diabetes[diabetes == "?"] <- NA

# remove zero variance - ty James http://stackoverflow.com/questions/8805298/quickly-remove-zero-variance-
variables-from-a-data-frame
diabetes <- diabetes[sapply(diabetes, function(x) length(levels(factor(x,exclude=NULL)))>1)]

# prep outcome variable to those readmitted under 30 days
diabetes$readmitted <- ifelse(diabetes$readmitted == "<30",1,0)

# generalize outcome name
outcomeName <- 'readmitted'
```

```
# large factors to deal with
length(unique(diabetes$diag_1))
length(unique(diabetes$diag_2))
length(unique(diabetes$diag_3))

# dummy var version -------------------------------------------------------
diabetes_dummy <- diabetes
# alwasy a good excersize to see the length of data that will need to be transformed
# charcolumns <- names(diabetes_dummy[sapply(diabetes_dummy, is.character)])
# for (thecol in charcolumns)
#         print(paste(thecol,length(unique(diabetes_dummy[,thecol]))))

# warning will need 2GB at least free memory
require(caret)
dmy <- dummyVars(" ~ .", data = diabetes_dummy)
diabetes_dummy <- data.frame(predict(dmy, newdata = diabetes_dummy))

# many features
dim(diabetes_dummy)

# change all NAs to 0
diabetes_dummy[is.na(diabetes_dummy)] <- 0

# split the data into training and testing data sets
set.seed(1234)
split <- sample(nrow(diabetes_dummy), floor(0.5*nrow(diabetes_dummy)))
objTrain <-diabetes_dummy[split,]
objTest <- diabetes_dummy[-split,]

predictorNames <- setdiff(names(diabetes_dummy),outcomeName)

# cv.glmnet expects a matrix
library(glmnet)
# straight matrix model not recommended - works but very slow, go with a sparse matrix
# glmnetModel <- cv.glmnet(model.matrix(~., data=objTrain[,predictorNames]), objTrain[,outcomeName],
#             family = "binomial", type.measure = "auc")
```

```
glmnetModel <- cv.glmnet(sparse.model.matrix(~., data=objTrain[,predictorNames]), objTrain[,outcomeName],
                         family = "binomial", type.measure = "auc")
glmnetPredict <- predict(glmnetModel,sparse.model.matrix(~., data=objTest[,predictorNames]), s="lambda.min")

# dummy version score:
auc(objTest[,outcomeName], glmnetPredict)

# feature hashed version -------------------------------------------------
diabetes_hash <- diabetes
predictorNames <- setdiff(names(diabetes_hash),outcomeName)

# change all NAs to 0
diabetes_hash[is.na(diabetes_hash)] <- 0

set.seed(1234)
split <- sample(nrow(diabetes_hash), floor(0.5*nrow(diabetes_hash)))
objTrain <-diabetes_hash[split,]
objTest <- diabetes_hash[-split,]

library(FeatureHashing)
objTrain_hashed = hashed.model.matrix(~., data=objTrain[,predictorNames], hash_size=2^12, transpose=FALSE, keep.hashing_mapping=TRUE)
objTrain_hashed = as(objTrain_hashed, "dgCMatrix")
objTest_hashed = hashed.model.matrix(~., data=objTest[,predictorNames], hash_size=2^12, transpose=FALSE, keep.hashing_mapping=TRUE)
objTest_hashed = as(objTest_hashed, "dgCMatrix")

library(glmnet)
glmnetModel <- cv.glmnet(objTrain_hashed, objTrain[,outcomeName],
                     family = "binomial", type.measure = "auc")
# hashed version score:
glmnetPredict <- predict(glmnetModel, objTest_hashed, s="lambda.min")
auc(objTest[,outcomeName], glmnetPredict)
```

Yelp, httr and a Romantic Trip Across the United States, One Florist at a Time

January 14, 2015 ***Tags:*** **visualizing exploring**

OREGON
IDAHO
WYOMING
DAKOTA
MICHIGAN
Toronto
NEW YORK
Chicago
IOWA
NEBRASKA
ILLINOIS
OHIO
PENNSYLVANIA
NEVADA
United States
UTAH
COLORADO
INDIANA
NJ
MD
DE
KANSAS
MISSOURI
WEST VIRGINIA
KENTUCKY
VIRGINIA
CALIFORNIA

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{httr}** - Tools for Working with URLs and HTTP
- **{jsonlite}** - Robust, High Performance JSON Parser and Generator for R
- **{ggplot2}** - Implementation of the Grammar of Graphics
- **{ggmap}** - Spatial visualization with Google Maps and OpenStreetMap

Just in time for **Valentine's Day**, if you happen to be planning a trip across the United States and want to offer your companion a rose at every degree of latitude traveled, then this walkthrough is for you! The title says it all, we’re going to use the Yelp API to cross the United States from San Francisco, CA to New York City, NY, and be 60 miles from a florist at all times.

This walkthrough has two parts, first exploring the basics of Yelp and httr, and finally looking at a mapping script to hop from florist to florist across the country. The itinerary will be plotted on ggmap. *If all you need is to pull locations via Yelp, then the first part is all you need.*

Part 1: Yelp and httr

We'll look at two different ways of using the Yelp Search API. For either of them, you will need to sign up for a free Yelp account and request API keys.

The simplest and probably most common way of using the API, is to request a term for a specific city or address. This is what the URL would look like for the term `food` around `San Franciso`:

http://api.yelp.com/v2/search?term=food&location=San+Francisco

Unfortunately, its a bit more complicated than simply dropping that in a browser. To authenticate (i.e. be recognized as a registered user), Yelp uses OAuth v1.0a protocol.

This is a cleaver of way authenticating you without exposing your account or password. On a very high level, it is done with a two-step process, first we ask Yelp for a signature key through a GET command, then we use that returned temporary key to retrieve our locations. Check out Yelp's authentication docs for more details.

This is a lot of work to do manually, but the handy httr package from Hadley Wickham handles a lot of the authentication details automatically. It's a really cool package as it can 'shake hands' with many **HTTP** API's (Twitter, Facebook, Linkedin, Google, etc).

After you get your keys from **Yelp**, swap them for the x's in the code below:

```
consumerKey = "xxxx"

consumerSecret = "xxxx"

token = "xxxx"

tokenSecret = "xxxx"
```

Load up the **httr** library and call the `oauth_app` function passing it a name tag (can be anything you want), and your `consumer` credentials. Next call up the `sign_oauth1.0`, pass it the `myapp` object along with your `token` credentials:

```
require(httr)

myApp <- oauth_app("YELP", key=consumerKey, secret=consumerSecret)

mySignature <- sign_oauth1.0(myApp, token=token, token_secret=tokenSecret)
```

Searching by Locations

Your signature is ready to be used on a **Yelp** search query:

```
yelpURL <- paste0("http://api.yelp.com/v2/search/?limit=3&term=food&location=San%20Francisco")

locationData <- GET(yelpURL, mySignature)
```

That is it! We entered the term **food** in **San Francisco** and asked to limit it to `3` result. Let's use the JSON parser, jsonlite to extract the results the results:

```
require(jsonlite)
locationDataContent = content(locationData)
locationList=jsonlite::fromJSON(toJSON(locationDataContent))
results <- data.frame(locationList)
results$businesses.name
## [[1]]
## [1] "Ike's Place"
##
## [[2]]
## [1] "The Italian Homemade Company"
##
## [[3]]
## [1] "The Codmother Fish and Chips"
```

`results$businesses.name` returned 3 businesses, but offers a whole lot more:

```
names(results)
##  [1] "region.span.latitude_delta"      "region.span.longitude_delta"
##  [3] "region.center.latitude"          "region.center.longitude"
##  [5] "total"                           "businesses.is_claimed"
##  [7] "businesses.rating"               "businesses.mobile_url"
##  [9] "businesses.rating_img_url"       "businesses.review_count"
## [11] "businesses.name"                 "businesses.snippet_image_url"
## [13] "businesses.rating_img_url_small" "businesses.url"
## [15] "businesses.menu_date_updated"    "businesses.phone"
## [17] "businesses.snippet_text"         "businesses.image_url"
## [19] "businesses.categories"           "businesses.display_phone"
## [21] "businesses.rating_img_url_large" "businesses.menu_provider"
## [23] "businesses.id"                   "businesses.is_closed"
## [25] "businesses.location"
```

Obviously, the terms **food** and **San Francisco** are way too vague, but you get the idea. Next we'll search by geo-spatial coordinates.

Searching by Geo-Spatial Coordinates

This is very close to what we did above except we specify the location down to the latitude and longitude of the desired area. How about 3 bars by Montgomery and Market Streets in San Francisco:

```
yelpURL <- paste0("http://api.yelp.com/v2/search/?limit=3&ll=37.788022,-122.399797&term=bar")
locationData <- GET(yelpURL, mySignature)
require(jsonlite)
locationDataContent = content(locationData)
locationList=jsonlite::fromJSON(toJSON(locationDataContent))
results <- data.frame(locationList)
results$businesses.name
## [[1]]
## [1] "83 Proof"
##
## [[2]]
## [1] "Topsy's Fun House"
##
## [[3]]
## [1] "Rickhouse"
```

The difference is syntactical, instead of building the URL with `location`, we do it with `ll`. We'll use a third way of pulling locations in part 2 - geo-spatial bounded boxes. That's it for part 1.

Part 2: Hoping from Florist to Florist Across the United States

Let me tell you, this was a blast to design and build! I'll go over the pertinent parts but if you want to see this in action yourself, copy the code at the end of the walkthrough and replace the Yelp credentials with yours and you're good to go!

Though not a full-proof application (matter of fact it can only travel west to east), it will take a departure and destination set of latitudes and longitudes, a search term and attempt to link both points with a path using your search term every 60 miles (approximately 1 geo-coordinate degree). Here, we will go from San Francisco, CA to New York City, NY, florist by florist.

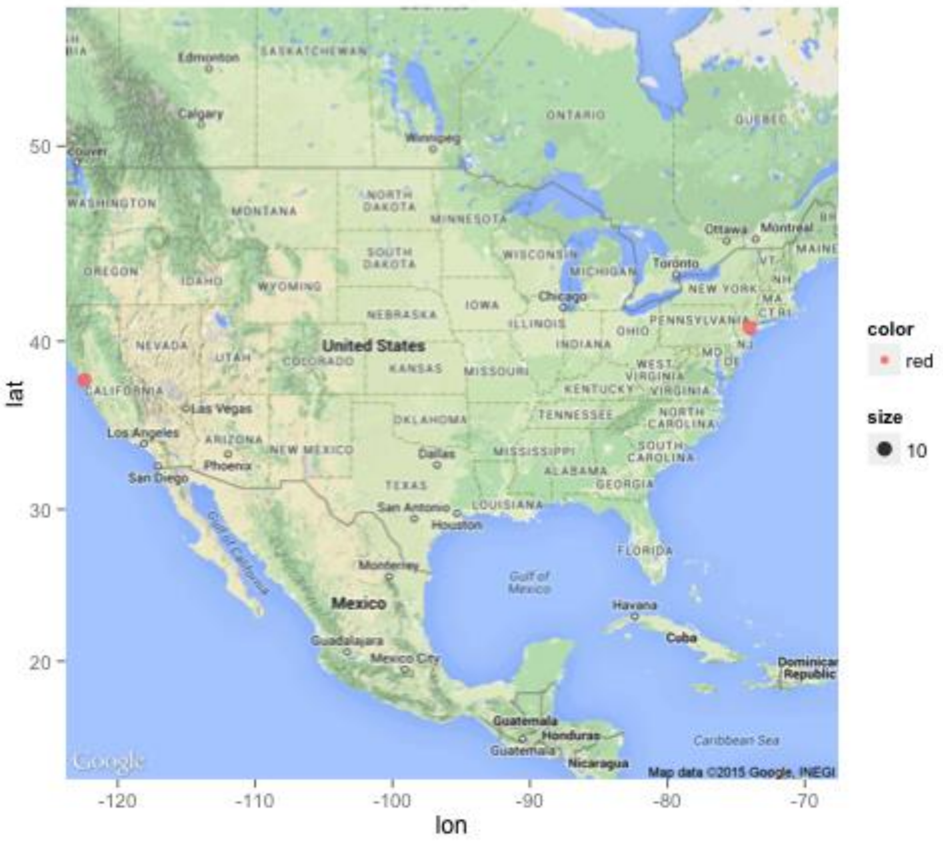
`ggmap` has a handy function called `geocode`. You can give it addresses, zip codes, even famous monuments, and it will return latitude and longitude coordinates.

```
require(ggplot2)

require(ggmap)

startingpoint <- geocode(c("Fishersman's Wharf, San Francisco, CA"))

print(startingpoint)

##      lon   lat

## 1 -122.4 37.81

endingpoint <- geocode(c("Chelsea Piers, NY"))

print(endingpoint)

##      lon   lat

## 1 -74.01 40.75
```

Now that we know our starting and ending points, let's start our first point manually then generalize the tasks through functions.

We'll use `ggmap` to import a map of the United States from Google (see Mapping The United States Census With {ggmap} for more details on `ggmap`). `ggplot2` accepts a data frame, let's make one with our start and end points and see the journey that lies ahead. We'll assign them large sizes to differentiate them from the traveling points:

```
latitudes <- c(startingpoint$lat, endingpoint$lat)

longitudes <- c(startingpoint$lon, endingpoint$lon)



objdf <- data.frame('latitude'=latitudes,

                    'longitude'=longitudes,
                    'size'=c(10,10),
                    'color'=c('red','red'))

# get a Google map
map<-get_map(location='united states', zoom=4, maptype = "terrain",
             source='google',color='color')

objMap <- ggmap(map) + geom_point(
        aes(x=longitude, y=latitude, size=size,
            show_guide = TRUE, colour = color),
        data=objdf, alpha=.8, na.rm = T)

# call the objMap object to see the plot
objMap
```



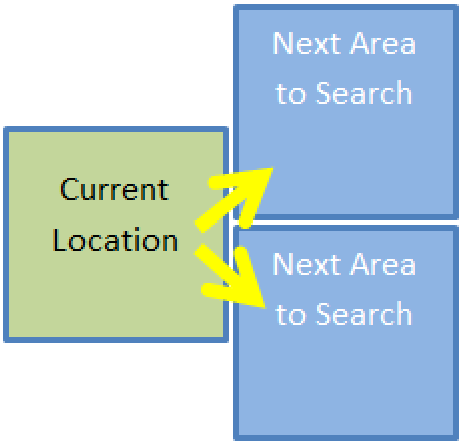
I will keep the details short on the next few functions. `MapIt` takes geo-spatial coordinates, a size, and the `ggmap` object. The function will return the `ggmap` object and we need to always pass the same `ggmap` back if we want to update and track the journey's history.

```
MapIt <- function(latitude, longitude, size, objggmap) {
        objdf <- data.frame('latitude'=latitude,
                            'longitude'=longitude,
                            'tsize'=size)

        # add new point to ggmap
        require("ggplot2")
        require("ggmap")
        objggmap <- objggmap + geom_point(
                aes(x=longitude, y=latitude, size=tsize,
                    show_guide = TRUE),
                data=objdf, alpha=.8, na.rm = T)
        return (objggmap)
```

`GetBestYelpLocation`, as its name implies, takes a vector of two sets of geo-spatial coordinates, the upper left point and the lower right one, along with the search term and sends it to Yelp. It returns a data frame with the `name`, `city`, `rating`, `state`, `latitude` and `longitude` of the top location using the search term.

```
GetBestYelpLocation <- function(boundedcoordinates, term) {
        limit <- 1

        YelpUrl <- paste0("http://api.yelp.com/v2/search/?limit=20&bounds=",
                          boundedcoordinates[1],",",boundedcoordinates[2],
                          "|",boundedcoordinates[3],",",boundedcoordinates[4],"&term=",term)

        locationdata=GET(YelpUrl, mySignature)
        listMembersContent = content(locationdata)
        listMembers=jsonlite::fromJSON(toJSON(listMembersContent))

        yelpResults = tryCatch({
                data.frame(listMembers)
        }, error = function(e) {
                NULL
        })
        if (!is.null(yelpResults)) {
                set1 <- data.frame("name"=yelpResults$businesses.name,
                   'city'=yelpResults$businesses.location$city,
                   'rating'=yelpResults$businesses.rating,
                   'latitude'=yelpResults$businesses.location$coordinate$latitude,

                   'longitude'=yelpResults$businesses.location$coordinate$longitude,

                   'state'=yelpResults$businesses.location$state_code)


                for (ind in seq(1:nrow(set1))) {

                        if ((set1$latitude[ind] <= boundedcoordinates[1]) &

                                (set1$latitude[ind] >= boundedcoordinates[3]) &

                                (set1$longitude[ind] >= boundedcoordinates[2]) &

                                (set1$longitude[ind] <= boundedcoordinates[4]))

                                return(set1[ind,])

                }

        }

        return(NULL)

}
```

`GetPossibleCoordinates` takes the current position of our traveler and calculates its bounded box. The default bounded box unit is 1 degree (equates loosely around 60 miles), so we add 30 miles in all four directions from the current point to created the bounded square (making it a 1x1 degree box, or 60x60 miles box).



It returns the next two possible moves, both forward but one higher and one lower (see image below).

```
GetPossibleCoordinates<-function(lat, lon, area=1) {
        # this is hard to keep track but each location's next move
        # equates attempting to move forward one square, the same area
        # as the previous one but half-way above and half-way below the
        # previous one.
        halfarea <- area/2

        # new forward top square area
        topArea_upperleft_latitude <- lat+area
        topArea_upperleft_longitude <- lon+halfarea

        topArea_lowerright_latitude <- lat
        topArea_lowerright_longitude <- lon+(halfarea+area)

        # new forward bottom square area
        bottomArea_upperleft_latitude <- lat
        bottomArea_upperleft_longitude <- lon+halfarea

        bottomArea_lowerright_latitude <- lat-area
        bottomArea_lowerright_longitude <- lon+(halfarea+area)


        rownames <- c('new_top_area','new_bottom_area')
        latitude_point1 <- c(topArea_upperleft_latitude, bottomArea_upperleft_latitude)
        longitude_point1 <- c(topArea_upperleft_longitude, bottomArea_upperleft_longitude)
        latitude_point2 <- c(topArea_lowerright_latitude, bottomArea_lowerright_latitude)
        longitude_point2 <- c(topArea_lowerright_longitude, bottomArea_lowerright_longitude)

        return (data.frame('direction'=rownames,
                'latitude_point1'=latitude_point1,
                           'longitude_point1'=longitude_point1,
                           'latitude_point2'=latitude_point2,
                           'longitude_point2'=longitude_point2))


}
```

Finally, `MakeAMove` is the central relay that communicates with the other functions. It will get the next coordinates via `GetPossibleCoordinates`, figure out if it should try up or down first depending on where it is in relation to the end point's latitude, and call on the Yelp API. If it finds a florist, it will report back the pertinent information such as the florist's name, state, and coordinates. If it can't find one, it will report so by returning `NULL`.

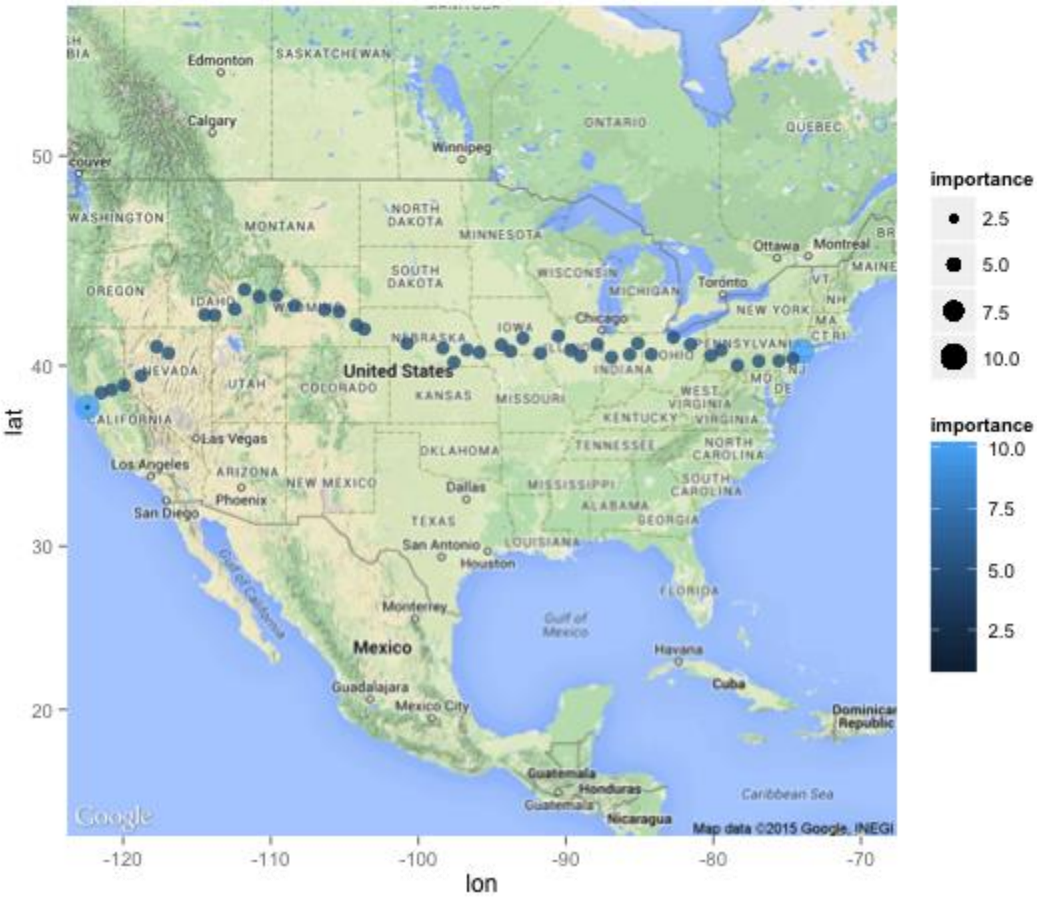
```
MakeAMove <- function(lat,lon,sizebox, searchTerm, lat_endPoint) {
        possibleCoordinates <- GetPossibleCoordinates(lat,lon,sizebox)

        # go up or down first depending on latitude of end point
        searchOrder <- c('new_top_area','new_bottom_area')
        if (lat > lat_endPoint)
                searchOrder <- c('new_bottom_area','new_top_area')

        for (directiontogo in searchOrder) {
                coords <- possibleCoordinates[possibleCoordinates$direction==directiontogo,]
                print(paste('Scanning',directiontogo, 'for', searchTerm,'...'))
                foundLocation <- GetBestYelpLocation(as.numeric(as.vector(coords[2:5])), searc
hTerm)
                if (!is.null(foundLocation))
                        return (foundLocation)
        }
        return (NULL)
}
```

Let's finally look at the main logic. Here we set a search term `florist`, and a bounded square size unit, `1` degree in this case. `madeIt` is the boolean flag that keeps the while loop going until we make it close to our final destination (or if we fail after 100 tries). `Sys.sleep(0.5)` will pause the code between each loop, this is critical so you don't flood the Yelp server and get yourself banned.

```
searchTerm <- 'florist'
squareSize <- 1 # setting the bounded area to a square

# start trip info vectors - we need to remember where we've been!
currentLatitude <- startingpoint$lat
currentLongitude <- startingpoint$lon

# let ggmap keep track of where we've been
objMap <- MapIt(currentLatitude, currentLongitude, 1, objMap)

madeIt=FALSE
safetyCount <- 0
foundCount <- 0
while(madeIt == FALSE) {
        safetyCount <- safetyCount + 1
        foundLocation <- MakeAMove(lat=currentLatitude,
                                   lon=currentLongitude,
                                   sizebox=squareSize,
                                   searchTerm=searchTerm,
                                   endingpoint$lat)

        if (!is.null(foundLocation)) {
                print (paste('Our new',searchTerm, 'is', foundLocation$name, 'in',
                             foundLocation$city, foundLocation$state,
                             'with a', foundLocation$rating, 'start rating'))
                currentLatitude <- foundLocation$latitude
                currentLongitude <- foundLocation$longitude

                # reset temporary setting
                squareSize <- 1

                # let ggmap keep track of where we've been
                objMap <- MapIt(currentLatitude, currentLongitude, squareSize+3, objMap)

                # let's keep track how our successes!
                foundCount <- foundCount + 1
        } else {
                # increase squareSize
                print(paste("Can't find any", searchTerm,", enlarging square search area to",s
quareSize + 1))

                # temporary settings to get us out of desert
                squareSize <- squareSize + 1
        }

        # have we arrived at our end point
        if ((currentLongitude < (endingpoint$lon + squareSize))
                & (currentLongitude > (endingpoint$lon - squareSize)))
        {
                print(paste('We made it!! It took',foundCount,'hops...'))
                break
        }
        if (safetyCount > 100)
        {
                print(paste('Giving up!! Failed after',foundCount,'hops'))
                break
        }

        # be considerate with your Yelp requests
        Sys.sleep(0.5)
}
```

This is the output you will see when you run the code. It will log each point traveled and print the name of the florist. In case it cannot find one in either the upper or lower forward areas, the system will increase the square size by a degree (making it a 2x2 box instead of a 1x1 box) until it can find a florist. This is why in the final map below you will see gaps in deserted or mountainous zones.

```
## [1] "Scanning new_top_area for florist ..."
## [1]   40.46699 -118.28123   39.46699 -117.28123
## [1] "Scanning new_bottom_area for florist ..."
## [1]   39.46699 -118.28123   38.46699 -117.28123
## [1] "Can't find any florist , enlarging square search area to 2"
## [1] "Scanning new_top_area for florist ..."
## [1]   41.46699 -117.78123   39.46699 -115.78123
## [1] "Our new florist is Second Street Seasonals in Winnemucca NV with a 4 start rating"

...

## [1] "Our new florist is Achin' Back Garden Center in Pottstown PA with a 4.5 start rating"
## [1] "Scanning new_top_area for florist ..."
## [1]  41.24341 -75.06065  40.24341 -74.06065
## [1] "Our new florist is Monday Morning Flower and Balloon Co. in Princeton NJ with a 4.5 start rating"
## [1] "We made it!! It took 41 hops..."
```



Conclusion

So, I didn't quite deliver on my promise of being within 60 miles of a florist at all times, it may or may not be possible (and we certainly won't find out with the state of my current code). Yet, we managed to be within those parameters during most of the journey. You can try with with different terms, different locations, even upgrade the algorithm to do cooler things - just don't abuse the API as it may end the party for all of us.

A special thanks to **Nathan Yau**; this walkthrough was loosely based but strongly inspired by his In search of food deserts.

Full source code (also on GitHub):

```
######################## functions #################################
```

```
MapIt <- function(latitude, longitude, size, objggmap) {
        objdf <- data.frame('latitude'=latitude,
                            'longitude'=longitude,
                            'tsize'=size)

        # add new point to ggmap
        require("ggplot2")
        require("ggmap")
        objggmap <- objggmap + geom_point(
                aes(x=longitude, y=latitude, size=tsize,
                    show_guide = TRUE),
                data=objdf, alpha=.8, na.rm = T)
        return (objggmap)
}

# yelp rest API call
GetBestYelpLocation <- function(boundedcoordinates, term) {
        limit <- 1
        print(boundedcoordinates)
        # Pick first florist in San Francisco, CA
        #YelpUrl <- paste0("http://api.yelp.com/v2/search/?limit=",limit,
        #                  "&location=San%20Francisco%20CA&term=florist")
        # or 10 bars by geo-coordinates
        #YelpUrl <- paste0("http://api.yelp.com/v2/search/?limit=",
        #       limit,"&ll=",latitude,",",longitude,"&term=florist")
        # or by bounded geo-spatial coordinates
        YelpUrl <- paste0("http://api.yelp.com/v2/search/?limit=20&bounds=",
                          boundedcoordinates[1],",",boundedcoordinates[2],
                          "|",boundedcoordinates[3],",",boundedcoordinates[4],"&term=",term)

        locationdata=GET(YelpUrl, mySignature)
        listMembersContent = content(locationdata)
        listMembers=jsonlite::fromJSON(toJSON(listMembersContent))

        yelpResults = tryCatch({
                data.frame(listMembers)
        }, error = function(e) {
                NULL
        })
        if (!is.null(yelpResults)) {
                set1 <- data.frame("name"=yelpResults$businesses.name,'city'=yelpResults$businesses.locati
on$city,
                                   'rating'=yelpResults$businesses.rating,
                                   'latitude'=yelpResults$businesses.location$coordinate$latitude,
                                   'longitude'=yelpResults$businesses.location$coordinate$longitude,
                                   'state'=yelpResults$businesses.location$state_code)

                for (ind in seq(1:nrow(set1))) {
                        if ((set1$latitude[ind] <= boundedcoordinates[1]) &
                                (set1$latitude[ind] >= boundedcoordinates[3]) &
                                (set1$longitude[ind] >= boundedcoordinates[2]) &
                                (set1$longitude[ind] <= boundedcoordinates[4]))
                                return(set1[ind,])
                }
        }
        return(NULL)
}

# Movement controls
GetPossibleCoordinates<-function(lat, lon, area=1) {
        # this is hard to keep track but each location's next move
        # equates attempting to move forward one square, the same area
        # as the previous one but half-way above and half-way below the
        # previous one.
        halfarea <- area/2

        # new forward top square area
        topArea_upperleft_latitude <- lat+area
        topArea_upperleft_longitude <- lon+halfarea

        topArea_lowerright_latitude <- lat
        topArea_lowerright_longitude <- lon+(halfarea+area)

        # new forward bottom square area
        bottomArea_upperleft_latitude <- lat
        bottomArea_upperleft_longitude <- lon+halfarea
        bottomArea_lowerright_latitude <- lat-area
        bottomArea_lowerright_longitude <- lon+(halfarea+area)


        rownames <- c('new_top_area','new_bottom_area')
        latitude_point1 <- c(topArea_upperleft_latitude, bottomArea_upperleft_latitude)
        longitude_point1 <- c(topArea_upperleft_longitude, bottomArea_upperleft_longitude)
        latitude_point2 <- c(topArea_lowerright_latitude, bottomArea_lowerright_latitude)
        longitude_point2 <- c(topArea_lowerright_longitude, bottomArea_lowerright_longitude)

        return (data.frame('direction'=rownames,
                'latitude_point1'=latitude_point1,
                           'longitude_point1'=longitude_point1,
                           'latitude_point2'=latitude_point2,
                           'longitude_point2'=longitude_point2))

}

MakeAMove <- function(lat,lon,sizebox, searchTerm, lat_endPoint) {
        possibleCoordinates <- GetPossibleCoordinates(lat,lon,sizebox)

        # go up or down first depending on latitude of end point
        searchOrder <- c('new_top_area','new_bottom_area')
        if (lat > lat_endPoint)
                searchOrder <- c('new_bottom_area','new_top_area')

        for (directiontogo in searchOrder) {
                coords <- possibleCoordinates[possibleCoordinates$direction==directiontogo,]
                print(paste('Scanning',directiontogo, 'for', searchTerm,'...'))
                foundLocation <- GetBestYelpLocation(as.numeric(as.vector(coords[2:5])), searchTerm)
                if (!is.null(foundLocation))
                        return (foundLocation)
        }
        return (NULL)
}

############################## YELP SETUP ###############################
# yelp credentials
```

```
consumerKey = "xxxxx"
consumerSecret = "xxxxx"
token = "xxxxx"
tokenSecret = "xxxxx"

require(httr)
require(jsonlite)

# authorization
myapp = oauth_app("Yelp", key=consumerKey, secret=consumerSecret)
mySignature=sign_oauth1.0(myapp, token=token,token_secret=tokenSecret)

############################### Manual Code ###############################
require(ggplot2)
require(ggmap)

# Get our starting and ending points
# Fisherman's Wharf, San Francisco, CA 94109
startingpoint <- geocode(c("Fishersman's Wharf, San Francisco, CA"))
# Chelsea Piers 10011
endingpoint <- geocode(c("Chelsea Piers, NY"))

# ggplots accepts a data frame, let's make with with start and end points
latitudes <- c(startingpoint$lat, endingpoint$lat)
longitudes <- c(startingpoint$lon, endingpoint$lon)
# importance holds the size and color (we'll leave it up to ggplot to choose)
objdf <- data.frame('latitude'=latitudes,
                    'longitude'=longitudes,
                    'importance'=c(10,10))

# sample ggplot/ggmap with start and end locations
# get a Google map
require(ggmap)
map<-get_map(location='united states', zoom=4, maptype = "terrain",
             source='google',color='color')

# plot it with ggplot2
require("ggplot2")
```

```
objMap <- NULL
objMap <- ggmap(map) + geom_point(
        aes(x=longitude, y=latitude, size=importance,
            show_guide = TRUE, colour = importance),
        data=objdf, alpha=.8, na.rm = T)


################################### Logic #####################################

searchTerm <- 'florist'
squareSize <- 1 # setting the bounded area to a square

# start trip info vectors - we need to remember where we've been!
currentLatitude <- startingpoint$lat
currentLongitude <- startingpoint$lon

# let ggmap keep track of where we've been
objMap <- MapIt(currentLatitude, currentLongitude, 1, objMap)

madeIt=FALSE
safetyCount <- 0
foundCount <- 0
while(madeIt == FALSE) {
        safetyCount <- safetyCount + 1
        foundLocation <- MakeAMove(lat=currentLatitude,
                                   lon=currentLongitude,
                                   sizebox=squareSize,
                                   searchTerm=searchTerm,
                                   endingpoint$lat)

        if (!is.null(foundLocation)) {
                print (paste('Our new',searchTerm, 'is', foundLocation$name, 'in',
                             foundLocation$city, foundLocation$state,
                             'with a', foundLocation$rating, 'start rating'))
                currentLatitude <- foundLocation$latitude
                currentLongitude <- foundLocation$longitude

                # reset temporary setting
                squareSize <- 1

                # let ggmap keep track of where we've been
                objMap <- MapIt(currentLatitude, currentLongitude, squareSize+3, objMap)

                # let's keep track how our successes!
                foundCount <- foundCount + 1
        } else {
                # increase squareSize
                print(paste("Can't find any", searchTerm,", enlarging square search area to",squareSize +
1))

                # temporary settings to get us out of desert
                squareSize <- squareSize + 1
        }

        # have we arrived at our end point
        if ((currentLongitude < (endingpoint$lon + squareSize))
                & (currentLongitude > (endingpoint$lon - squareSize)))
        {
                print(paste('We made it!! It took',foundCount,'hops...'))
                break
        }
        if (safetyCount > 100)
        {
                print(paste('Giving up!! Failed after',foundCount,'hops'))
                break
        }

        # be considerate with your Yelp requests
        Sys.sleep(0.5)
}

objMap
```

Quantifying the Spread: Measuring Strength and Direction of Predictors with the Summary Function

December 27, 2014 **Tags:** **modeling**

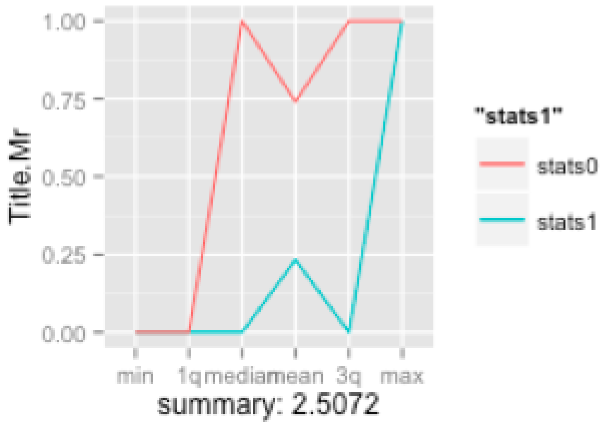
Resources

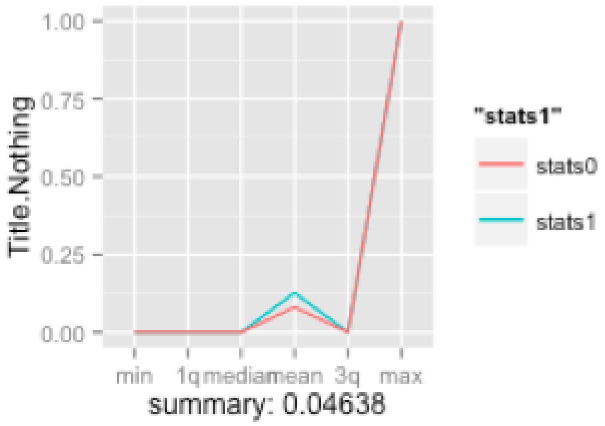
- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{caret}** - dummyVars function
- **{ggplot2}** - Graphics/Grammar of Graphics
- **{grid}** - Grid Graphics Package

You're probably familiar within the summary() function in **R**. It's an essential function, used all the time, that can reveal so much about your data. Yet, by extending it just a tad, we can quickly highlight top predictors, even on extremely large data sets.





The idea is not to summarize the variable in of itself, but to split the data into two sets, one for each outcome and summarize each. Comparing the results from both sets will tell you how well you predictor behaves towards your outcome variable. The above plots shows the summary of two predictors and their individual spreads - clearly, this first plot is a powerful predictor as the spread between the green and red line is large, while the second one isn't.

Let's first measure a single predictor to see how this works, then we'll quantify this technique to it can be applied to entire data sets.

We’ll pull in the classic **Titanic** data set that I’ve already used in many of my walkthroughs. The code below will download the data from the **University of Colorado**, clean it up and yield a numeric-only, modeling-ready data frame:

```
# using dataset from the UCI Machine Learning Repository (http://archive.ics.uci.edu/ml/)
titanicDF <- read.csv('http://math.ucdenver.edu/RTutorial/titanic.txt',sep='\t')


# creating new title feature
titanicDF$Title <- ifelse(grepl('Mr ',titanicDF$Name),'Mr',ifelse(grepl('Mrs ',titanicDF$Name)
,'Mrs',ifelse(grepl('Miss',titanicDF$Name),'Miss','Nothing')))
titanicDF$Title <- as.factor(titanicDF$Title)


# impute age to remove NAs
titanicDF$Age[is.na(titanicDF$Age)] <- median(titanicDF$Age, na.rm=T)


# reorder data set so target is last column
titanicDF <- titanicDF[c('PClass', 'Age',    'Sex',   'Title', 'Survived')]


# binarize all factors
require(caret)
titanicDummy <- dummyVars("~.",data=titanicDF, fullRank=F)
titanicDF <- as.data.frame(predict(titanicDummy,titanicDF))
```

Let's call `head` on our variables to see what we are dealing with. `Survived` is our **outcome** variable:

```
head(titanicDF, 3)
##   PClass.1st PClass.2nd PClass.3rd Age Sex.female Sex.male Title.Miss
## 1          1          0          0  29           1        0          1
## 2          1          0          0   2           1        0          1
## 3          1          0          0  30           0        1          0
##   Title.Mr Title.Mrs Title.Nothing Survived
## 1        0         0             0        1
## 2        0         0             0        0
## 3        1         0             0        0
```

Calling the `summary` function or the str function on an entire data frame is a great way of getting acquainted with it:

```
summary(titanicDF)
##    PClass.1st      PClass.2nd      PClass.3rd          Age
##  Min.   :0.000   Min.   :0.000   Min.   :0.000   Min.   : 0.17
##  1st Qu.:0.000   1st Qu.:0.000   1st Qu.:0.000   1st Qu.:26.00
##  Median :0.000   Median :0.000   Median :1.000   Median :28.00
##  Mean   :0.245   Mean   :0.213   Mean   :0.541   Mean   :29.38
##  3rd Qu.:0.000   3rd Qu.:0.000   3rd Qu.:1.000   3rd Qu.:30.00
##  Max.   :1.000   Max.   :1.000   Max.   :1.000   Max.   :71.00
##    Sex.female       Sex.male       Title.Miss       Title.Mr
##  Min.   :0.000   Min.   :0.000   Min.   :0.00   Min.   :0.000
##  1st Qu.:0.000   1st Qu.:0.000   1st Qu.:0.00   1st Qu.:0.000
##  Median :0.000   Median :1.000   Median :0.00   Median :1.000
##  Mean   :0.352   Mean   :0.648   Mean   :0.18   Mean   :0.572
##  3rd Qu.:1.000   3rd Qu.:1.000   3rd Qu.:0.00   3rd Qu.:1.000
##  Max.   :1.000   Max.   :1.000   Max.   :1.00   Max.   :1.000
##    Title.Mrs     Title.Nothing      Survived
##  Min.   :0.000   Min.   :0.000   Min.   :0.000
##  1st Qu.:0.000   1st Qu.:0.000   1st Qu.:0.000
##  Median :0.000   Median :0.000   Median :0.000
##  Mean   :0.152   Mean   :0.096   Mean   :0.343
##  3rd Qu.:0.000   3rd Qu.:0.000   3rd Qu.:1.000
##  Max.   :1.000   Max.   :1.000   Max.   :1.000
```

As you can see, this yields a version of the five-number summary displaying the **min, max, 1st & 3rd quantile, mean, medium** of each variable.

Though extremely useful, this doesn't help us understand how our outcome variable interacts with its predictors. To remedy this, we split the data into two separate data sets, an outcome-positive data set, and an outcome-negative data set. Let's look at `Sex.female`:

```
df_survived_1 <- subset(titanicDF, Survived==1)
df_survived_0 <- subset(titanicDF, Survived==0)
summary(df_survived_1$Sex.female)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   1.000   0.684   1.000   1.000
summary(df_survived_0$Sex.female)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   0.000   0.178   0.000   1.000
```

Now, a much clearer picture emerges regarding that variable when using an outcome-specific perspective. We learn that almost 70% of those that survived were females, and that only 18% of those that died were females. I think you can see where I am going with this.

Let's see how we can graph this information in an intuitive way. The idea is to create a vector of summary information for both outcomes, overlaying them together and measuring the spread. We'll continue with the `Sex.female` variable as it should already be apparent how strong of a predictor it is.

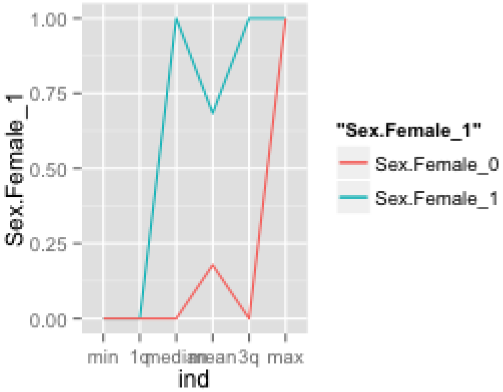
```
Sex.Female_0 <- (summary(df_survived_0$Sex.female))
Sex.Female_0 <- c(Sex.Female_0[1:6])
Sex.Female_1 <- (summary(df_survived_1$Sex.female))
Sex.Female_1 <- c(Sex.Female_1[1:6])
stats <- data.frame('ind'=c(1:6),
                    'Sex.Female_1'=Sex.Female_1,

                    'Sex.Female_0'=Sex.Female_0)
```

`stats` is the data frame holding the summary data for `Sex.female` for both outcomes. This data frame holds three variables, an index column, a predictor column where the outcome is positive and another where the outcome is negative, and 6 rows, one for each summary output:

```
head(stats,6)

##         ind Sex.Female_1 Sex.Female_0

## Min.      1        0.000        0.000

## 1st Qu.   2        0.000        0.000

## Median    3        1.000        0.000

## Mean      4        0.684        0.178

## 3rd Qu.   5        1.000        0.000

## Max.      6        1.000        1.000
```

The logical next step is to plot it (that's how the plots were generated in the start of the walkthrough):

```
require(ggplot2)

p <- ggplot(data=stats, aes(ind)) +

               geom_line(aes(y = Sex.Female_1, colour = "Sex.Female_1")) +

               geom_line(aes(y = Sex.Female_0, colour = "Sex.Female_0")) +

               scale_x_discrete(breaks = 1:6,

                                labels=c("min","1q","median","mean","3q","max"))

p
```



Obviously, we're going to generalize this into a neat function, but, by doing it by hand for a single variable you should start seeing the power of looking at summary data is such manner. By using ggplot2, it is easy to display the summary measures on the `x` axis. The green line represents the females that survived and the red one, those that didn't. As you can see both lines are far apart, the green line is well above the red one, and this means that the variable is a powerful predictor for survivability (i.e. we have a positive spread).

If we generalize this spread into a single number, we'll be able to bypass the graphing phase and apply this to huge data sets.

```
spread <- ((Sex.Female_1[[1]] - Sex.Female_0[[1]]) +

             (Sex.Female_1[[2]] - Sex.Female_0[[2]]) +

             (Sex.Female_1[[3]] - Sex.Female_0[[3]]) +

             (Sex.Female_1[[4]] - Sex.Female_0[[4]]) +

             (Sex.Female_1[[5]] - Sex.Female_0[[5]]) +

             (Sex.Female_1[[6]] - Sex.Female_0[[6]]))

print(spread)

## [1] 2.506
```

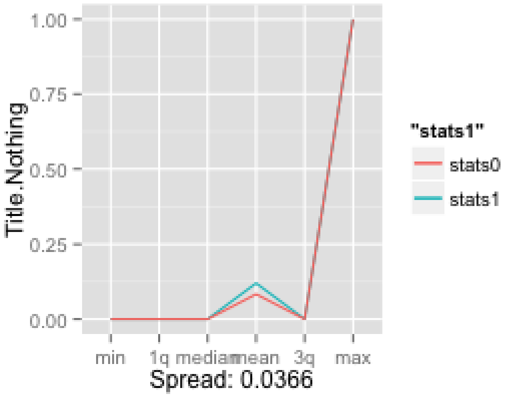
2.56 is a large spread, let's compare it with the weaker variable `Title.Nothing`:

```
Title.Nothing_0 <- (summary(df_survived_0$Title.Nothing))

Title.Nothing_0 <- c(Title.Nothing_0[1:6])

Title.Nothing_1 <- (summary(df_survived_1$Title.Nothing))

Title.Nothing_1 <- c(Title.Nothing_1[1:6])
```

```
stats <- data.frame('ind'=c(1:6), 'stats1'=Title.Nothing_1,'stats0'=Title.Nothing_0)
spread <- ((Title.Nothing_1[[1]] - Title.Nothing_0[[1]]) +
             (Title.Nothing_1[[2]] - Title.Nothing_0[[2]]) +
             (Title.Nothing_1[[3]] - Title.Nothing_0[[3]]) +
             (Title.Nothing_1[[4]] - Title.Nothing_0[[4]]) +
             (Title.Nothing_1[[5]] - Title.Nothing_0[[5]]) +
             (Title.Nothing_1[[6]] - Title.Nothing_0[[6]]))
print(spread)
## [1] 0.0366
```

There you go, **0.0366** is a much smaller spread than **2.56**! This quantative value will allow us to cycle through any number of variables without having to plot them individually:

```
p <- ggplot(data=stats, aes(ind)) +
               geom_line(aes(y = stats1, colour = "stats1")) +
               geom_line(aes(y = stats0, colour = "stats0")) +
               scale_x_discrete(breaks = 1:6,
                                labels=c("min","1q","median","mean","3q","max")) +
                ylab('Title.Nothing') + xlab(paste('Spread:',spread))
p
```



Let's Generalize

It is time to generalize all this so we don't have to tediously type each variable name every time we want to measure these relationships.

We'll create a function called `GetSummaryAndPlots()`. It takes a scaled data set of each outcome, the `predictorName` variable, and whether we want it to be plotted. If we don't want it plotted, it will return the spread value:

```
GetSummaryPlot <- function(objdfscaled0, objdfscaled1, predictorName, plotit=TRUE) {
     require(ggplot2)

     stats0 <- (summary(objdfscaled0[,predictorName]))
     stats0 <- c(stats0[1:6])
     stats1 <- (summary(objdfscaled1[,predictorName]))
     stats1 <- c(stats1[1:6])
     stats <- data.frame('ind'=c(1:6), 'stats1'=stats1,'stats0'=stats0)

     spread <- ((stats1[[1]] - stats0[[1]]) +
                     (stats1[[2]] - stats0[[2]]) +
                     (stats1[[3]] - stats0[[3]]) +
                     (stats1[[4]] - stats0[[4]]) +
                     (stats1[[5]] - stats0[[5]]) +
                     (stats1[[6]] - stats0[[6]]))

     if (plotit) {
          print(paste('Scaled spread:',spread))
          p <- ggplot(data=stats, aes(ind)) +
               geom_line(aes(y = stats1, colour = "stats1")) +
               geom_line(aes(y = stats0, colour = "stats0")) +
               scale_x_discrete(breaks = 1:6,
                                labels=c("min","1q","median","mean","3q","max")) +
               ylab(predictorName) + xlab(paste('Spread:',spread))
          return (p)

     } else {

          return (spread)

     }

}
```

Let's look at another piece of code that will allow us to generalize this further. By using the `scale` function and scaling the entire data set into a standard unit of measurement, we can easily compare the spread between each predictor. Let's give it a whirl on the entire data set...

```
outcomeName <- 'Survived'

predictorNames <- names(titanicDF)[!names(titanicDF) %in% outcomeName]

# Temporarily remove the outcome variable before scaling the data set

outcomeValue <- titanicDF$Survived

# scale returns a matrix so we need to tranform it back to a data frame

scaled_titanicDF <- as.data.frame(scale(titanicDF))

scaled_titanicDF$Survived <- outcomeValue

# split your data sets

scaled_titanicDF_0 <- scaled_titanicDF[scaled_titanicDF[,outcomeName]==0,]

scaled_titanicDF_1 <- scaled_titanicDF[scaled_titanicDF[,outcomeName]==1,]


for (predictorName in predictorNames)

        print(paste(predictorName,':',GetSummaryPlot(scaled_titanicDF_0,

                scaled_titanicDF_1, predictorName, plotit=FALSE)))

## [1] "PClass.1st : 2.969"

## [1] "PClass.2nd : 2.6301"

## [1] "PClass.3rd : -2.727"

## [1] "Age : -0.2498"

## [1] "Sex.female : 5.253"

## [1] "Sex.male : -5.253"
## [1] "Title.Miss : 3.119"
## [1] "Title.Mr : -5.071"
## [1] "Title.Mrs : 3.54"
## [1] "Title.Nothing : 0.1241"
```

Pretty cool, right? With a quick glance we see that `Sex.male` and `Title.Mr` have a substantial negative effect on survivability while `Sex.female` has a substantial positive.

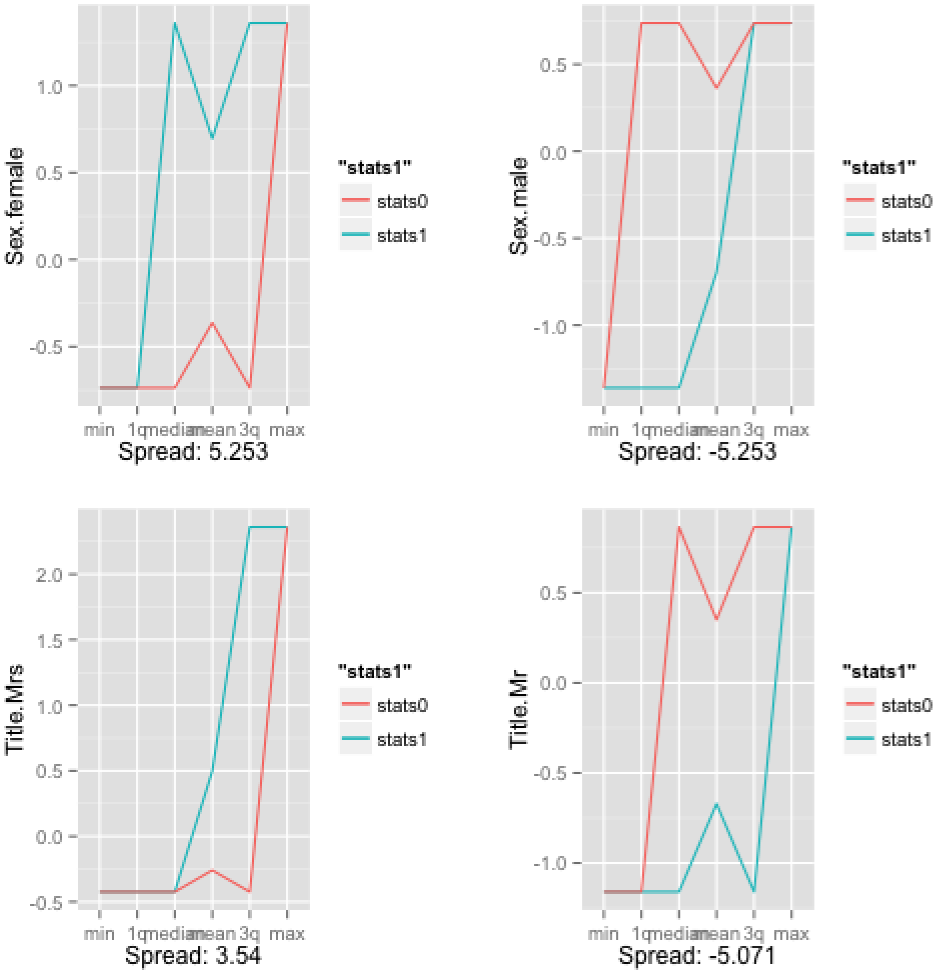
How about plotting this efficiently? We'll use the `multiplot` function from the great resource: Cookbook for R. This enables the stacking of multiple `ggplots` on the same page, just like `par` and `mfrow` does for regular R plots. For more information, please visit the above link.

```
multiplot <- function(..., plotlist=NULL, file, cols=1, layout=NULL) {
    #http://www.cookbook-r.com/Graphs/Multiple_graphs_on_one_page_(ggplot2)/
    # Multiple plot function
    #
    # ggplot objects can be passed in ..., or to plotlist (as a list of ggplot objects)
    # - cols:   Number of columns in layout
    # - layout: A matrix specifying the layout. If present, 'cols' is ignored.
    #
    # If the layout is something like matrix(c(1,2,3,3), nrow=2, byrow=TRUE),
    # then plot 1 will go in the upper left, 2 will go in the upper right, and
    # 3 will go all the way across the bottom.
    #
    require(grid)

    # Make a list from the ... arguments and plotlist
    plots <- c(list(...), plotlist)

    numPlots = length(plots)
```

```
     # If layout is NULL, then use 'cols' to determine layout
     if (is.null(layout)) {
          # Make the panel
          # ncol: Number of columns of plots
          # nrow: Number of rows needed, calculated from # of cols
          layout <- matrix(seq(1, cols * ceiling(numPlots/cols)),
                           ncol = cols, nrow = ceiling(numPlots/cols))
     }

     if (numPlots==1) {
          print(plots[[1]])

     } else {
          # Set up the page
          grid.newpage()
          pushViewport(viewport(layout = grid.layout(nrow(layout), ncol(layout))))

          # Make each plot, in the correct location
          for (i in 1:numPlots) {
               # Get the i,j matrix positions of the regions that contain this subplot
               matchidx <- as.data.frame(which(layout == i, arr.ind = TRUE))

               print(plots[[i]], vp = viewport(layout.pos.row = matchidx$row,
                                               layout.pos.col = matchidx$col))
          }
     }
}
```

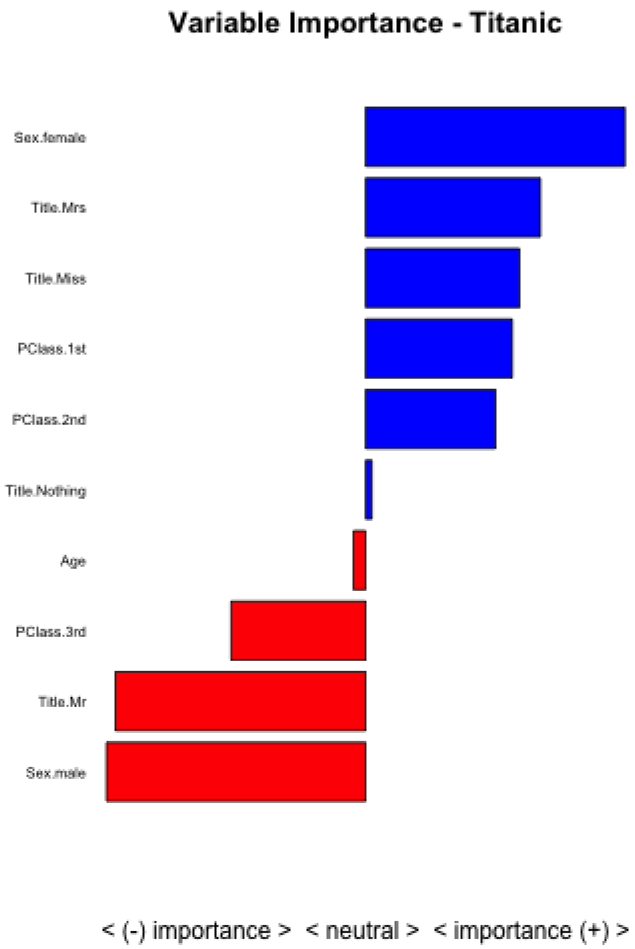
Now, let's plot the 4 strongest positive predictors for survivability and the 4 strongest negative ones:

```
p1 <- GetSummaryPlot(scaled_titanicDF_0, scaled_titanicDF_1, 'Sex.female', plotit=TRUE)

## [1] "Scaled spread: 5.253"

p2 <- GetSummaryPlot(scaled_titanicDF_0, scaled_titanicDF_1, 'Title.Mrs', plotit=TRUE)

## [1] "Scaled spread: 3.54"

p3 <- GetSummaryPlot(scaled_titanicDF_0, scaled_titanicDF_1, 'Sex.male', plotit=TRUE)

## [1] "Scaled spread: -5.253"

p4 <- GetSummaryPlot(scaled_titanicDF_0, scaled_titanicDF_1, 'Title.Mr', plotit=TRUE)

## [1] "Scaled spread: -5.071"

multiplot(p1,p2,p3,p4,cols=2)
```



And now for the final capper, let's plot everything on a bar graph so we can easily compare the strongest predictors and the direction they affect the model:

```
summaryImportance <- c()
variableName <- c()
for (predictorName in predictorNames) {
        summaryImportance <-  c(summaryImportance, GetSummaryPlot(scaled_titanicDF_0, scaled_titanicDF_1, predictorName, plotit=FALSE))
        variableName <- c(variableName, predictorName)
}
results <- data.frame('VariableName'=variableName, 'Weight'=summaryImportance)


# display variable importance on a +/- scale
results <- results[order(results$Weight),]
results <- results[(results$Weight != 0),]


par(mar=c(5,15,4,2)) # increase y-axis margin.
xx <- barplot(results$Weight, width = 0.85,
              main = paste("Variable Importance - Titanic"), horiz = T,
              xlab = "< (-) importance >  < neutral >  < importance (+) >", axes = FALSE,
              col = ifelse((results$Weight > 0), 'blue', 'red'))
axis(2, at=xx, labels=results$VariableName, tick=FALSE, las=2, line=-0.3, cex.axis=0.6)
```



Conclusion

This approach only works when attempting to predict a binary outcome and won't work for continuous outcomes (such as linear regression models). Also, it is a naive-based variable importance calculator as it doesn't account for interactions between predictors. Yet, it is a fast way to size up data for any supervised modeling and should handle extremely large dimensions. Happy explorations!

Full source code (also on GitHub):

```
# using dataset from the UCI Machine Learning Repository (http://archive.ics.uci.edu/ml/)
titanicDF <- read.csv('http://math.ucdenver.edu/RTutorial/titanic.txt',sep='\t')

# creating new title feature
```

```
titanicDF$Title <- ifelse(grepl('Mr ',titanicDF$Name),'Mr',ifelse(grepl('Mrs ',titanicDF$Name),'Mrs',ifelse(grepl('Miss',titanicDF$Name),'Miss','Nothing')))
titanicDF$Title <- as.factor(titanicDF$Title)

# impute age to remove NAs
titanicDF$Age[is.na(titanicDF$Age)] <- median(titanicDF$Age, na.rm=T)

# reorder data set so target is last column
titanicDF <- titanicDF[c('PClass', 'Age',    'Sex',   'Title', 'Survived')]

# binarize all factors
require(caret)
titanicDummy <- dummyVars("~.",data=titanicDF, fullRank=F)
titanicDF <- as.data.frame(predict(titanicDummy,titanicDF))

head(titanicDF, 3)

summary(titanicDF)

df_survived_1 <- subset(titanicDF, Survived==1)
df_survived_0 <- subset(titanicDF, Survived==0)
summary(df_survived_1$Sex.female)

summary(df_survived_0$Sex.female)

Sex.Female_0 <- (summary(df_survived_0$Sex.female))
Sex.Female_0 <- c(Sex.Female_0[1:6])
Sex.Female_1 <- (summary(df_survived_1$Sex.female))
Sex.Female_1 <- c(Sex.Female_1[1:6])
stats <- data.frame('ind'=c(1:6),
                    'Sex.Female_1'=Sex.Female_1,
                    'Sex.Female_0'=Sex.Female_0)

head(stats,6)

# plot Sex.Female predictor
require(ggplot2)
p <- ggplot(data=stats, aes(ind)) +
        geom_line(aes(y = Sex.Female_1, colour = "Sex.Female_1")) +
        geom_line(aes(y = Sex.Female_0, colour = "Sex.Female_0")) +
        scale_x_discrete(breaks = 1:6,
        labels=c("min","1q","median","mean","3q","max"))
p

# calculate predictor's spread value
spread <- ((Sex.Female_1[[1]] - Sex.Female_0[[1]]) +
                (Sex.Female_1[[2]] - Sex.Female_0[[2]]) +
                (Sex.Female_1[[3]] - Sex.Female_0[[3]]) +
                (Sex.Female_1[[4]] - Sex.Female_0[[4]]) +
                (Sex.Female_1[[5]] - Sex.Female_0[[5]]) +
                (Sex.Female_1[[6]] - Sex.Female_0[[6]]))
print(spread)

# do the same thing for predictor Title.Nothing
Title.Nothing_0 <- (summary(df_survived_0$Title.Nothing))
Title.Nothing_0 <- c(Title.Nothing_0[1:6])
Title.Nothing_1 <- (summary(df_survived_1$Title.Nothing))
Title.Nothing_1 <- c(Title.Nothing_1[1:6])
stats <- data.frame('ind'=c(1:6), 'stats1'=Title.Nothing_1,'stats0'=Title.Nothing_0)
spread <- ((Title.Nothing_1[[1]] - Title.Nothing_0[[1]]) +
        (Title.Nothing_1[[2]] - Title.Nothing_0[[2]]) +
        (Title.Nothing_1[[3]] - Title.Nothing_0[[3]]) +
        (Title.Nothing_1[[4]] - Title.Nothing_0[[4]]) +
        (Title.Nothing_1[[5]] - Title.Nothing_0[[5]]) +
        (Title.Nothing_1[[6]] - Title.Nothing_0[[6]]))
print(spread)

p <- ggplot(data=stats, aes(ind)) +
geom_line(aes(y = stats1, colour = "stats1")) +
geom_line(aes(y = stats0, colour = "stats0")) +
scale_x_discrete(breaks = 1:6,
labels=c("min","1q","median","mean","3q","max")) +
ylab('Title.Nothing') + xlab(paste('Spread:',spread))
p

# generalize this in a handy function
GetSummaryPlot <- function(objdfscaled0, objdfscaled1, predictorName, plotit=TRUE) {
        require(ggplot2)

        stats0 <- (summary(objdfscaled0[,predictorName]))
        stats0 <- c(stats0[1:6])
        stats1 <- (summary(objdfscaled1[,predictorName]))
        stats1 <- c(stats1[1:6])
        stats <- data.frame('ind'=c(1:6), 'stats1'=stats1,'stats0'=stats0)

        spread <- ((stats1[[1]] - stats0[[1]]) +
        (stats1[[2]] - stats0[[2]]) +
        (stats1[[3]] - stats0[[3]]) +
        (stats1[[4]] - stats0[[4]]) +
        (stats1[[5]] - stats0[[5]]) +
        (stats1[[6]] - stats0[[6]]))

        if (plotit) {
                # returns a ggplot
                print(paste('Scaled spread:',spread))
                p <- ggplot(data=stats, aes(ind)) +
                geom_line(aes(y = stats1, colour = "stats1")) +
                geom_line(aes(y = stats0, colour = "stats0")) +
                scale_x_discrete(breaks = 1:6,
                labels=c("min","1q","median","mean","3q","max")) +
                ylab(predictorName) + xlab(paste('Spread:',spread))
                return (p)
        } else {
                # only returns spread final value
                return (spread)
        }
}

outcomeName <- 'Survived'
predictorNames <- names(titanicDF)[!names(titanicDF) %in% outcomeName]
# Temporarily remove the outcome variable before scaling the data set
outcomeValue <- titanicDF$Survived
# scale returns a matrix so we need to tranform it back to a data frame
scaled_titanicDF <- as.data.frame(scale(titanicDF))
scaled_titanicDF$Survived <- outcomeValue
```

```
# split your data sets
scaled_titanicDF_0 <- scaled_titanicDF[scaled_titanicDF[,outcomeName]==0,]
scaled_titanicDF_1 <- scaled_titanicDF[scaled_titanicDF[,outcomeName]==1,]

for (predictorName in predictorNames)
        print(paste(predictorName,':',GetSummaryPlot(scaled_titanicDF_0,
        scaled_titanicDF_1, predictorName, plotit=FALSE)))

multiplot <- function(..., plotlist=NULL, file, cols=1, layout=NULL) {
     #http://www.cookbook-r.com/Graphs/Multiple_graphs_on_one_page_(ggplot2)/
     # Multiple plot function
     #
     # ggplot objects can be passed in ..., or to plotlist (as a list of ggplot objects)
     # - cols:   Number of columns in layout
     # - layout: A matrix specifying the layout. If present, 'cols' is ignored.
     #
     # If the layout is something like matrix(c(1,2,3,3), nrow=2, byrow=TRUE),
     # then plot 1 will go in the upper left, 2 will go in the upper right, and
     # 3 will go all the way across the bottom.
     #
     require(grid)

     # Make a list from the ... arguments and plotlist
     plots <- c(list(...), plotlist)

     numPlots = length(plots)

     # If layout is NULL, then use 'cols' to determine layout
     if (is.null(layout)) {
             # Make the panel
             # ncol: Number of columns of plots
             # nrow: Number of rows needed, calculated from # of cols
             layout <- matrix(seq(1, cols * ceiling(numPlots/cols)),
                              ncol = cols, nrow = ceiling(numPlots/cols))
     }

     if (numPlots==1) {
             print(plots[[1]])
     } else {
             # Set up the page
             grid.newpage()
             pushViewport(viewport(layout = grid.layout(nrow(layout), ncol(layout))))

             # Make each plot, in the correct location
             for (i in 1:numPlots) {
                     # Get the i,j matrix positions of the regions that contain this subplot
                     matchidx <- as.data.frame(which(layout == i, arr.ind = TRUE))

                     print(plots[[i]], vp = viewport(layout.pos.row = matchidx$row,
                                                     layout.pos.col = matchidx$col))
             }
     }
}

p1 <- GetSummaryPlot(scaled_titanicDF_0, scaled_titanicDF_1, 'Sex.female', plotit=TRUE)
p2 <- GetSummaryPlot(scaled_titanicDF_0, scaled_titanicDF_1, 'Title.Mrs', plotit=TRUE)
p3 <- GetSummaryPlot(scaled_titanicDF_0, scaled_titanicDF_1, 'Sex.male', plotit=TRUE)
p4 <- GetSummaryPlot(scaled_titanicDF_0, scaled_titanicDF_1, 'Title.Mr', plotit=TRUE)
multiplot(p1,p2,p3,p4,cols=2)

summaryImportance <- c()
variableName <- c()
for (predictorName in predictorNames) {
     summaryImportance <-  c(summaryImportance, GetSummaryPlot(scaled_titanicDF_0, scaled_titanicDF_1, predictorName, plotit=FALSE))
     variableName <- c(variableName, predictorName)
}
results <- data.frame('VariableName'=variableName, 'Weight'=summaryImportance)

# display variable importance on a +/- scale
results <- results[order(results$Weight),]
results <- results[(results$Weight != 0),]

par(mar=c(5,15,4,2)) # increase y-axis margin.
xx <- barplot(results$Weight, width = 0.85,
           main = paste("Variable Importance - Titanic"), horiz = T,
           xlab = "< (-) importance >  < neutral >  < importance (+) >", axes = FALSE,
           col = ifelse((results$Weight > 0), 'blue', 'red'))
axis(2, at=xx, labels=results$VariableName, tick=FALSE, las=2, line=-0.3, cex.axis=0.6)
```

Using String Distance {stringdist} To Handle Large Text Factors, Cluster Them into Supersets

November 30, 2014 Tags: *exploring*, *visualizing*

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{stringdist}** - feature selection for ensembles
- **{RCurl}** - machine learning tools

If you're wondering whether you're getting the most out of a **text-based, factor variable** from a large data set, then you're not alone. There are so many ways of deconstructing text variables. If every entry is made of repeated text from a small set of possibilities, then dummifying it is the easiest way to proceed. On the other hand, if every entry is unique, then resorting to Natural Language Processing (NLP) may be required. This article tackles the gray area in between, where the data is neither unique nor small, where dummifying won't work yet NLP may still be avoided.

So that we are on the same page, imagine a data set with 10 million rows with at least one feature/column being a text-based factor. It is not made up of free-text, where every entry is unique, instead, it is made up of repeated text: for example 10,000 possibilities repeated over 10 million rows. This would be hard to dummify, as it will blow up your feature space, and would take forever to group by hand.

What Is One To Do?

- We could encode the text entries as integers or binaries and hope for the best (as it is not ordinal in nature, linear models would suffer but classification models would be OK).
- We could take the top X most popular ones and overwrite the rest as 'other' and dummify the resulting set (I have used that method many times and will write up a post on the subject).

- But a more interesting approach that affords much less loss of information, is grouping them into supersets and the subject of this walkthrough.

Grouping With {stringdist}

Could those 10,000 possibilities mentioned earlier be grouped into a superset representing only a tenth or a fifth of its original size? What is close to impossible to do by hand is trivial with string distance:

...a metric that measures distance ("inverse similarity") between two text-strings for approximate string matching or comparison and in fuzzy string searching. (Source: Wikipedia)

The {strndist} package offers 'Approximate string matching and string distance functions'. It offers many algorithms but the two I found the most interesting for short sets of words are:

...the **Jaro–Winkler distance** (Winkler, 1990) is a measure of similarity between two strings. The higher the Jaro–Winkler distance for two strings is, the more similar the strings are. The Jaro–Winkler distance metric is designed and best suited for short strings such as person names. The score is normalized such that 0 equates to no similarity and 1 is an exact match. (Source: Wikipedia)

and

...the **Levenshtein distance** between two words is the minimum number of single-character edits (i.e. insertions, deletions or substitutions) required to change one word into the other.
(Source: Wikipedia)

Let's Code!

Enough chitchat, let's download the vehicles data set from Hadley Wickham hosted on Github. It is a big and diverse data set, perfect for our needs:

```
library(RCurl)

urlfile <-'https://raw.githubusercontent.com/hadley/fueleconomy/master/data-raw/vehicles.csv'

x <- getURL(urlfile, ssl.verifypeer = FALSE)

vehicles <- read.csv(textConnection(x))


# alternative way of getting the data if the above snippet doesn't work:

# urlData <- getURL('https://raw.githubusercontent.com/hadley/fueleconomy/master/data-raw/vehicles.csv')
# vehicles <- read.csv(text = urlData)
```

We're going to focus on one single feature in the data set: `model`. Let's start with some basic statistics on that feature to understand what we're dealing with:

```
nrow(vehicles)

[1] 34631

length(unique(vehicles$model))

[1] 3234
```

So, we have a data set of over **34,631** vehicles, but `model` is comprised of only **3,234** unique model names repeated through out all the observations/rows. Let's look at the first **100**rows so we don't get overwhelmed with too much the data:

```
vehicles_small <- vehicles[1:100,]
```

Out of those **100** observations, `model` has only **45** unique model names and here is a small sample of what it holds:

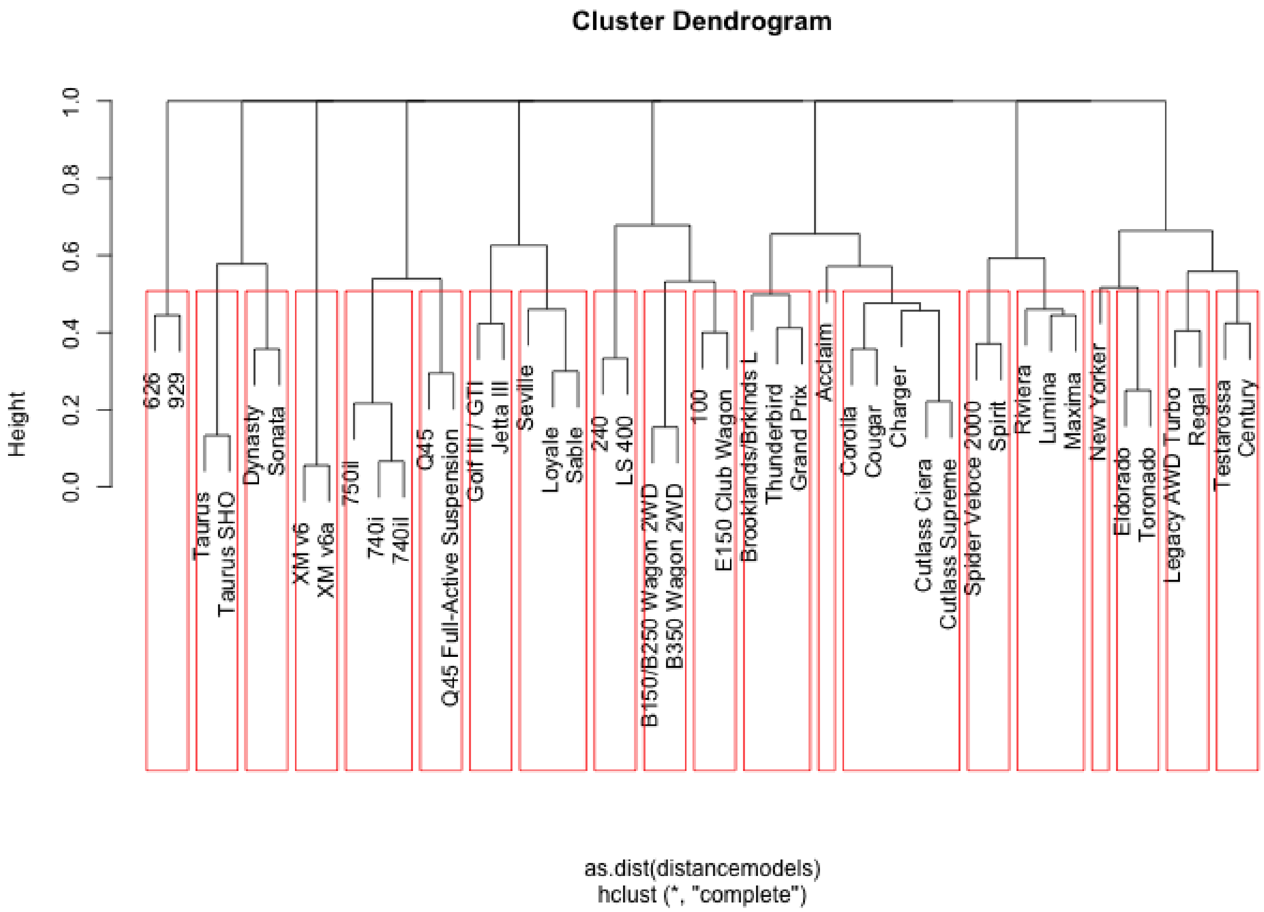
```
length(unique(vehicles_small$model))

[1] 45

head(unique(as.character(vehicles_small$model)))

[1] "Spider Veloce 2000"  "Testarossa"          "Charger"

[4] "B150/B250 Wagon 2WD" "Legacy AWD Turbo"    "Loyale"
```

Let's run some basic string distance on this subset by calling the `stringdistmatrix` function to see how it can help us tame the `model` variable into smaller supersets. In its simplest form, the function `stringdistmatrix` only requires a unique set of text values and the method to cluster the data:

```
stringdistmatrix(a, b, method = c("osa", "lv", "dl", "hamming", "lcs",

        "qgram", "cosine", "jaccard", "jw", useBytes = FALSE,

        weight = c(d = 1, i = 1, s = 1, t = 1), maxDist = Inf, q = 1, p = 0,

        useNames = FALSE, ncores = 1, cluster = NULL)
```

We'll pass it the unique list of `models`, request the **Jaro–Winkler distance** algorithm (my favorite for this task), cluster the results into **20** groups with the hclust function and plot the resulting dendrogram:

```
library(stringdist)
uniquemodels <- unique(as.character(vehicles_small$model))
distancemodels <- stringdistmatrix(uniquemodels,uniquemodels,method = "jw")
rownames(distancemodels) <- uniquemodels
hc <- hclust(as.dist(distancemodels))
plot(hc)
rect.hclust(hc,k=20)
```



`stringdistmatrix` works in tandem with `hclust`, one creates the model, the other enforces the clusters. Our algorithm created a **Wagon** and **Taurus** group along with some number groups. So far it isn't extremely impressive as its only using a small subset of data - wait till we open things up!

We now look at a bigger subset of the **vehicles** data set. Let's pull the first 2000 observations:

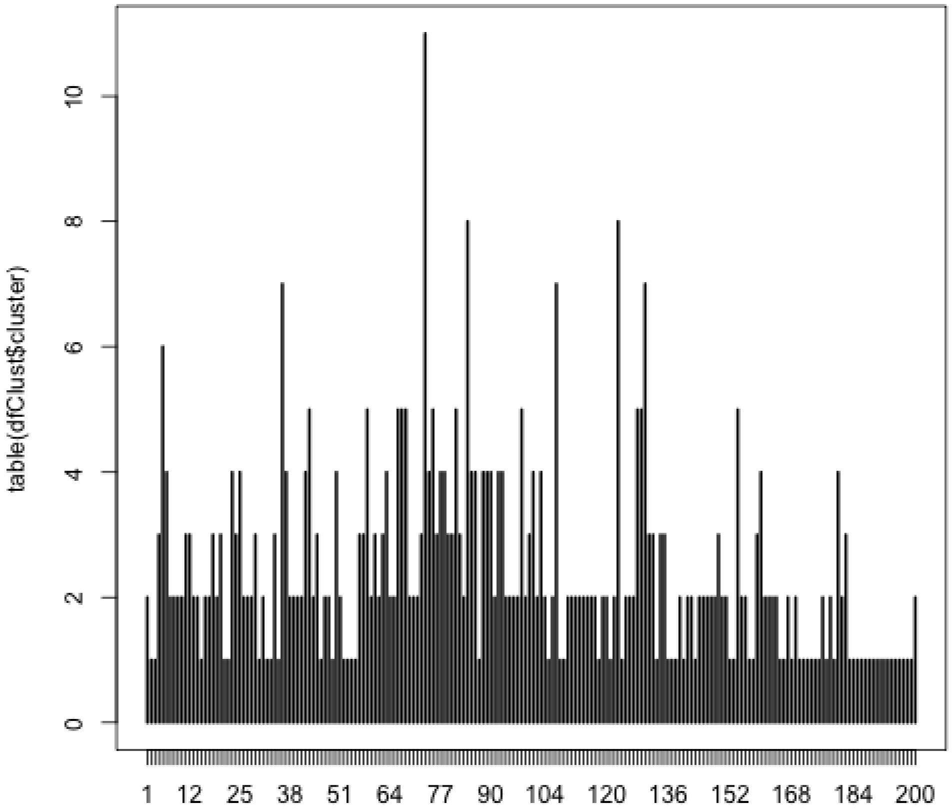
```
vehicles_small <- vehicles[1:2000,]
length(unique(vehicles_small$model))
[1] 481
```

Out of those 2000 observations, we have 481 unique model names. Let's ask `stringdistmatrix` to group those into 200 groups:

```
uniquemodels <- unique(as.character(vehicles_small$model))
distancemodels <- stringdistmatrix(uniquemodels,uniquemodels,method = "jw")
rownames(distancemodels) <- uniquemodels
hc <- hclust(as.dist(distancemodels))
dfClust <- data.frame(uniquemodels, cutree(hc, k=200))
names(dfClust) <- c('modelname','cluster')
```

Let's visualize the quantities of models in each group created by the **Jaro–Winkler distance** algorithm:

```
plot(table(dfClust$cluster))
```



```
print(paste('Average number of models per cluster:', mean(table(dfClust$cluster))))

## [1] "Average number of models per cluster: 2.405"
```

The largest cluster contains over **10** models but the average is **2.4** models per cluster. Now, lets look at the top groups and see what the algorithm did (don't sweat this code, it simply orders the data by cluster size):

```
t <- table(dfClust$cluster)
t <- cbind(t,t/length(dfClust$cluster))
t <- t[order(t[,2], decreasing=TRUE),]
p <- data.frame(factorName=rownames(t), binCount=t[,1], percentFound=t[,2])
dfClust <- merge(x=dfClust, y=p, by.x = 'cluster', by.y='factorName', all.x=T)
dfClust <- dfClust[rev(order(dfClust$binCount)),]
names(dfClust) <-  c('cluster','modelname')
head (dfClust[c('cluster','modelname')],50)
##     cluster              modelname
## 192      73       K1500 Pickup 4WD
## 191      73         S10 Pickup 2WD
## 190      73        W250 Pickup 4WD
## 189      73        F150 Pickup 2WD
## 188      73         S10 Pickup 4WD
## 187      73   D100/D150 Pickup 2WD
## 186      73        F250 Pickup 2WD
## 185      73       C1500 Pickup 2WD
## 184      73        F150 Pickup 4WD
## 183      73        D250 Pickup 2WD
## 182      73   W100/W150 Pickup 4WD
## 341     123 Postal Cab Chassis 2WD
## 340     123    S10 Cab Chassis 2WD
## 339     123 Dakota Cab Chassis 2WD
## 338     123        Cab/Chassis 2WD
## 337     123        Cab Chassis 2WD
## 336     123    S15 Cab Chassis 2WD
## 335     123  Truck Cab Chassis 2WD
## 334     123   D250 Cab Chassis 2WD
## 236      84         Yukon 1500 4WD
## 235      84       Suburban C10 2WD
## 234      84            SJ 410V 4WD
## 233      84      Suburban 1500 2WD
## 232      84       Suburban K10 4WD
## 231      84        Yukon K1500 4WD
## 230      84             SJ 410 4WD
## 229      84      Suburban 1500 4WD
## 365     130        900 Convertible
## 364     130       318i Convertible
## 363     130            Convertible
## 362     130    XJS V12 Convertible
## 361     130       E320 Convertible
## 360     130       325i Convertible
## 359     130        XJS Convertible
## 307     107     Sidekick 2Door 2WD
## 306     107   Sidekick Hardtop 2WD
## 305     107     Sidekick 4Door 2WD
## 304     107     Sidekick 2Door 4WD
## 303     107           Sidekick 2WD
## 302     107   Sidekick Hardtop 4WD
## 301     107     Sidekick 4Door 4WD
## 86       36              240 Wagon
## 85       36             E320 Wagon
## 84       36              940 Wagon
## 83       36              850 Wagon
## 82       36              960 Wagon
## 81       36        E150 Club Wagon
## 80       36              100 Wagon
## 13        5       Legacy AWD Turbo
## 12        5           Legacy Wagon
```

Out of the **200** clusters we requested, cluster 73 is the largest holding **11** models. Clearly, it picked up on the word **pickup** flanked by two words on either side with the right one being**2WD** or **4WD**. Cluster 123 looked for **Cab Chassis**, even picking up a **Cab/Chassis** in the process. You get the idea and, hopefully, are impressed with how a few lines of code reduced**2000** observations into **200** groups. The exact same process would apply to **20,000** observations or **20** million...

Creating New Variables through Combining Multiple Features

An offshoot of this process is to create new groups by combining existing features and running the results through `stringdistmatrix`. Let's try combining `model` with `trany`:

```
vehicles_small$modelAndTrany <- paste0(as.character(vehicles_small$model)," ",as.character(vehicles_small$
trany))

print(length(unique(vehicles_small$modelAndTrany)))

## [1] 808
```

Our new field has **808** unique values out of our **2000** `small_vehicles` data frame. Let's run it through the **Jaro–Winkler distance** algorithm, request **500** clusters and check out the top groups:

```
uniquemodels <- unique(as.character(vehicles_small$modelAndTrany))

distancemodels <- stringdistmatrix(uniquemodels,uniquemodels,method = "jw")

rownames(distancemodels) <- uniquemodels

hc <- hclust(as.dist(distancemodels))

dfClust <- data.frame(uniquemodels, cutree(hc, k=500))

names(dfClust) <- c('modelname','cluster')

t <- table(dfClust$cluster)

t <- cbind(t,t/length(dfClust$cluster))

t <- t[order(t[,2], decreasing=TRUE),]

p <- data.frame(factorName=rownames(t), binCount=t[,1], percentFound=t[,2])

dfClust <- merge(x=dfClust, y=p, by.x = 'cluster', by.y='factorName', all.x=T)

dfClust <- dfClust[rev(order(dfClust$binCount)),]

names(dfClust) <-  c('cluster','modelname')

head (dfClust[c('cluster','modelname')],50)

##     cluster                           modelname

## 38       16                 960 Automatic 4-spd

## 37       16                  90 Automatic 4-spd

## 36       16                 940 Automatic 4-spd

## 35       16                 900 Automatic 4-spd

## 34       16                E500 Automatic 4-spd

## 33       16                 100 Automatic 4-spd
## 32       16                9000 Automatic 4-spd
## 31       16                 850 Automatic 4-spd
## 27       14                 G20 Automatic 4-spd
## 26       14                 240 Automatic 4-spd
## 25       14                S420 Automatic 4-spd
## 24       14               240SX Automatic 4-spd
## 23       14                C280 Automatic 4-spd
## 22       14                C220 Automatic 4-spd
## 21       14                E420 Automatic 4-spd
## 221     120           960 Wagon Automatic 4-spd
## 220     120          E320 Wagon Automatic 4-spd
## 219     120           850 Wagon Automatic 4-spd
## 218     120           240 Wagon Automatic 4-spd
## 217     120           100 Wagon Automatic 4-spd
## 216     120           940 Wagon Automatic 4-spd
## 185      99                     SW Manual 5-spd
## 184      99                     S4 Manual 5-spd
## 183      99                     S6 Manual 5-spd
## 182      99                    NSX Manual 5-spd
## 181      99                     SL Manual 5-spd
## 180      99                     SC Manual 5-spd
## 248     129    Ram 1500 Pickup 4WD Manual 5-spd
## 247     129    Ram 2500 Pickup 2WD Manual 5-spd
## 246     129    Ram 2500 Pickup 4WD Manual 5-spd
## 245     129    Ram 1500 Pickup 2WD Manual 5-spd
## 244     129      Ram 50 Pickup 2WD Manual 5-spd
## 243     128 Ram 1500 Pickup 2WD Automatic 4-spd
## 242     128 Ram 2500 Pickup 4WD Automatic 4-spd
## 241     128   Ram 50 Pickup 2WD Automatic 4-spd
## 240     128 Ram 1500 Pickup 4WD Automatic 4-spd
## 239     128 Ram 2500 Pickup 2WD Automatic 4-spd
## 177      97                 NSX Automatic 4-spd
## 176      97                  SC Automatic 4-spd
## 175      97                 SVX Automatic 4-spd
## 174      97                  SW Automatic 4-spd
## 173      97                  SL Automatic 4-spd
## 154      83               SL600 Automatic 4-spd
## 153      83              500SEL Automatic 4-spd
## 152      83               SL500 Automatic 4-spd
## 151      83              400SEL Automatic 4-spd
## 150      83               500SL Automatic 4-spd
## 47       18                540i Automatic 5-spd
## 46       18               840ci Automatic 5-spd
## 45       18               740il Automatic 5-spd
```

Conclusion

`stringdistmatrix` is a very flexible function with many tunable features. The cluster size, the algorithm, the concatenation of text with text and/or numbers create numerous and mind-boggling possibilities. Even with all these settings it is still so much easier than creating supersets by hand! String distance matching is case sensitive so you may get better groups if you force all text to once case. Have fun with this...

Full source code (also on GitHub):

```
# get the Hadley Wickham's vehicles data set
library(RCurl)
urlfile <-'https://raw.githubusercontent.com/hadley/fueleconomy/master/data-raw/vehicles.csv'
x <- getURL(urlfile, ssl.verifypeer = FALSE)
vehicles <- read.csv(textConnection(x))

# size the data
nrow(vehicles)
length(unique(vehicles$model))

# get a small sample for starters
vehicles_small <- vehicles[1:100,]
length(unique(vehicles_small$model))
head(unique(as.character(vehicles_small$model)))

# call the stringdistmatrix function and request 20 groups
library(stringdist)
uniquemodels <- unique(as.character(vehicles_small$model))
distancemodels <- stringdistmatrix(uniquemodels,uniquemodels,method = "jw")
```

```
rownames(distancemodels) <- uniquemodels
hc <- hclust(as.dist(distancemodels))

# visualize the dendrogram
plot(hc)
rect.hclust(hc,k=20)

# get a bigger sample
vehicles_small <- vehicles[1:2000,]
length(unique(vehicles_small$model))

# run the stringdistmatrix function and request 200 groups
uniquemodels <- unique(as.character(vehicles_small$model))
distancemodels <- stringdistmatrix(uniquemodels,uniquemodels,method = "jw")
rownames(distancemodels) <- uniquemodels
hc <- hclust(as.dist(distancemodels))
dfClust <- data.frame(uniquemodels, cutree(hc, k=200))
names(dfClust) <- c('modelname','cluster')

# visualize the groupings
plot(table(dfClust$cluster))
print(paste('Average number of models per cluster:', mean(table(dfClust$cluster))))

# lets look at the top groups and see what the algorithm did:
t <- table(dfClust$cluster)
t <- cbind(t,t/length(dfClust$cluster))
t <- t[order(t[,2], decreasing=TRUE),]
p <- data.frame(factorName=rownames(t), binCount=t[,1], percentFound=t[,2])
dfClust <- merge(x=dfClust, y=p, by.x = 'cluster', by.y='factorName', all.x=T)
dfClust <- dfClust[rev(order(dfClust$binCount)),]
names(dfClust) <-  c('cluster','modelname')
head (dfClust[c('cluster','modelname')],50)

# try combining fields together
vehicles_small$modelAndTrany <- paste0(as.character(vehicles_small$model)," ",as.character(vehicles_small$trany))
print(length(unique(vehicles_small$modelAndTrany)))

uniquemodels <- unique(as.character(vehicles_small$modelAndTrany))
```

```
distancemodels <- stringdistmatrix(uniquemodels,uniquemodels,method = "jw")
rownames(distancemodels) <- uniquemodels
hc <- hclust(as.dist(distancemodels))
dfClust <- data.frame(uniquemodels, cutree(hc, k=500))
names(dfClust) <- c('modelname','cluster')
t <- table(dfClust$cluster)
t <- cbind(t,t/length(dfClust$cluster))
t <- t[order(t[,2], decreasing=TRUE),]
p <- data.frame(factorName=rownames(t), binCount=t[,1], percentFound=t[,2])
dfClust <- merge(x=dfClust, y=p, by.x = 'cluster', by.y='factorName', all.x=T)
dfClust <- dfClust[rev(order(dfClust$binCount)),]
names(dfClust) <-  c('cluster','modelname')
head (dfClust[c('cluster','modelname')],50)

# build a convenient function to do all of the above
GroupFactorsTogether <- function(objData, variableName, clustersize=200, method='jw') {
        #      osa: Optimal string aligment, (restricted Damerau-Levenshtein distance).
        #      lv: Levenshtein distance (as in R's native adist).
        #      dl: Full Damerau-Levenshtein distance.
        #      hamming: Hamming distance (a and b must have same nr of characters).
        #      lcs: Longest common substring distance.
        #      qgram: q-gram distance.
        #      cosine: cosine distance between q-gram profiles
        #      jaccard: Jaccard distance between q-gram profiles
        #      jw: Jaro, or Jaro-Winker distance.
        #      soundex: Distance based on soundex encoding

        #       stringdistmatrix(a, b, method = c("osa", "lv", "dl", "hamming", "lcs",
        #               "qgram", "cosine", "jaccard", "jw", useBytes = FALSE,
        #               weight = c(d = 1, i = 1, s = 1, t = 1), maxDist = Inf, q = 1, p = 0,
        #               useNames = FALSE, ncores = 1, cluster = NULL)
        #               require(stringdist)

        str <- unique(as.character(objData[,variableName]))
        print(paste('Uniques:',length(str)))

        d <- stringdistmatrix(str,str,method = c(method))
        rownames(d) <- str
        hc <- hclust(as.dist(d))

        dfClust <- data.frame(str, cutree(hc, k=clustersize))

        plot(table(dfClust$'cutree.hc..k...k.'))

        most_populated_clusters <- dfClust[dfClust$'cutree.hc..k...k.' > 5,]
        names(most_populated_clusters) <- c('entry','cluster')

        # sort by most frequent
        t <- table(most_populated_clusters$cluster)
        t <- cbind(t,t/length(most_populated_clusters$cluster))
        t <- t[order(t[,2], decreasing=TRUE),]
        p <- data.frame(factorName=rownames(t), binCount=t[,1], percentFound=t[,2])
        most_populated_clusters <- merge(x=most_populated_clusters, y=p, by.x = 'cluster', by.y='factorName', all.x=T)
        most_populated_clusters <- most_populated_clusters[rev(order(most_populated_clusters$binCount)),]
        names(most_populated_clusters) <-  c('cluster','entry')
        return (most_populated_clusters[c('cluster','entry')])
}
```

Downloading Data from Google Trends and Analyzing It with R

November 17, 2014 Tags: *exploring*, *visualizing*

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{ggplot2}** - graphics

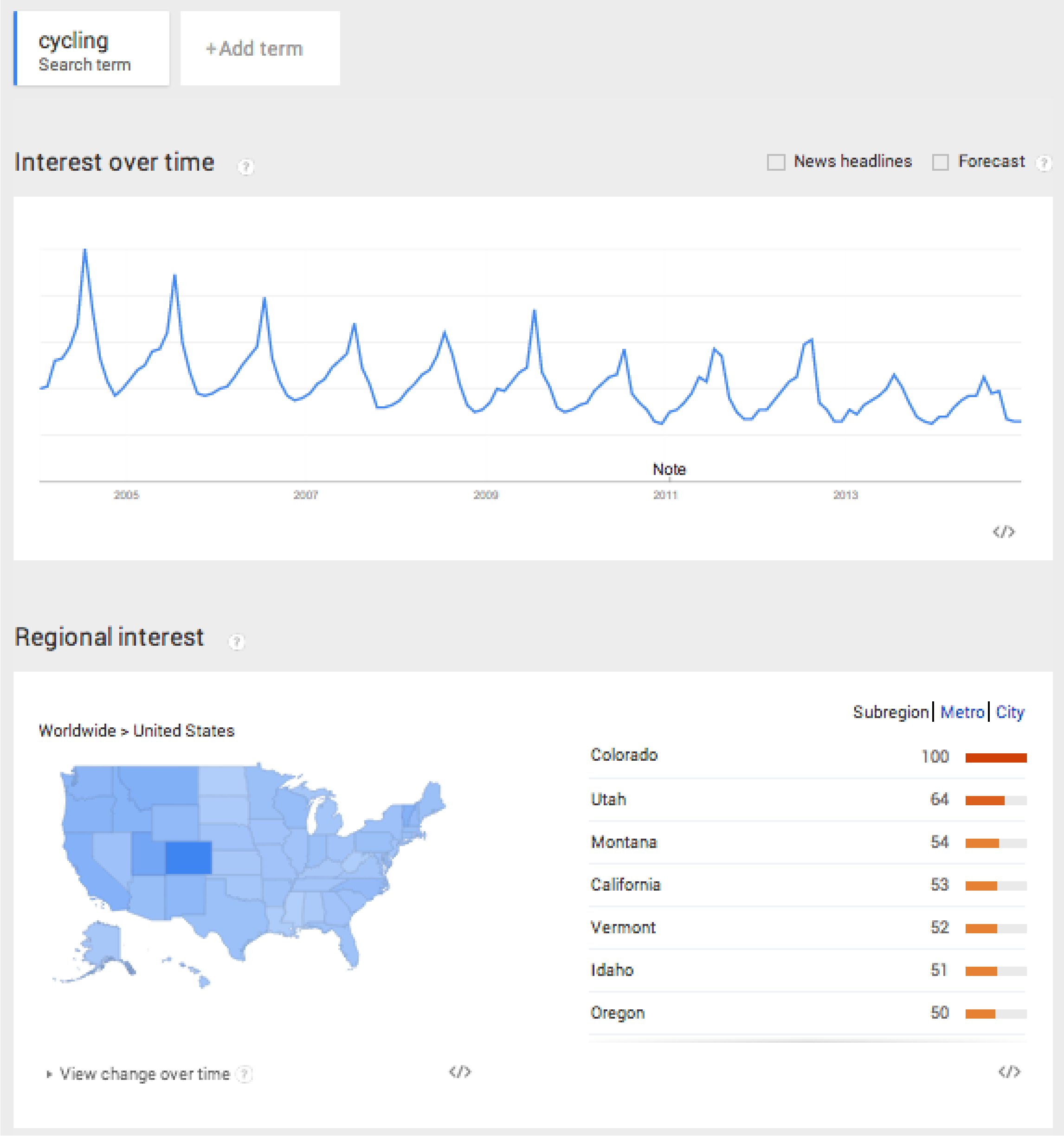
Google Trends has been around, in one form or another, for many years. Yet, it wasn't until I needed free Internet sentiment data that I took a closer look at this service and have been loving it ever since. With this tool you can easily target a time or region and find out what is trending up, down, or not at all - imagine all the possibilities!

Google Trends is a public web facility of Google Inc., based on Google Search that shows how often a particular search-term is entered relative to the total search-volume across various regions of the world, and in various languages. (Source: Wikipedia.com)

In this walkthrough, I introduce the tool by accessing it directly through a web browser to extract data and analyze it in R.

Google Trends

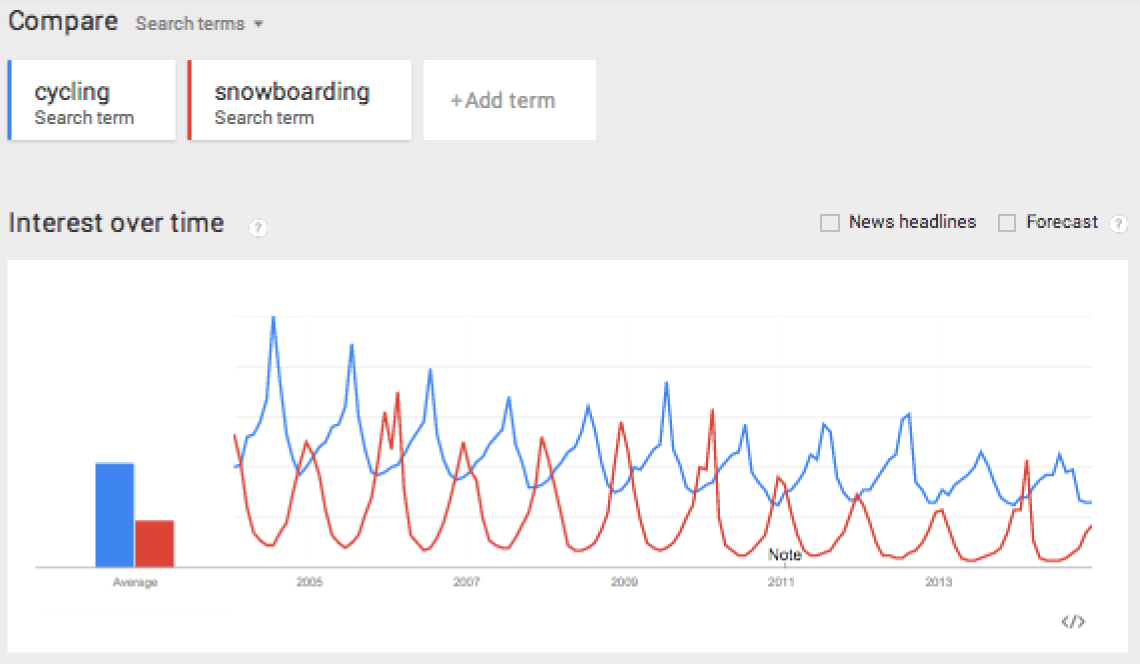
Let's start by entering the term `cycling` and limiting our scope to the United States. There seems to be a decline in usage for that term from 2005 to 2014 as the oscillations are constant but the overall trend is dropping.



Colorado	100
Utah	64
Montana	54
California	53
Vermont	52
Idaho	51
Oregon	50

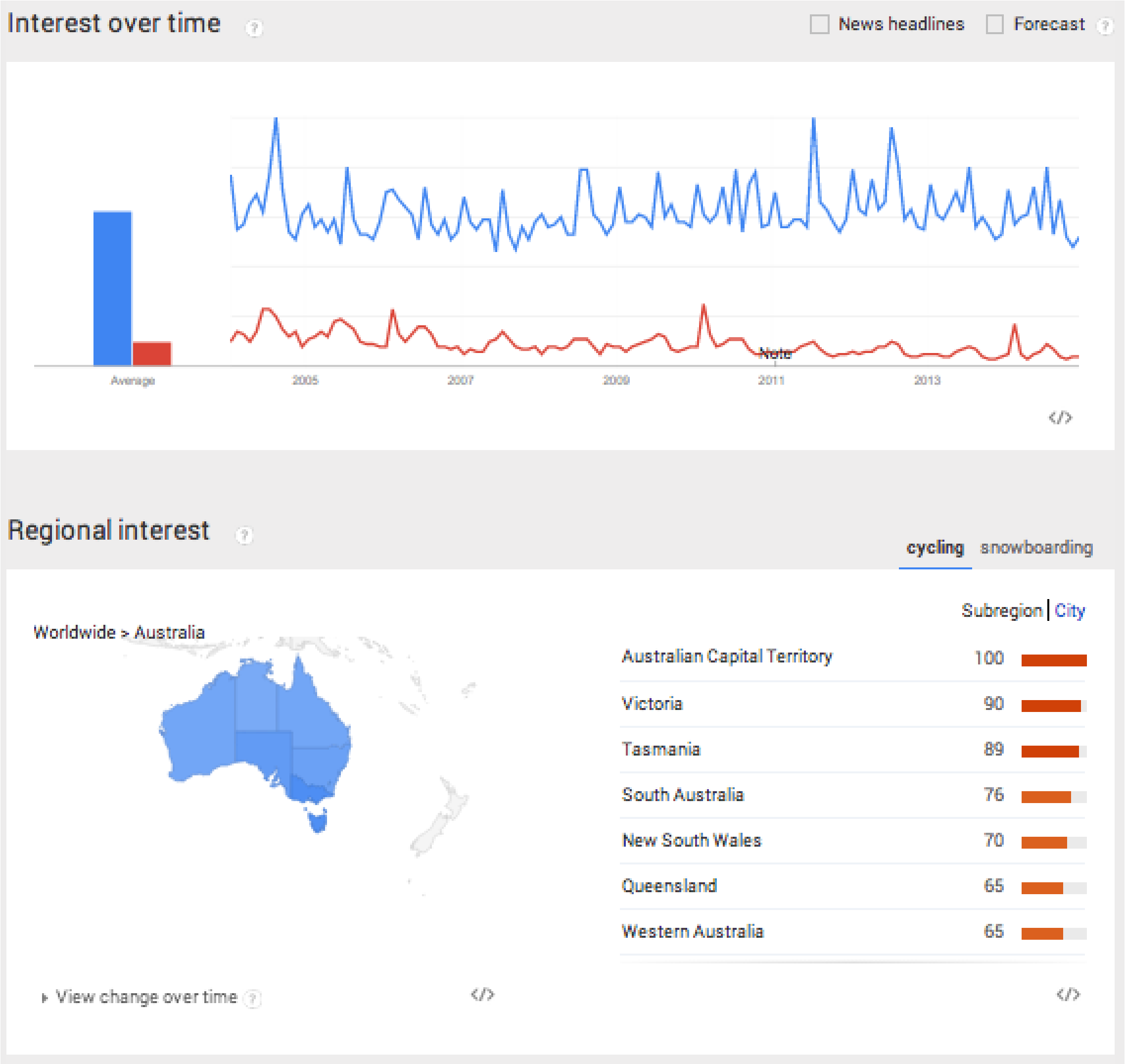
The highest peak is July 2014 and represents the 100% maximum search for the term. Everything else is scaled from that peak, and that is how Google Trends displays a single search term over time (i.e. nothing will be over 100 in the graph). The term also peaks with clockwork regularity every summer. This decline can mean that people's interest in cycling is declining, that the term cycling in the English language is replaced by another more popular term, or that cyclists aren't using Google like they used to (your theory is as good as mine).

Let's make things more interesting and add a second search term to our graph. Let's add the term `snowboarding`.



This creates a mirror image of `cycling`, where the peaks of one term are the troughs of the other. `Snowboarding` peaks every December, and, unlike cycling, the term hasn't dropped over the years.

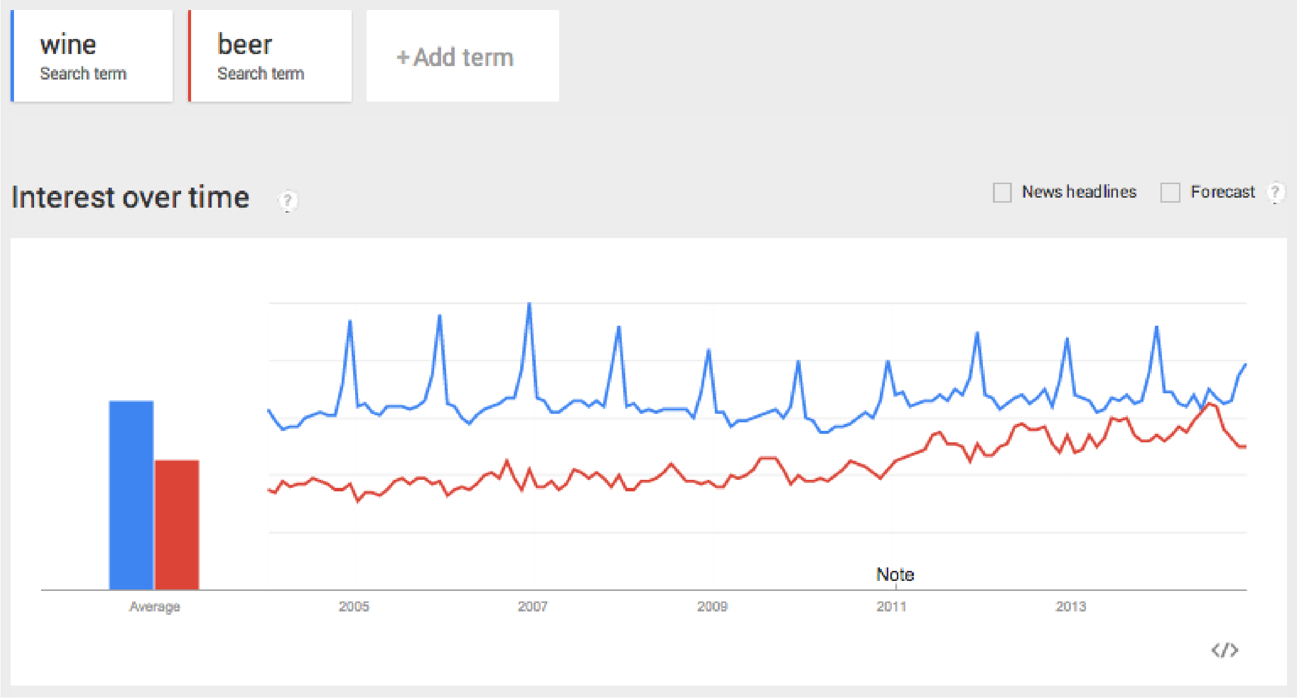
So far, we’ve seen two interesting pieces of data using Google Trends: the term's popularity and its seasonal effect. There is plenty more to explore and compare as trends can be narrowed by time, region and city. Here we see both terms applied to `Australia`. Clearly, `cycling` is more popular than `snowboarding`...



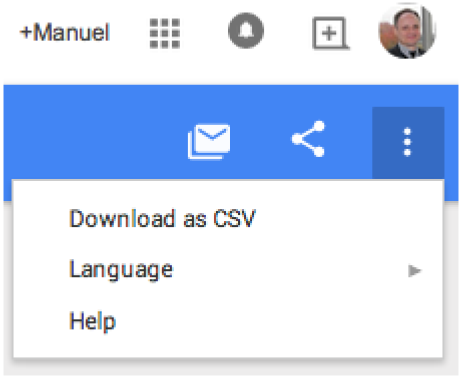
Australian Capital Territory	100
Victoria	90
Tasmania	89
South Australia	76
New South Wales	70
Queensland	65
Western Australia	65

Let's Code!

OK, let's pull some data and analyze it in **R**. Query `wine` as the first term and `beer` as the second one and limit it to the US:



Download the **csv** file from the dropdown located in the upper right corner:



Now, let's access the file through **R**:

```
filename <- "beervswine.csv"
```

If you open the csv file you will notice all sorts of information there. In order to just pull the main time series we need to loop through each line and start pulling the data at the 5th line and stop pulling as soon as we encounter empty fields.

```
con  <- file(filename, open = "r")
linecount <- 0
stringdata <- ""
while (length(oneLine <- readLines(con, n = 1, warn = FALSE)) > 0) {
        linecount <- linecount + 1

        if (linecount < 3) {
                filename <- paste0(filename,oneLine)
        }

        # get headers at line 5
        if (linecount == 5) rowheaders = strsplit(oneLine, ",")[[1]]

        # skip firt 5 lines
        if (linecount > 5) {
                # break when there is no more main data
                if (gsub(pattern=",", x=oneLine, replacement="") == "") break

                stringdata <- paste0(stringdata,oneLine,"\n")
        }
}
close(con)
```

There are a few caveats worth talking about when working with **Google Trends** data. It can come in three time flavors: monthly, weekly, and daily. To get daily data, you need to query less than 3 months timespan. For longer term trends, you will usually get weekly data unless it is low popularity, and then you will get montly data. One more point, if you query multiple terms and some are don't return enough data, the csv will automatically exclude them.

```
> head(newData)
                       V1 V2 V3
1 2004-01-04 - 2004-01-10 31 57
2 2004-01-11 - 2004-01-17 30 54
3 2004-01-18 - 2004-01-24 31 57
4 2004-01-25 - 2004-01-31 30 50
5 2004-02-01 - 2004-02-07 29 47
6 2004-02-08 - 2004-02-14 30 53
```

In order to avoid the uncertainties of the final exported format, it is best to not hard code anything. To circumvent all this, we read the data line by line and store it all in one long string `stringdata` and add a line feed at the end of each line. We then use the `read.table` with `textConnection` to parse the `stringdata` into a flat file and append the dynamic column names pulled from line 5 of the csv. This allows us to get the correct header names whether Google returns `1` or `10` features/columns - this should limit surprises especially when working with multiple downloads.

```
newData <- read.table(textConnection(stringdata), sep=",", header=FALSE, stringsAsFactors = FALSE)
names(newData) <- rowheaders

newData$StartDate <- as.Date(sapply(strsplit(as.character(newData[,1]), " - "), `[`, 1))
newData$EndDate <- as.Date(sapply(strsplit(as.character(newData[,1]), " - "), `[`, 2))
newData$year <- sapply(strsplit(as.character(newData$StartDate), "-"), `[`, 1)
newData<- newData[c("StartDate", "EndDate", "beer", "wine", "year")]
```

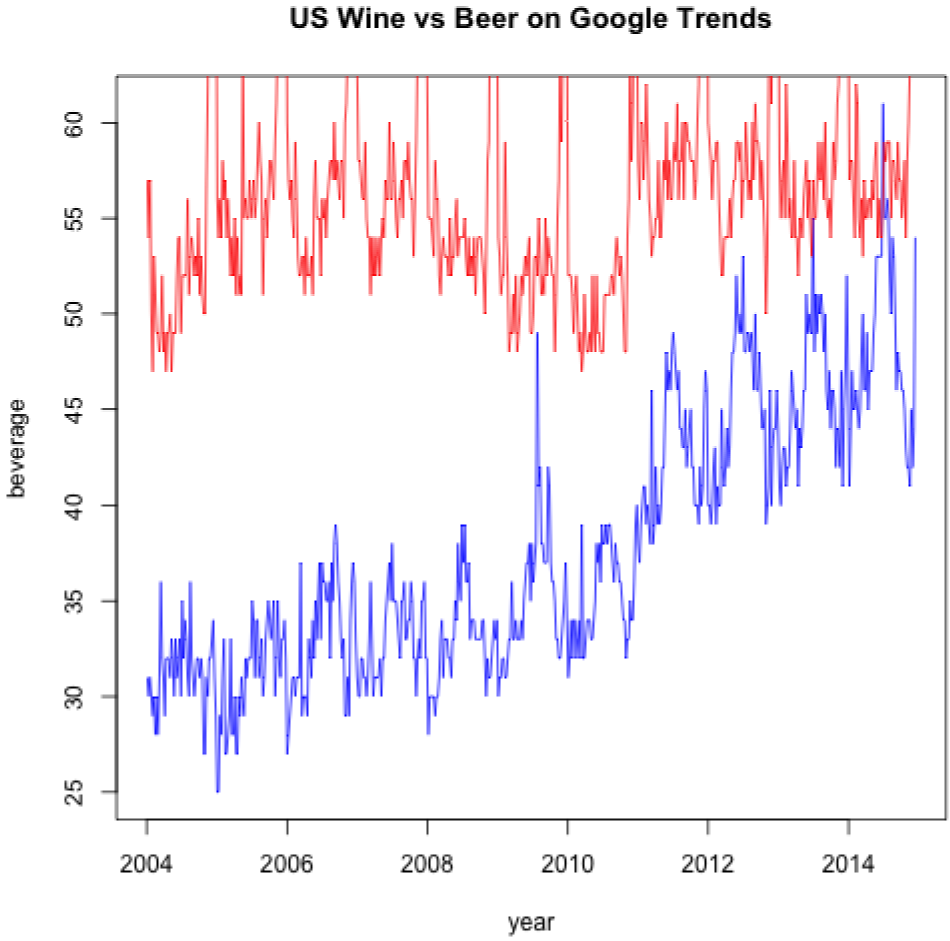
Google Trends returns date ranges for each observations (rows), so we need to separate those into a `StartDate` and `EndDate` column. I also create a `year` column from the start date for use in boxplots.

```
> head(newData)
   StartDate    EndDate beer wine year wine_clean
1 2004-01-04 2004-01-10   31   57 2004         57
2 2004-01-11 2004-01-17   30   54 2004         54
3 2004-01-18 2004-01-24   31   57 2004         57
4 2004-01-25 2004-01-31   30   50 2004         50
5 2004-02-01 2004-02-07   29   47 2004         47
6 2004-02-08 2004-02-14   30   53 2004         53
```

Plots

Here we confirm that the data is correct and should be similar to what we saw earlier on **Google Trends**.

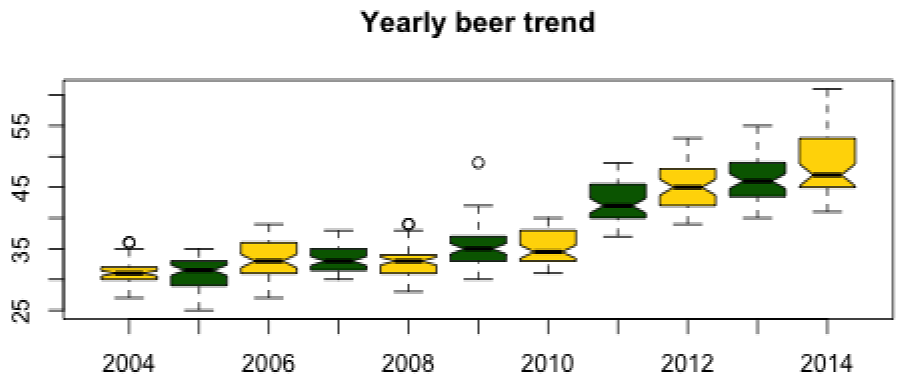
```
plot(newData$StartDate, newData$beer, type='l', col='Blue', main="US Wine vs Beer on Google Trends", xlab=
"year", ylab="beverage")

lines(newData$StartDate, newData$wine, type='l', col='Red')
```

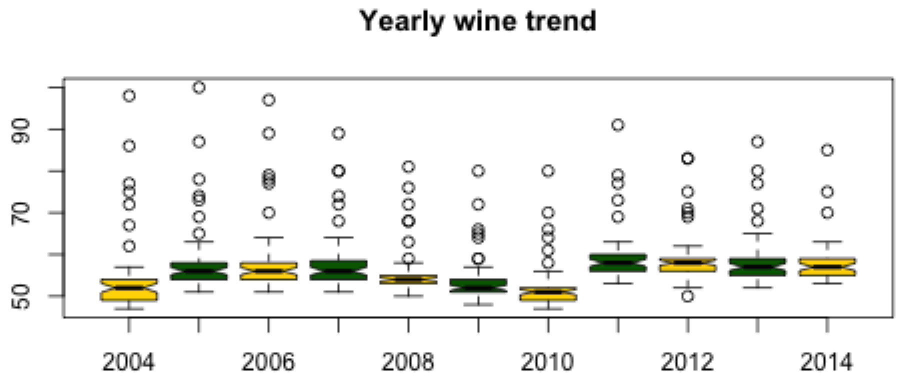


At first glance, `beer` seems to be trending upwards. Let's plot it by year in a `boxplot` to better view the differences:

```
par(mfrow = c(2, 1))

# show box plots to account for seasonal outliers and stagnant trend

boxplot(beer~year, data=newData, notch=TRUE,

        col=(c("gold","darkgreen")),

        main="Yearly beer trend")
```

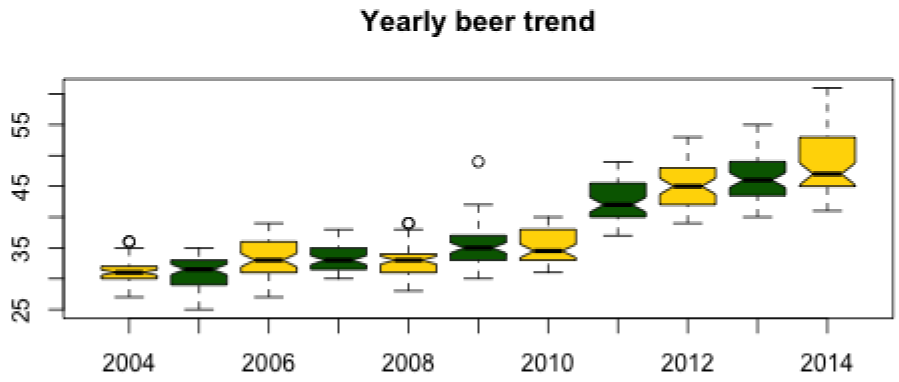
```
boxplot(wine~year, data=newData, notch=TRUE,

        col=(c("gold","darkgreen")),

        main="Yearly wine trend")
```

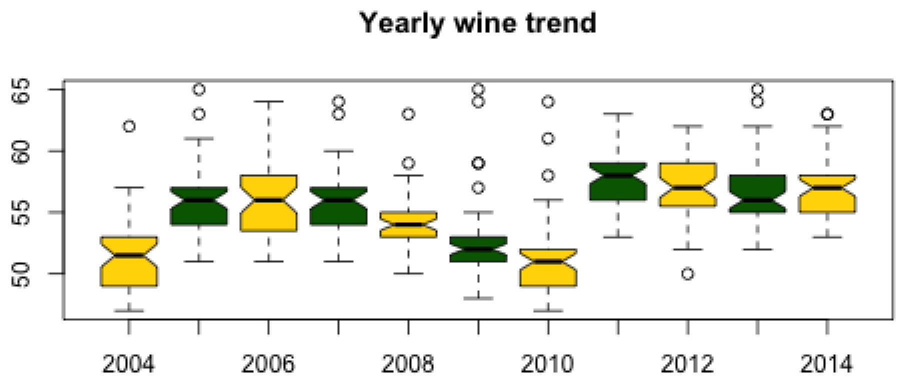




Outliers seem to be plaguing `wine` - this may account for the December spikes. Let's remove them and try plotting again:

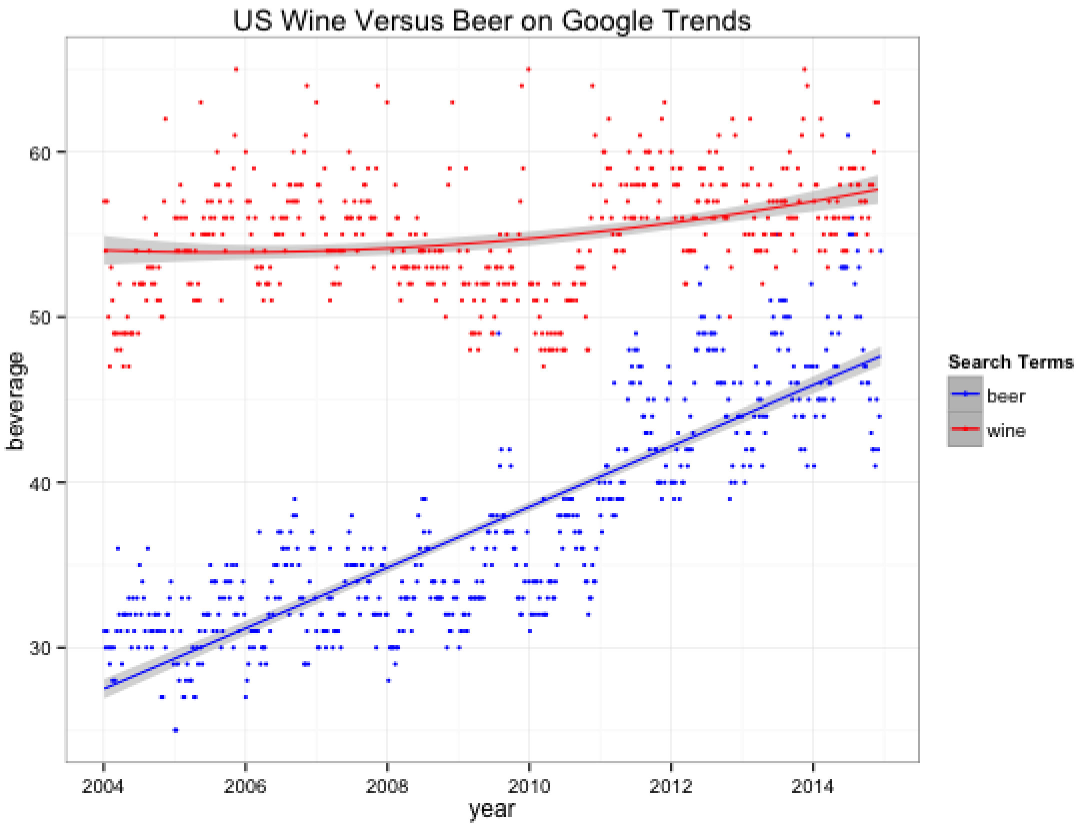
```
# shamelessly borrowed from aL3xa -

# http://stackoverflow.com/questions/4787332/how-to-remove-outliers-from-a-dataset

remove_outliers <- function(x, na.rm = TRUE, ...) {

        qnt <- quantile(x, probs=c(.25, .75), na.rm = na.rm, ...)

        H <- 1.5 * IQR(x, na.rm = na.rm)

        y <- x

        y[x < (qnt[1] - H)] <- NA
        y[x > (qnt[2] + H)] <- NA
        y
}

par(mfrow = c(2, 1))
# show box plots to account for seasonal outliers and stagnant trend
boxplot(beer~year, data=newData, notch=TRUE,
        col=(c("gold","darkgreen")),
        main="Yearly beer trend")

newData$wine_clean <- remove_outliers(newData$wine)
boxplot(wine_clean~year, data=newData, notch=TRUE,
        col=(c("gold","darkgreen")),
        main="Yearly wine trend")
```





Clearly, the term `beer` is trending upwards while `wine` is more erratic and range bound. {ggplot2} has great plotting functions to smooth out noise and amplify trends for better understanding:

```
library(ggplot2)
ggplot(newData,aes(x=StartDate)) +
        stat_smooth(aes(y = beer, group=1, colour="beer"), method=lm, formula = y ~ poly(x,1), level=0.95)
 +
        stat_smooth(aes(y = wine_clean, group=1, colour="wine"), method=lm, formula = y ~ poly(x,2), level
=0.95) +
        geom_point (aes(y = beer, colour = "beer"), size=1) +
        geom_point (aes(y = wine_clean, colour ="wine"), size=1) +
        scale_colour_manual("Search Terms", breaks = c("beer", "wine"), values = c("blue","red")) +
        theme_bw() +
        xlab("year") +
        ylab("beverage") +

        ggtitle("US Wine Versus Beer on Google Trends")
```



Conclusion

There isn't an easy way of concluding this - possibilities are endless. Analyze away! For more information on this tool, read this great article by Hyunyoung Choi and Hal Varian: Predicting the Present with Google Trends.

Full source code (also on GitHub):

```
setwd('//yourpath')


filename <- "beervswine.csv"

con  <- file(filename, open = "r")


linecount <- 0
```

```
stringdata <- ""
while (length(oneLine <- readLines(con, n = 1, warn = FALSE)) > 0) {
        linecount <- linecount + 1

        if (linecount < 3) {
                filename <- paste0(filename,oneLine)
        }

        # get headers at line 5
        if (linecount == 5) rowheaders = strsplit(oneLine, ",")[[1]]

        # skip firt 5 lines
        if (linecount > 5) {
                # break when there is no more main data
                if (gsub(pattern=",", x=oneLine, replacement="") == "") break

                stringdata <- paste0(stringdata,oneLine,"\n")
        }
}
close(con)

newData <- read.table(textConnection(stringdata), sep=",", header=FALSE, stringsAsFactors = FALSE)
names(newData) <- rowheaders

head(newData)

newData$StartDate <- as.Date(sapply(strsplit(as.character(newData[,1]), " - "), `[`, 1))
newData$EndDate <- as.Date(sapply(strsplit(as.character(newData[,1]), " - "), `[`, 2))
newData$year <- sapply(strsplit(as.character(newData$StartDate), "-"), `[`, 1)
newData<- newData[c("StartDate", "EndDate", "beer", "wine", "year")]

head(newData)

plot(newData$StartDate, newData$beer, type='l', col='Blue')
lines(newData$StartDate, newData$wine, type='l', col='Red')

par(mfrow = c(2, 1))
# show box plots to account for seasonal outliers and stagnant trend
```

```
boxplot(beer~year, data=newData, notch=TRUE,
        col=(c("gold","darkgreen")),
        main="Yearly beer trend")

boxplot(wine~year, data=newData, notch=TRUE,
        col=(c("gold","darkgreen")),
        main="Yearly wine trend")

# shamelessly borrowed from aL3xa -
# http://stackoverflow.com/questions/4787332/how-to-remove-outliers-from-a-dataset
remove_outliers <- function(x, na.rm = TRUE, ...) {
        qnt <- quantile(x, probs=c(.25, .75), na.rm = na.rm, ...)
        H <- 1.5 * IQR(x, na.rm = na.rm)
        y <- x
        y[x < (qnt[1] - H)] <- NA
        y[x > (qnt[2] + H)] <- NA
        y
}

par(mfrow = c(2, 1))
# show box plots to account for seasonal outliers and stagnant trend
boxplot(beer~year, data=newData, notch=TRUE,
        col=(c("gold","darkgreen")),
        main="Yearly beer trend")

newData$wine_clean <- remove_outliers(newData$wine)
boxplot(wine_clean~year, data=newData, notch=TRUE,
        col=(c("gold","darkgreen")),
        main="Yearly wine trend")


# smooth both trends and plot them together
par(mfrow = c(1, 1))
plot(newData$StartDate, newData$beer, type='l', col='Blue')
lines(newData$StartDate, newData$wine_clean, type='l', col='Red')


library(ggplot2)
```

```
ggplot(newData,aes(x=StartDate)) +
        stat_smooth(aes(y = beer, group=1, colour="beer"), method=lm, formula = y ~ poly(x,1), level=0.95)
 +
        stat_smooth(aes(y = wine_clean, group=1, colour="wine"), method=lm, formula = y ~ poly(x,2), level
=0.95) +
        geom_point (aes(y = beer, colour = "beer"), size=1) +
        geom_point (aes(y = wine_clean, colour ="wine"), size=1) +
        scale_colour_manual("Search Terms", breaks = c("beer", "wine"), values = c("blue","red")) +
        theme_bw() +
        xlab("year") +
        ylab("beverage") +
        ggtitle("US Wine Versus Beer on Google Trends")
```

How To Work with Files Too Large for a Computer’s RAM? Using R to Process Large Data in Chunks

November 4, 2014 Tags: *exploring*

Resources

- YouTube Companion Video
- Full Source Code

There are times when files are just too large to fit in a computer’s live memory.

If you’re brand-new to **R** you may not have encountered this yet but we all do eventually. The problem happens when calling functions such as `read.csv()` or `` `read.table() `` on large data files and your computer ends up freezing or choking. I usually end up losing patience and killing the process.

In **R** you cannot open a 20 GB file on a computer with 8 GBs of RAM - it just won't work. By default, **R** will load all the data into RAM. Even files that are smaller than your RAM may not be opened depending on what you have running, on your OS, 32/64 bit, etc.

To get an estimate on how much memory a data frame needs remember that an integer uses 4 bytes and a float about 8. So if you have 100 columns and 100,000 rows, you would multiply **8 by 100 by 100000** (NOTE: 2^{20} will convert bytes to megabytes):

```
options(scipen=999) # block scientific notation

print(paste((8*100*100000) / 2^20, 'megabytes'))

## [1] "76.2939453125 megabytes"
```

76 megabytes isn’t a problem for most computers, but what to do when it is? There are various ways of dealing with such issues such as using command line tools to break the files into smaller ones or rent a larger computer from a cloud service. But an easy **R** solution is to iteratively read the data in smaller-sized chunks that your computer can handle.

Let's download a large **CSV** file from the University of California, Irvine's Machine Learning Repository. Download the compressed HIGGS Data Set and unzip it (NOTE: this is a huge file that unzips at over 8 GB):

```
setwd('Enter Your Folder Path Here...')

download.file('http://archive.ics.uci.edu/ml/machine-learning-databases/00280/HIGGS.csv.gz', '
HIGGS.csv.gz')
```

If the above code doesn't work, you can download it directly here.

Once you unzipped it, we can run the `file.info` to get some details about it without loading it in memory (NOTE: 2^{30} will convert bytes to gigabytes):

```
print(paste(file.info('HIGGS.csv')$size  / 2^30, 'gigabytes'))

## [1] "7.48364066705108 gigabytes"
```

This is a big one, coming in at around **7.5 GB**, a lot of machines won't be able to read it directly into memory with a typical `read.csv()` call.

The `readLines()` function is a workhorse when it comes to peeking into a very large file without loading the whole thing. I often use it to get the column headers and a handful of rows:

```
transactFile <- 'HIGGS.csv'

readLines(transactFile, n=1)

## [1] "1.000000000000000000e+00,8.692932128906250000e-01,-6.350818276405334473e-01,2.25690260
5295181274e-01,3.274700641632080078e-01,-6.899932026863098145e-01,7.542022466659545898e-01,-2.
485731393098831177e-01,-1.092063903808593750e+00,0.000000000000000000e+00,1.374992132186889648
e+00,-6.536741852760314941e-01,9.303491115570068359e-01,1.107436060905456543e+00,1.13890433311
```

```
4624023e+00,-1.578198313713073730e+00,-1.046985387802124023e+00,0.000000000000000000e+00,6.579
295396804809570e-01,-1.045456994324922562e-02,-4.576716944575309753e-02,3.101961374282836914e+
00,1.353760004043579102e+00,9.795631170272827148e-01,9.780761599540710449e-01,9.20004844665527
3438e-01,7.216574549674987793e-01,9.887509346008300781e-01,8.766783475875854492e-01"
```

You could easily use `readLines` to loop through smaller chunks in memory one at a time. But I prefer `read.table()` and that is what I use.

I copied the column names from the UCI repository:

```
higgs_colnames <- c('label','lepton_pT','lepton_eta','lepton_phi','missing_energy_magnitude','
missing_energy_phi','jet_1_pt','jet_1_eta','jet_1_phi','jet_1_b_tag','jet_2_pt','jet_2_eta','j
et_2_phi','jet_2_b_tag','jet_3_pt','jet_3_eta','jet_3_phi','jet_3_b-tag','jet_4_pt','jet_4_eta
','jet_4_phi','jet_4_b_tag','m_jj','m_jjj','m_lv','m_jlv','m_bb','m_wbb','m_wwbb')


transactFile <- 'HIGGS.csv'

chunkSize <- 100000

con <- file(description= transactFile, open="r")

data <- read.table(con, nrows=chunkSize, header=T, fill=TRUE, sep=",")

close(con)

names(data) <- higgs_colnames

print(head(data))

##   label lepton_pT lepton_eta lepton_phi missing_energy_magnitude

## 1     1    0.9075     0.3291   0.359412                   1.4980

## 2     1    0.7988     1.4706  -1.635975                   0.4538

## 3     0    1.3444    -0.8766   0.935913                   1.9921

## 4     1    1.1050     0.3214   1.522401                   0.8828

## 5     0    1.5958    -0.6078   0.007075                   1.8184

## 6     1    0.4094    -1.8847  -1.027292                   1.6725

##   missing_energy_phi jet_1_pt jet_1_eta jet_1_phi jet_1_b_tag jet_2_pt

## 1            -0.3130   1.0955  -0.55752  -1.58823       2.173   0.8126

## 2             0.4256   1.1049   1.28232   1.38166       0.000   0.8517

## 3             0.8825   1.7861  -1.64678  -0.94238       0.000   2.4233

## 4            -1.2053   0.6815  -1.07046  -0.92187       0.000   0.8009

## 5            -0.1119   0.8475  -0.56644   1.58124       2.173   0.7554

## 6            -1.6046   1.3380   0.05543   0.01347       2.173   0.5098
##   jet_2_eta jet_2_phi jet_2_b_tag jet_3_pt jet_3_eta jet_3_phi jet_3_b-tag
## 1   -0.2136    1.2710       2.215   0.5000   -1.2614    0.7322       0.000
## 2    1.5407   -0.8197       2.215   0.9935    0.3561   -0.2088       2.548
## 3   -0.6760    0.7362       2.215   1.2987   -1.4307   -0.3647       0.000
## 4    1.0210    0.9714       2.215   0.5968   -0.3503    0.6312       0.000
## 5    0.6431    1.4264       0.000   0.9217   -1.1904   -1.6156       0.000
## 6   -1.0383    0.7079       0.000   0.7469   -0.3585   -1.6467       0.000
##   jet_4_pt jet_4_eta  jet_4_phi jet_4_b_tag   m_jj  m_jjj   m_lv  m_jlv
## 1   0.3987   -1.1389 -0.0008191       0.000 0.3022 0.8330 0.9857 0.9781
## 2   1.2570    1.1288  0.9004608       0.000 0.9098 1.1083 0.9857 0.9513
## 3   0.7453   -0.6784 -1.3603563       0.000 0.9467 1.0287 0.9987 0.7283
## 4   0.4800   -0.3736  0.1130406       0.000 0.7559 1.3611 0.9866 0.8381
## 5   0.6511   -0.6542 -1.2743449       3.102 0.8238 0.9382 0.9718 0.7892
## 6   0.3671    0.0695  1.3771303       3.102 0.8694 1.2221 1.0006 0.5450
##     m_bb  m_wbb m_wwbb
## 1 0.7797 0.9924 0.7983
## 2 0.8033 0.8659 0.7801
## 3 0.8692 1.0267 0.9579
## 4 1.1333 0.8722 0.8085
## 5 0.4306 0.9614 0.9578
## 6 0.6987 0.9773 0.8288
```

The Loop

The next step is to build the looping mechanism to repeat this for each subsequent chunk and keep track of each chunk:

```
index <- 0
chunkSize <- 100000
con <- file(description=transactFile,open="r")
dataChunk <- read.table(con, nrows=chunkSize, header=T, fill=TRUE, sep=",")

repeat {
        index <- index + 1
        print(paste('Processing rows:', index * chunkSize))


        if (nrow(dataChunk) != chunkSize){
                print('Processed all files!')
                break}


        dataChunk <- read.table(con, nrows=chunkSize, skip=0, header=FALSE, fill = TRUE, sep="
,")
        print(head(dataChunk))
        break
}
close(con)
## [1] "Processing rows: 100000"
##   V1     V2      V3      V4     V5       V6     V7      V8      V9   V10
## 1  1 0.7238 -0.9146  0.9109 1.1948 -0.44829 0.8395 -0.8714  0.5878 0.000
## 2  0 0.2822 -0.4130  0.1064 0.5119 -1.33140 1.1591 -1.0576  1.5801 1.087
## 3  0 1.6288  0.9291  0.2052 2.0936  0.07923 1.4937 -0.3219 -1.6858 2.173
## 4  1 1.9741  0.6603 -1.3624 1.2341  1.67772 1.4788  0.4089 -0.1053 0.000
## 5  0 0.4209 -0.4529  0.3383 0.4856 -0.51379 0.5136 -0.5189 -0.2322 2.173
## 6  1 0.9469  0.1694  1.2100 0.3433 -1.57955 0.9994  1.0308 -0.4750 0.000
##      V11       V12       V13   V14    V15       V16     V17   V18    V19
## 1 0.6544  1.15988 -0.72592 0.000 0.4220  1.63680 -0.8806 0.000 1.0340
## 2 1.0966 -1.61631 -0.34808 0.000 1.0557 -1.45258  0.2002 0.000 1.0082
## 3 1.0244  0.61105  1.57284 0.000 1.7970  0.61550 -0.4662 1.274 0.6991
## 4 1.0170 -0.12719  0.36331 2.215 0.9187  0.07208  1.1626 0.000 0.9615
## 5 0.8919  0.01852  1.65662 2.215 0.5904 -0.78992 -1.4586 0.000 0.6996
## 6 0.4354  0.05446 -0.08398 0.000 1.4650  0.61368  1.4927 2.548 1.1927
##       V20     V21   V22    V23    V24    V25    V26    V27    V28    V29
## 1 -0.7042 -0.9170 3.102 0.8671 1.1272 1.2117 0.6959 0.6941 0.7558 0.7617
## 2 -1.0215  1.0814 3.102 0.9061 0.7504 0.9942 1.6251 0.5069 1.1208 1.2031
## 3 -0.7059  1.3311 0.000 0.9216 0.9004 0.9633 0.9176 2.1077 1.5558 1.3126
## 4  0.6292  1.6041 3.102 1.9387 1.2339 0.9901 0.5249 0.9006 0.9176 1.0834
## 5 -1.8235  0.7967 0.000 0.8101 0.9102 0.9831 0.7197 1.0245 0.8279 0.7211
## 6  0.1903  0.5586 3.102 0.8816 0.8454 0.9974 0.6951 0.7871 0.6577 0.7211
```

If you need the column names, then you will have to reapply them after each loop. This is easy to do as the `read.table` function has a parameter just for that:

```
dataChunk <- read.table(con, nrows=chunkSize, skip=0, header=FALSE, fill = TRUE, col.names=higgs_colnames)
```

Getting Something Out Of Each Loop

Now that you understand this chunking mechanism, let’s see if we can get a total mean for a row from multiple chunks.

```
index <- 0
chunkSize <- 100000
con <- file(description=transactFile,open="r")
dataChunk <- read.table(con, nrows=chunkSize, header=T, fill=TRUE, sep=",", col.names=higgs_colnames)

counter <- 0
total_lepton_pT <- 0
repeat {
        index <- index + 1
        print(paste('Processing rows:', index * chunkSize))

        total_lepton_pT <- total_lepton_pT + sum(dataChunk$lepton_pT)
        counter <- counter + nrow(dataChunk)

        if (nrow(dataChunk) != chunkSize){
                print('Processed all files!')
                break}

        dataChunk <- read.table(con, nrows=chunkSize, skip=0, header=FALSE, fill = TRUE, sep=",", col.names=higgs_colnames)
        if (index > 3) break


}

close(con)

## [1] "Processing rows: 100000"

## [1] "Processing rows: 200000"

## [1] "Processing rows: 300000"

## [1] "Processing rows: 400000"

print(paste0('lepton_pT mean: ',  total_lepton_pT / counter))

## [1] "lepton_pT mean: 0.992386268476397"
```

We broke out of the loop a little early but you get the point. This type of approach may not work for a real median, unless your live memory can hold the entire column of data at the very least. But anything that can be worked in chunks, like the above mean, can easily be extended into parallel or distributed systems.

I hope this helps.

Full source code (also on GitHub):

```
options(scipen=999) # block scientific notation

print(paste((8*100*100000) / 2^20, 'megabytes'))


setwd('Enter Your Folder Path Here...')

download.file('http://archive.ics.uci.edu/ml/machine-learning-databases/00280/HIGGS.csv.gz', 'HIGGS.csv.gz')


print(paste(file.info('HIGGS.csv')$size  / 2^30, 'gigabytes'))


transactFile <- 'HIGGS.csv'

readLines(transactFile, n=1)


higgs_colnames <- c('label','lepton_pT','lepton_eta','lepton_phi','missing_energy_magnitude','missing_energy_phi','jet_1_pt','jet_1_eta','jet_1_phi','jet_1_b_tag','jet_2_pt','jet_2_eta','jet_2_phi','jet_2_b_tag','jet_3_pt','jet_3_eta','jet_3_phi','jet_3_b-tag','jet_4_pt','jet_4_eta','jet_4_phi','jet_4_b_tag','m_jj','m_jjj','m_lv','m_jlv','m_bb','m_wbb','m_wwbb')
```

```
transactFile <- 'HIGGS.csv'
chunkSize <- 100000
con <- file(description= transactFile, open="r")
data <- read.table(con, nrows=chunkSize, header=T, fill=TRUE, sep=",")
close(con)
names(data) <- higgs_colnames
print(head(data))

index <- 0
chunkSize <- 100000
con <- file(description=transactFile,open="r")
dataChunk <- read.table(con, nrows=chunkSize, header=T, fill=TRUE, sep=",")

repeat {
        index <- index + 1
        print(paste('Processing rows:', index * chunkSize))

        if (nrow(dataChunk) != chunkSize){
                print('Processed all files!')
                break}

        dataChunk <- read.table(con, nrows=chunkSize, skip=0, header=FALSE, fill = TRUE, sep=",")
        print(head(dataChunk))
        break
}
close(con)

dataChunk <- read.table(con, nrows=chunkSize, skip=0, header=FALSE, fill = TRUE, col.names=higgs_colnames)

index <- 0
chunkSize <- 100000
con <- file(description=transactFile,open="r")
dataChunk <- read.table(con, nrows=chunkSize, header=T, fill=TRUE, sep=",", col.names=higgs_colnames)

counter <- 0
total_lepton_pT <- 0
repeat {
        index <- index + 1
        print(paste('Processing rows:', index * chunkSize))


        total_lepton_pT <- total_lepton_pT + sum(dataChunk$lepton_pT)

        counter <- counter + nrow(dataChunk)


        if (nrow(dataChunk) != chunkSize){

                print('Processed all files!')

                break}


        dataChunk <- read.table(con, nrows=chunkSize, skip=0, header=FALSE, fill = TRUE, sep=",", col.names=higgs_colnames)


        if (index > 3) break


}

close(con)

print(paste0('lepton_pT mean: ',  total_lepton_pT / counter))
```

Brief Walkthrough Of The dummyVars Function From {caret}

October 2, 2014 Tags: *exploring*

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{caret}** - dummyVars function

As the name implies, the `dummyVars` function allows you to create dummy variables - in other words it translates text data into numerical data for modeling purposes.

If you are planning on doing predictive analytics or machine learning and want to use regression or any other modeling technique that requires numerical data, you will need to transform your text data into numbers otherwise you run the risk of leaving a lot of information on the table...

In R, there are plenty of ways of translating text into numerical data. You can do it manually, use a base function, such as **matrix**, or a packaged function like dummyVarfrom the caret package. One of the big advantages of going with the caret package is that it's full of features, including hundreds of algorithms and pre-processing functions. Once your data fits into caret's modular design, it can be run through different models with minimal tweaking.

Let's look at a few examples of dummy variables. If you have a survey question with 5 categorical values such as very unhappy, unhappy, neutral, happy and very happy.

```
survey <- data.frame(service=c('very unhappy','unhappy','neutral','happy','very happy'))

print(survey)

##        service

## 1 very unhappy

## 2      unhappy
## 3      neutral

## 4        happy

## 5   very happy
```

You can easily translate this into a sequence of numbers from 1 to 5. Where 3 means neutral and, in the example of a linear model that thinks in fractions, 2.5 means somewhat unhappy, and 4.88 means very happy. So here we successfully transformed this survey question into a continuous numerical scale and do not need to add dummy variables - a simple rank column will do.

```
survey <- data.frame(service=c('very unhappy','unhappy','neutral','happy','very happy'), rank=c(1,2,3,4,5)
)

print(survey)

##        service rank

## 1 very unhappy    1

## 2      unhappy    2

## 3      neutral    3

## 4        happy    4

## 5   very happy    5
```

So, the above could easily be used in a model that needs numbers and still represent that data accurately using the 'rank' variable instead of 'service'. **But** this only works in specific situations where you have somewhat linear and continuous-like data. What happens with categorical values such as marital status, gender, alive?

Does it make sense to be a quarter female? Or half single? Even numerical data of a categorical nature may require transformation. Take the zip code system. Does the half-way point between two zip codes make geographical sense? Because that is how a regression model would use it.
<BR It may work in a fuzzy-logic way but it won't help in predicting much; therefore we need a more precise way of translating these values into numbers so that they can be regressed by the model.

```
library(caret)

# check the help file for more details

?dummyVars
```

The **dummyVars** function breaks out unique values from a column into individual columns - if you have 1000 unique values in a column, dummying them will add 1000 new columns to your data set (be careful). Let’s create a more complex data frame:

```
customers <- data.frame(
                id=c(10,20,30,40,50),
                gender=c('male','female','female','male','female'),
                mood=c('happy','sad','happy','sad','happy'),
                outcome=c(1,1,0,0,0))
```

And ask the `dummyVars` function to dummify it. The function takes a standard R formula: **something ~ (broken down) by something else or groups of other things**. So we simply use `~ .` and the `dummyVars` will transform all characters and factors columns (the function never transforms numeric columns) and return the entire data set:

```
# dummify the data
dmy <- dummyVars(" ~ .", data = customers)
trsf <- data.frame(predict(dmy, newdata = customers))
print(trsf)
##   id gender.female gender.male mood.happy mood.sad outcome
## 1 10             0           1          1        0       1
## 2 20             1           0          0        1       1
## 3 30             1           0          1        0       0
## 4 40             0           1          0        1       0
## 5 50             1           0          1        0       0
```

If you just want one column transform you need to include that column in the formula and it will return a data frame based on that variable only:

```
customers <- data.frame(
                id=c(10,20,30,40,50),
                gender=c('male','female','female','male','female'),
                mood=c('happy','sad','happy','sad','happy'),
                outcome=c(1,1,0,0,0))

dmy <- dummyVars(" ~ gender", data = customers)
trsf <- data.frame(predict(dmy, newdata = customers))
print(trsf)
##   gender.female gender.male
## 1             0           1
## 2             1           0
## 3             1           0
## 4             0           1
## 5             1           0
```

The `fullRank` parameter is worth mentioning here. The general rule for creating dummy variables is to have one less variable than the number of categories present to avoid perfect collinearity (**dummy variable trap**). You basically want to avoid highly correlated variables but it also save space. If you have a factor column comprised of two levels 'male' and 'female', then you don't need to transform it into two columns, instead, you pick one of the variables and you are either female, if it’s a **1**, or male if it’s a **0**.
Let's turn on `fullRank` and try our data frame again:

```
customers <- data.frame(
                id=c(10,20,30,40,50),
                gender=c('male','female','female','male','female'),
                mood=c('happy','sad','happy','sad','happy'),
                outcome=c(1,1,0,0,0))

dmy <- dummyVars(" ~ .", data = customers, fullRank=T)
trsf <- data.frame(predict(dmy, newdata = customers))
print(trsf)
##   id gender.male mood.sad outcome
## 1 10           1        0       1
## 2 20           0        1       1
## 3 30           0        0       0
## 4 40           1        1       0
## 5 50           0        0       0
```

As you can see, it picked male and sad, if you are **0** in both columns, then you are `female` and `happy`.

Things to keep in mind

• Don’t dummy a large data set full of zip codes; you more than likely don't have the computing muscle to add an extra 43,000 columns to your data set.

• You can dummify large, free-text columns. Before running the function, look for repeated words or sentences, only take the top 50 of them and replace the rest with 'others'. This will allow you to use that field without delving deeply into NLP.

Full source code (also on GitHub):

```
survey <- data.frame(service=c('very unhappy','unhappy','neutral','happy','very happy'))
print(survey)


survey <- data.frame(service=c('very unhappy','unhappy','neutral','happy','very happy'), rank=c(1,2,3,4,5)
)
print(survey)


library(caret)


?dummyVars # many options


customers <- data.frame(
        id=c(10,20,30,40,50),
        gender=c('male','female','female','male','female'),
        mood=c('happy','sad','happy','sad','happy'),
        outcome=c(1,1,0,0,0))


dmy <- dummyVars(" ~ .", data = customers)
trsf <- data.frame(predict(dmy, newdata = customers))
print(trsf)
print(str(trsf))


# works only on factors
```

```
customers$outcome <- as.factor(customers$outcome)

# tranform just gender
dmy <- dummyVars(" ~ gender", data = customers)
trsf <- data.frame(predict(dmy, newdata = customers))
print(trsf)

# use fullRank to avoid the 'dummy trap'
dmy <- dummyVars(" ~ .", data = customers, fullRank=T)
trsf <- data.frame(predict(dmy, newdata = customers))
print(trsf)
```

Using Correlations to Understand Your Data

September 27, 2014 Tags: *exploring*, *visualizing*

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{RCurl}** - downloads https data
- **{psych}** - correlation matrix plotting
- **{caret}** - dummyVars function

A great way to explore new data is to use a pairwise correlation matrix. This will measure the correlation between every combination of your variables. It doesn't really matter if you have an outcome (or response) variable at this point, it will compare everything against everything else.

For those not familiar with the correlation coefficient, it is simply a measure of similarity between two vectors of numbers. The measure value can range between **1** and **-1**, where **1** is perfectly correlated, **-1** is perfectly inversly correlated, and **0** is not correlated at all:

```
print(cor(1:5,1:5))

## 1

print(cor(1:5,5:1))

## -1

print(cor(1:5,c(1,2,3,4,4)))

## 0.9701425
```

To help us understand this process, let's download the adult.data set from the UCI Machine Learning Repository. The data is from the 1994 Census and attempts to predict those with income under $50,000 a year:

```
library(RCurl) # download https data
urlfile <- 'https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data'
x <- getURL(urlfile, ssl.verifypeer = FALSE)
adults <- read.csv(textConnection(x), header=F)

# if the above getURL command fails, try this:
# adults <-read.csv('https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data', header=F)
```

We fill in the missing headers for the UCI set and cast the outcome variable `Income` to a binary format of **1** and **0** (here I am reversing the orginal order, if it is under $50k, it is **0**, and above, **1**; this will make the final correlation-matrix plot easier to understand):

```
names(adults)=c('Age','Workclass','FinalWeight','Education','EducationNumber',
                'MaritalStatus','Occupation','Relationship','Race',
                'Sex','CapitalGain','CapitalLoss','HoursWeek',
                'NativeCountry','Income')

adults$Income <- ifelse(adults$Income==' <=50K',0,1)
```

We load the **caret** package to dummify (see my other walkthrough) all factor variables as the `cor` function only accepts numerical values:

```
dmy <- dummyVars(" ~ .", data = adults)
```

```
adultsTrsf <- data.frame(predict(dmy, newdata = adults))
```

We borrow two very useful functions from Stephen Turner: `cor.prob` and `flattenSquareMatrix`. **cor.prob** will create a correlation matrix along with *p*-values and **flattenSquareMatrix** will flatten all the combinations from the square matrix into a data frame of 4 columns made up of row names, column names, the correlation value and the *p*-value:

```
corMasterList <- flattenSquareMatrix (cor.prob(adultsTrsf))
print(head(corMasterList,10))
##                         i                       j       cor         p
## 1                     age            workclass...  0.042627 1.421e-14
## 2                     age  workclass..Federal.gov  0.051227 0.000e+00
## 3            workclass...  workclass..Federal.gov -0.042606 1.454e-14
## 4                     age    workclass..Local.gov  0.060901 0.000e+00
## 5            workclass...    workclass..Local.gov -0.064070 0.000e+00
## 6  workclass..Federal.gov    workclass..Local.gov -0.045682 2.220e-16
## 7                     age workclass..Never.worked -0.019362 4.759e-04
## 8            workclass... workclass..Never.worked -0.003585 5.178e-01
## 9  workclass..Federal.gov workclass..Never.worked -0.002556 6.447e-01
## 10   workclass..Local.gov workclass..Never.worked -0.003843 4.880e-01
```

This final format allows you to easily order the pairs however you want - for example, by those with the highest absolute correlation value:

```
corList <- corMasterList[order(-abs(corMasterList$cor)),]
print(head(corList,10))
##                                      i                            j        cor p
## 1953                       sex..Female                    sex..Male -1.0000000 0
## 597                       workclass...                occupation...  0.9979854 0
## 1256 maritalStatus..Married.civ.spouse        relationship..Husband  0.8932103 0
## 1829                       race..Black                  race..White -0.7887475 0
## 527  maritalStatus..Married.civ.spouse maritalStatus..Never.married -0.6448661 0
## 1881             relationship..Husband                  sex..Female -0.5801353 0
## 1942             relationship..Husband                    sex..Male  0.5801353 0
## 1258      maritalStatus..Never.married        relationship..Husband -0.5767295 0
## 1306 maritalStatus..Married.civ.spouse  relationship..Not.in.family -0.5375883 0
## 497                                age maritalStatus..Never.married -0.5343590 0
```

The top correlated pair (sex..Female and sex.Male), as seen above, won't be of much use as they are the only two levels of the same factor. We need to process this a little further to be of practical use. We create a single vector of variable names by filtering those with an absolute correlation of 0.2 or higher against the outcome variable of `Income`:

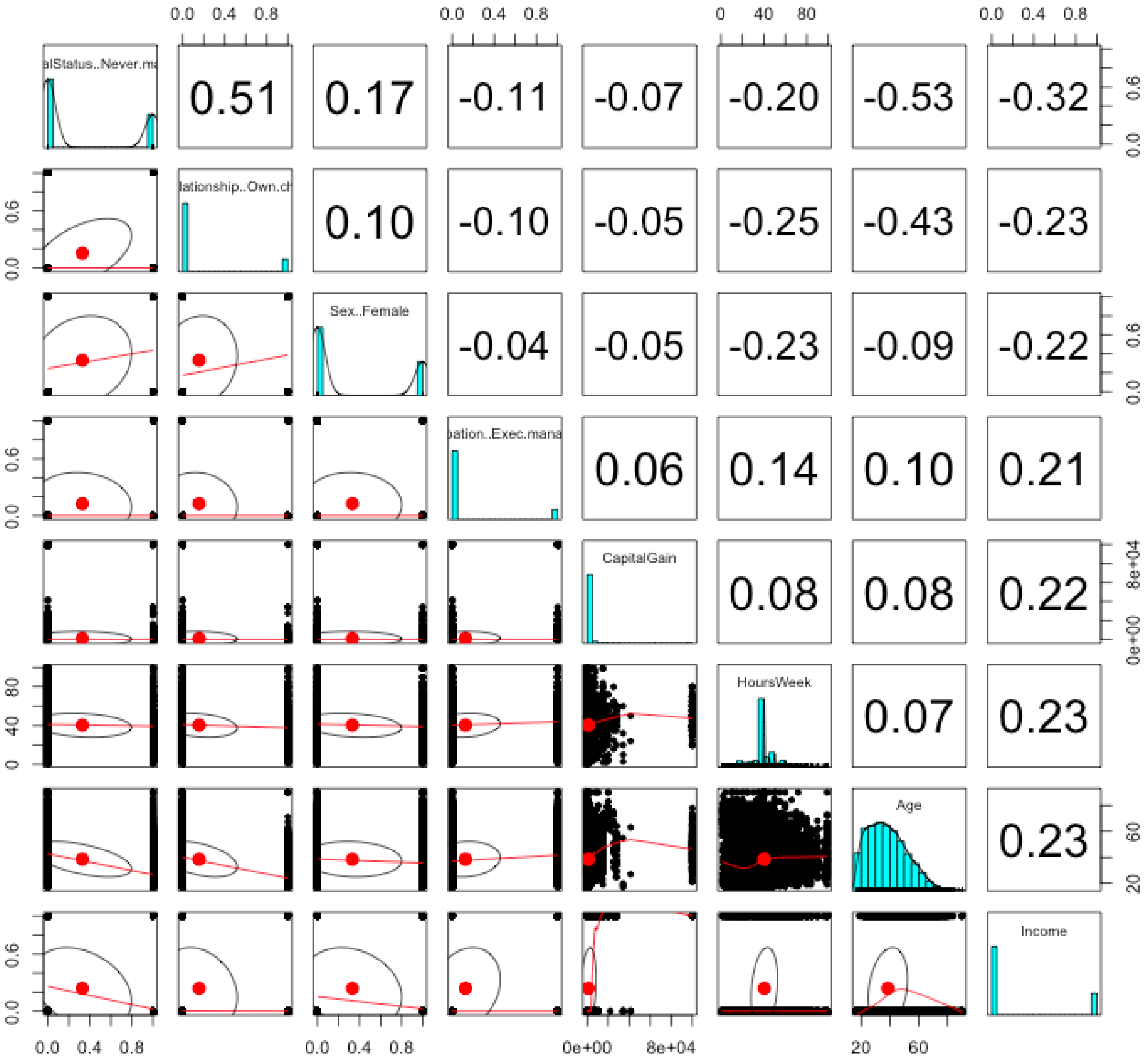
```
selectedSub <- subset(corList, (abs(cor) > 0.2 & j == 'Income'))
print(selectedSub)
                                     i      j        cor p
## 5811      MaritalStatus..Never.married Income -0.3184403 0
## 5832           Relationship..Own.child Income -0.2285320 0
## 5840                       Sex..Female Income -0.2159802 0
## 5818       Occupation..Exec.managerial Income  0.2148613 0
## 5841                         Sex..Male Income  0.2159802 0
## 5842                       CapitalGain Income  0.2233288 0
## 5844                         HoursWeek Income  0.2296891 0
## 5779                               Age Income  0.2340371 0
## 5806                  Education.Number Income  0.3351540 0
## 5829             Relationship..Husband Income  0.4010353 0
## 5809 MaritalStatus..Married.civ.spouse Income  0.4446962 0
```

We save the most correlated features to the `bestSub` variable:

```
bestSub <- as.character(selectedSub$i)
```

Finally we plot the highly correlated pairs using the **{psych}** package's `pair.panels` plot (this can also be done on the original data as `pair.panels` can handle factor and character variables):

```
library(psych)

pairs.panels(adultsTrsf[c(bestSub, 'Income')])
```



The pairs plot, and in particular the last `Income` column, tell us a lot about our data set. Being `never married` is the most negatively correlated with income over $50,000/year and `Hours Worked` and `Age` are the most postively correlated.

Things to keep in mind

- If you have a huge number of features in your data set, then be ready for extra computing time,
- and don't bother plotting it via `pair.panels`, it will end up hanging R.

Full source code (also on GitHub):

```
library(RCurl) # download https data
urlfile <- 'https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data'
x <- getURL(urlfile, ssl.verifypeer = FALSE)
adults <- read.csv(textConnection(x), header=F)

# if the above getURL command fails, try this:
# adults <-read.csv('https://archive.ics.uci.edu/ml/machine-learning-databases/adult/adult.data', header=F)

names(adults)=c('Age','Workclass','FinalWeight','Education','EducationNumber',
                'MaritalStatus','Occupation','Relationship','Race',
                'Sex','CapitalGain','CapitalLoss','HoursWeek',
                'NativeCountry','Income')

adults$Income <- ifelse(adults$Income==' <=50K',0,1)

library(caret)
dmy <- dummyVars(" ~ .", data = adults)
```

```
adultsTrsf <- data.frame(predict(dmy, newdata = adults))


## Correlation matrix with p-values. See http://goo.gl/nahmV for documentation of this function
cor.prob <- function (X, dfr = nrow(X) - 2) {
        R <- cor(X, use="pairwise.complete.obs")
        above <- row(R) < col(R)
        r2 <- R[above]^2
        Fstat <- r2 * dfr/(1 - r2)
        R[above] <- 1 - pf(Fstat, 1, dfr)
        R[row(R) == col(R)] <- NA
        R
}


## Use this to dump the cor.prob output to a 4 column matrix
## with row/column indices, correlation, and p-value.
## See StackOverflow question: http://goo.gl/fCUcQ
flattenSquareMatrix <- function(m) {
        if( (class(m) != "matrix") | (nrow(m) != ncol(m))) stop("Must be a square matrix.")
        if(!identical(rownames(m), colnames(m))) stop("Row and column names must be equal.")
        ut <- upper.tri(m)
        data.frame(i = rownames(m)[row(m)[ut]],
                   j = rownames(m)[col(m)[ut]],
                   cor=t(m)[ut],
                   p=m[ut])
}


corMasterList <- flattenSquareMatrix (cor.prob(adultsTrsf))
print(head(corMasterList,10))
```

```
corList <- corMasterList[order(-abs(corMasterList$cor)),]
print(head(corList,10))

corList <- corMasterList[order(corMasterList$cor),]
selectedSub <- subset(corList, (abs(cor) > 0.2 & j == 'Income'))
# to get the best variables from original list:
bestSub <-  sapply(strsplit(as.character(selectedSub$i),'[.]'), "[", 1)
bestSub <- unique(bestSub)

# or use the variables from top selectedSub:
bestSub <- as.character(selectedSub$i)

library(psych)
pairs.panels(adultsTrsf[c(bestSub, 'Income')])
```

SMOTE - Supersampling Rare Events in R

November 13, 2014 Tags: *exploring, modeling*

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{DMwR}** - Functions and data for the book "Data Mining with R" and SMOTE algorithm
- **{caret}** - modeling wrapper, functions, commands
- **{pROC}** - Area Under the Curve (AUC) functions

The `SMOTE` function oversamples your rare event by using bootstrapping and *k*-nearest neighbor to synthetically create additional observations of that event. The definition of **rare event** is usually attributed to any outcome/dependent/target/response variable that happens less than 15% of the time. For more details about this algorithm, read the original white paper, SMOTE, from its creators.

The best way to illustrate this tool is to apply it to an actual data set suffering from this so-called **rare event**. We'll use the **Thyroid Disease** data set from the UCI Machine Learning Repository (**University of California, Irvine**) containing positive and negative cases of hyperthyroidism. This is a condition in which the thyroid gland produces too much thyroid hormone (also known as "overactive thyroid").

Let's Clean Some Data

Let's load the **hypothyroid** data, clean it up by removing the colons and periods, and appending readable column names:

```
hyper <-read.csv('http://archive.ics.uci.edu/ml/machine-learning-databases/thyroid-disease/hypothyroid.data', header=F)

names <- read.csv('http://archive.ics.uci.edu/ml/machine-learning-databases/thyroid-disease/hypothyroid.names', header=F, sep='\t')[[1]]

names <- gsub(pattern =":|[.]",x = names, replacement="")

colnames(hyper) <- names
```

We change the target column name from `hypothyroid, negative.` to `target` and set any values of `negative` to **0** and everything else to **1**:

```
colnames(hyper) <-c("target", "age", "sex", "on_thyroxine", "query_on_thyroxine",
  "on_antithyroid_medication", "thyroid_surgery", "query_hypothyroid",
  "query_hyperthyroid", "pregnant", "sick", "tumor", "lithium",
  "goitre", "TSH_measured", "TSH", "T3_measured", "T3", "TT4_measured",
  "TT4", "T4U_measured", "T4U", "FTI_measured", "FTI", "TBG_measured",
  "TBG")
hyper$target <- ifelse(hyper$target=='negative',0,1)
```

Whether dealing with a **rare event** or not, it is a good idea to check the balance of positive versus negative outcomes:

```
print(table(hyper$target))
##    0    1
## 3012  151
print(prop.table(table(hyper$target)))
##       0       1
## 0.95226 0.04774
```

At **5%**, this is clearly a skewed data set, aka **rare event**.

A quick peek to see where we are:

```
head(hyper,2)
##   target age sex on_thyroxine query_on_thyroxine on_antithyroid_medication
## 1      1  72   M            f                  f                         f
## 2      1  15   F            t                  f                         f
##   thyroid_surgery query_hypothyroid query_hyperthyroid pregnant sick tumor
## 1               f                 f                  f        f    f     f
## 2               f                 f                  f        f    f     f
##   lithium goitre TSH_measured TSH T3_measured   T3 TT4_measured TT4
## 1       f      f            y  30           y 0.60            y  15
## 2       f      f            y 145           y 1.70            y  19
##   T4U_measured  T4U FTI_measured FTI TBG_measured TBG
## 1            y 1.48            y  10            n   ?
## 2            y 1.13            y  17            n   ?
```

The data is riddled with characters. These need to binarize into numbers to facilitate modeling:

```
ind <- sapply(hyper, is.factor)
hyper[ind] <- lapply(hyper[ind], as.character)

hyper[ hyper == "?" ] = NA
hyper[ hyper == "f" ] = 0
hyper[ hyper == "t" ] = 1
hyper[ hyper == "n" ] = 0
hyper[ hyper == "y" ] = 1
hyper[ hyper == "M" ] = 0
hyper[ hyper == "F" ] = 1

hyper[ind] <- lapply(hyper[ind], as.numeric)

repalceNAsWithMean <- function(x) {replace(x, is.na(x), mean(x[!is.na(x)]))}
hyper <- repalceNAsWithMean(hyper)
```

Reference Model

We randomly split the data set into 2 equal portions using the `createDataPartition` function from the **caret** package:

```
library(caret)
set.seed(1234)
splitIndex <- createDataPartition(hyper$target, p = .50,
                                  list = FALSE,
                                  times = 1)
trainSplit <- hyper[ splitIndex,]
testSplit <- hyper[-splitIndex,]

prop.table(table(trainSplit$target))
##       0       1
## 0.95006 0.04994
prop.table(table(testSplit$target))
##       0       1
## 0.95446 0.04554
```

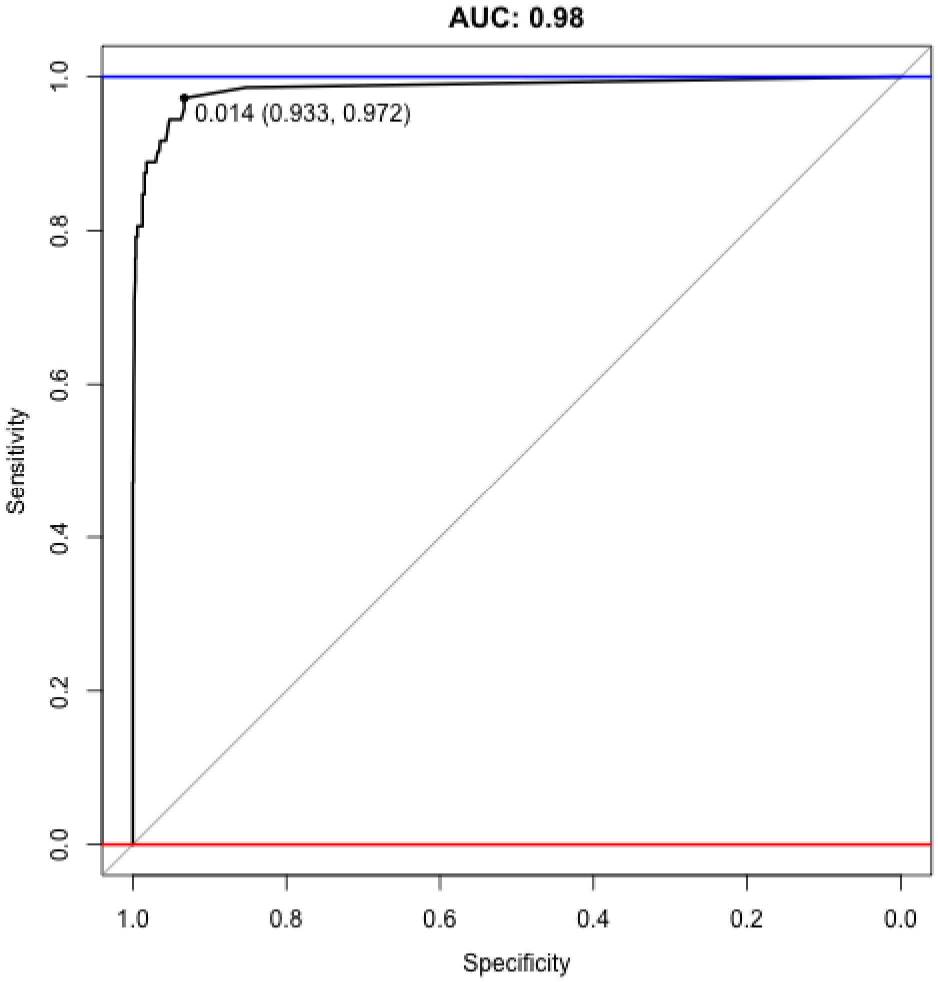
The outcome balance between both splits is still around **5%** therefore representative of the bigger set - we're in good shape.

We train a treebag model using **caret** syntax on `trainSplit` and predict hyperthyroidism on the `testSplit` portion:

```
ctrl <- trainControl(method = "cv", number = 5)
tbmodel <- train(target ~ ., data = trainSplit, method = "treebag",
                 trControl = ctrl)

predictors <- names(trainSplit)[names(trainSplit) != 'target']
pred <- predict(tbmodel$finalModel, testSplit[,predictors])
```

To evaluate the model, we call on package **pROC** for an `auc` score and plot:

```
library(pROC)
auc <- roc(testSplit$target, pred)
print(auc)
## Data: pred in 1509 controls (testSplit$target 0) < 72 cases (testSplit$target 1).
## Area under the curve: 0.985
plot(auc, ylim=c(0,1), print.thres=TRUE, main=paste('AUC:',round(auc$auc[[1]],2)))
abline(h=1,col='blue',lwd=2)
abline(h=0,col='red',lwd=2)
```



An `auc` score of **0.98** is great (remember it ranges on a scale between **0.5** and **1**, where **0.5** is random and **1** is perfect). It is hard to imagine that `SMOTE` can improve on this, but...

Let's SMOTE

Let's create extra positive observations using `SMOTE`. We set `perc.over = 100` to double the quantity of positive cases, and set `perc.under=200` to keep half of what was created as negative cases.

```
library(DMwR)

trainSplit$target <- as.factor(trainSplit$target)

trainSplit <- SMOTE(target ~ ., trainSplit, perc.over = 100, perc.under=200)

trainSplit$target <- as.numeric(trainSplit$target)
```

We can check the `outcome` balance with `prop.table` and confirm that we equalized the data set between positive and negative cases of hyperthyroidism.

```
prop.table(table(trainSplit$target))

##   1   2

## 0.5 0.5
```

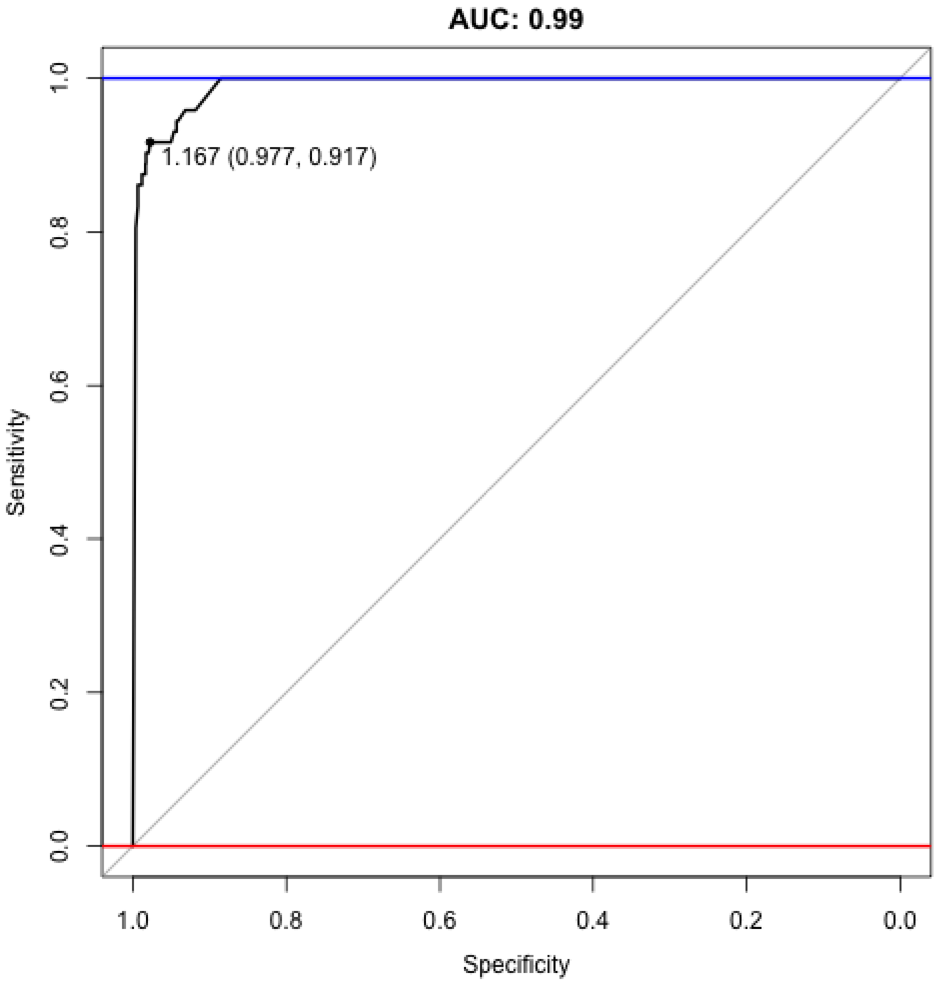
We then train using the `SMOTE`'d training set and predict using the **same** testing set as used before on the non-`SMOTE`'d training set to ensure we're comparing apples-to-apples:

```
tbmodel <- train(target ~ ., data = trainSplit, method = "treebag",

                 trControl = ctrl)


predictors <- names(trainSplit)[names(trainSplit) != 'target']

pred <- predict(tbmodel$finalModel, testSplit[,predictors])


auc <- roc(testSplit$target, pred)

print(auc)

## Data: pred in 1509 controls (testSplit$target 0) < 72 cases (testSplit$target 1).

## Area under the curve: 0.991
```

Wow **0.991**, it managed to better the old `auc` score of **0.985**!

```
plot(auc, ylim=c(0,1), print.thres=TRUE, main=paste('AUC:',round(auc$auc[[1]],2)))
```

```
abline(h=1,col='blue',lwd=2)

abline(h=0,col='red',lwd=2)
```



Conclusion

Not bad - we ended up reducing the overall size and getting a better score. `SMOTE` works great in some situation and not-so-great in others. This definitely requires some trial-and-error but the concept is very promising when stuck with extremely skewed and, therefore, overly sensitive data.

Full source code (also on GitHub):

```
# load data sets
hyper <-read.csv('http://archive.ics.uci.edu/ml/machine-learning-databases/thyroid-disease/hypothyroid.data', header=F)
names <- read.csv('http://archive.ics.uci.edu/ml/machine-learning-databases/thyroid-disease/hypothyroid.names', header=F, sep='\t')[[1]]
names <- gsub(pattern =":|[.]",x = names, replacement="")
colnames(hyper) <- names

# fix variables and column headers
colnames(hyper) <-c("target", "age", "sex", "on_thyroxine", "query_on_thyroxine",
  "on_antithyroid_medication", "thyroid_surgery", "query_hypothyroid",
  "query_hyperthyroid", "pregnant", "sick", "tumor", "lithium",
  "goitre", "TSH_measured", "TSH", "T3_measured", "T3", "TT4_measured",
  "TT4", "T4U_measured", "T4U", "FTI_measured", "FTI", "TBG_measured",
  "TBG")
hyper$target <- ifelse(hyper$target=='negative',0,1)
head(hyper,2)

# check balance of outcome variable
print(table(hyper$target))
print(prop.table(table(hyper$target)))

# binarize all character fields
ind <- sapply(hyper, is.factor)
hyper[ind] <- lapply(hyper[ind], as.character)

hyper[ hyper == "?" ] = NA
hyper[ hyper == "f" ] = 0
hyper[ hyper == "t" ] = 1
hyper[ hyper == "n" ] = 0
hyper[ hyper == "y" ] = 1
hyper[ hyper == "M" ] = 0
hyper[ hyper == "F" ] = 1

hyper[ind] <- lapply(hyper[ind], as.numeric)

repalceNAsWithMean <- function(x) {replace(x, is.na(x), mean(x[!is.na(x)]))}
```

```
hyper <- repalceNAsWithMean(hyper)

# split data into train and test portions
library(caret)
set.seed(1234)
splitIndex <- createDataPartition(hyper$target, p = .50,
                                  list = FALSE,
                                  times = 1)
trainSplit <- hyper[ splitIndex,]
testSplit <- hyper[-splitIndex,]

prop.table(table(trainSplit$target))
prop.table(table(testSplit$target))

# model using treebag
ctrl <- trainControl(method = "cv", number = 5)
tbmodel <- train(target ~ ., data = trainSplit, method = "treebag",
                 trControl = ctrl)

predictors <- names(trainSplit)[names(trainSplit) != 'target']
pred <- predict(tbmodel$finalModel, testSplit[,predictors])

# evaluate the model's performance
library(pROC)
auc <- roc(testSplit$target, pred)
print(auc)
plot(auc, ylim=c(0,1), print.thres=TRUE, main=paste('AUC:',round(auc$auc[[1]],2)))
abline(h=1,col='blue',lwd=2)
abline(h=0,col='red',lwd=2)

# SMOTE more positive cases
library(DMwR)
trainSplit$target <- as.factor(trainSplit$target)
trainSplit <- SMOTE(target ~ ., trainSplit, perc.over = 100, perc.under=200)
trainSplit$target <- as.numeric(trainSplit$target)
```

```
prop.table(table(trainSplit$target))

# evaluate the SMOTE performance
tbmodel <- train(target ~ ., data = trainSplit, method = "treebag",
                 trControl = ctrl)

predictors <- names(trainSplit)[names(trainSplit) != 'target']
pred <- predict(tbmodel$finalModel, testSplit[,predictors])

auc <- roc(testSplit$target, pred)
print(auc)

plot(auc, ylim=c(0,1), print.thres=TRUE, main=paste('AUC:',round(auc$auc[[1]],2)))
abline(h=1,col='blue',lwd=2)
abline(h=0,col='red',lwd=2
```

Let's Get Rich! See how {quantmod} And R Can Enrich Your Knowledge of The Financial Markets!

November 10, 2014 **Tags:** ***exploring, modeling, visualizing***

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{quantmod}** - financial data and modeling tools
- **{xts}** - handling of time-based data classes
- **{xgboost}** - fast modeling algorithm
- **{pROC}** - Area Under the Curve (AUC) functions

This walkthrough has two parts:

1. The first part is a very basic introduction to **quantmod** and, if you haven't used it before and need basic access to daily stock market data and charting, then you're in for a **huge**treat.
2. The second part goes deeper into quantitative finance by leveraging **quantmod** to access all the stocks composing the NASDAQ 100 Index to build a vocabulary of market moves and attempt to predict whether the following trading day's **volumne** will be **higher** or **lower**.

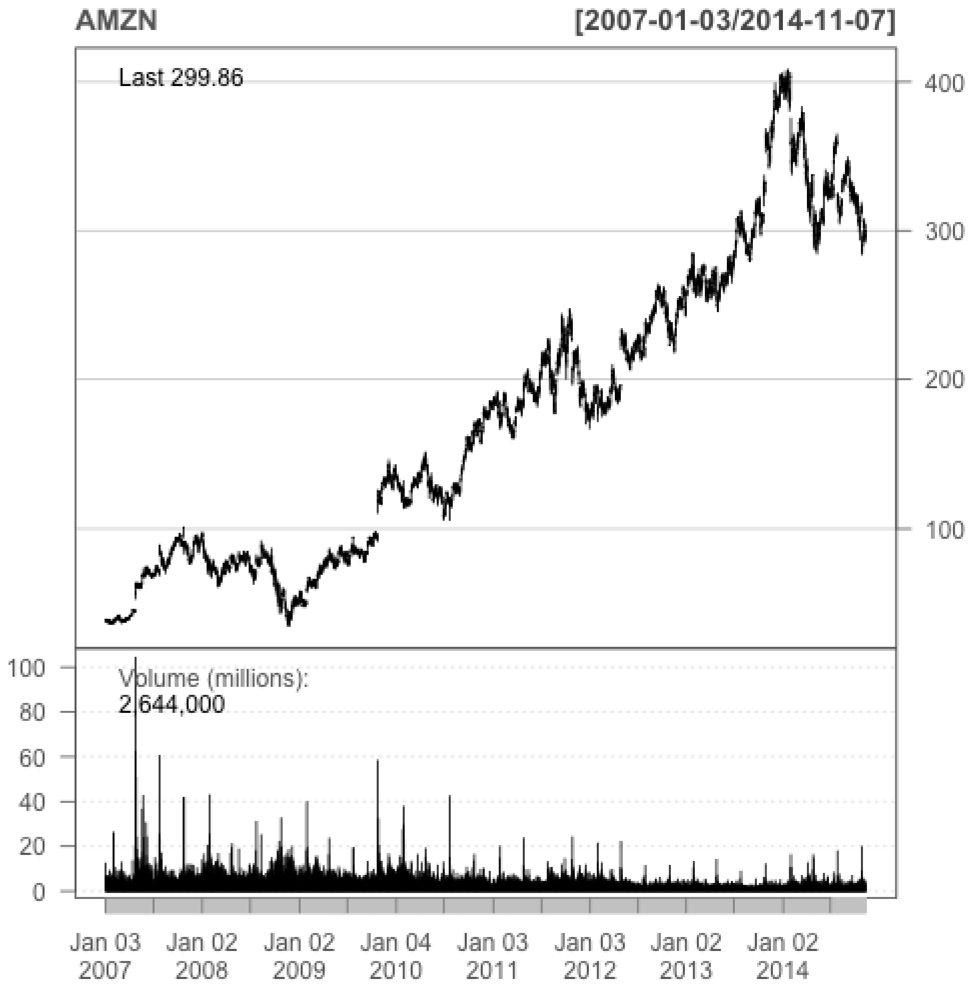
quantmod stands for *"Quantitative Financial Modeling and Trading Framework for R"*

It has many features so check out the help file for a full coverage or the Quantmod's official website.

Let's see how Amazon has been doing lately:

```
library(quantmod)
```

```
getSymbols(c("AMZN"))

barChart(AMZN,theme='white.mono',bar.type='hlc')
```

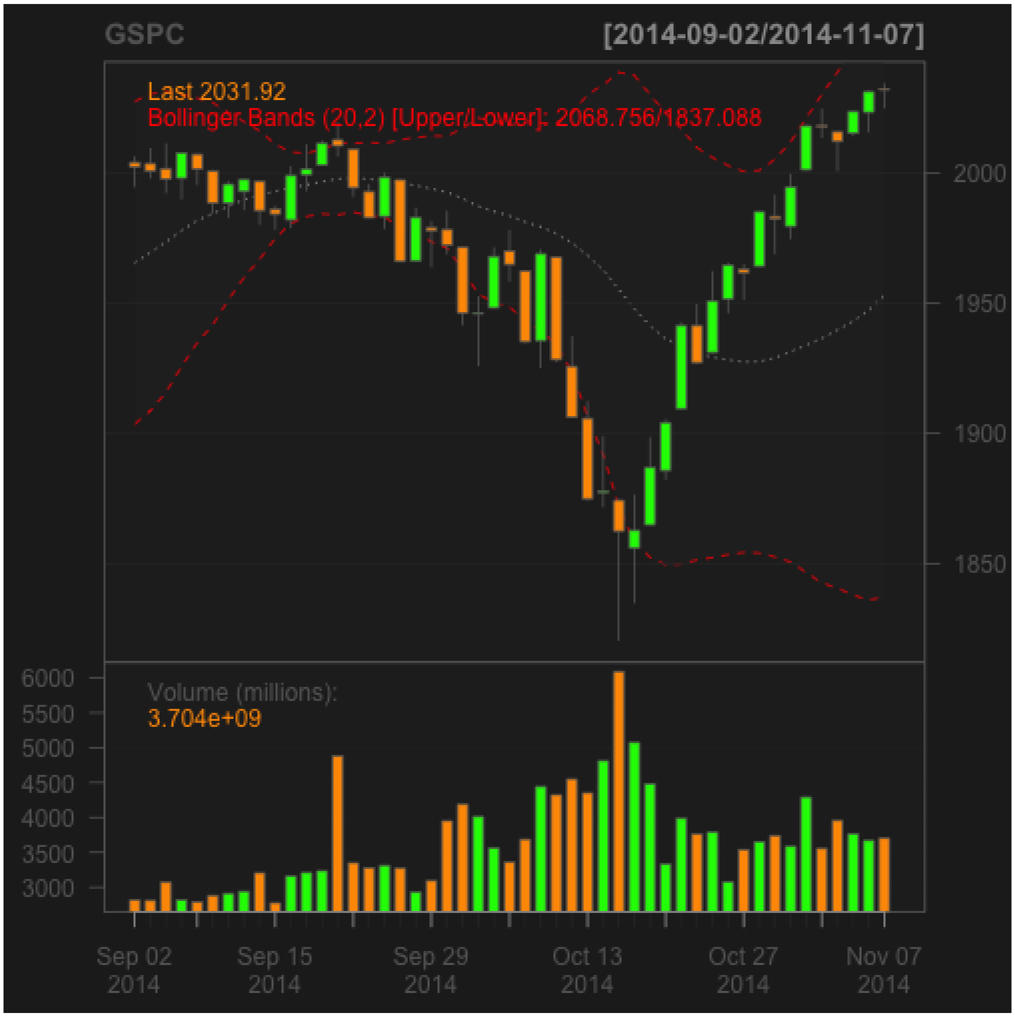


The `getSymbols` function downloaded daily data going all the way back to January 2007. The `barChart` function displays the data in a nice clean fashion following a theme-based parameter (see the help file for more). Not bad for 2 lines of code!!

It gets better - let's see how easy it is to display a full stock chart with indicators in just 3 lines of code:

```
getSymbols(c("^GSPC"))
```

```
chartSeries(GSPC, subset='last 3 months')

addBBands(n = 20, sd = 2, ma = "SMA", draw = 'bands', on = -1)
```



quantmod uses **Yahoo** to get its financial data. In the above example `^GSPC` represents the **S&P 500 Index**. Most financial product symbols are straightforward like **MSFT** for **Microsoft**. For indexes and other esoteric symbols, refer to finance.yahoo.com/lookup to see how they abbreviated it.

`chartSeries` is straightforward and will plot whatever symbol has been downloaded to memory using `getSymbols`. `addBBands` function will plot **Bollinger Bands** around your price series. There are many ways to customize the display, for some examples check out the Quantmod Gallery.

Quant Time

Moving deeper into quantitative finance, let's design a pattern-based system to predict whether a particular financial product will see a raise or drop in volume the following trading day.

We'll use **quantmod** to download all the stocks that compose the **NASDAQ 100 Index**. Then we'll `merge` together all our time series to synchronize them. This will collate the data by time and fill in any missing data with `NA`s:

```
Nasdaq100_Symbols <- c("AAPL", "ADBE", "ADI", "ADP", "ADSK", "AKAM", "ALTR", "ALXN",
"AMAT", "AMGN", "AMZN", "ATVI", "AVGO", "BBBY", "BIDU", "BIIB",
"BRCM", "CA", "CELG", "CERN", "CHKP", "CHRW", "CHTR", "CMCSA",
"COST", "CSCO", "CTRX", "CTSH", "CTXS", "DISCA", "DISCK", "DISH",
"DLTR", "DTV", "EBAY", "EQIX", "ESRX", "EXPD", "EXPE", "FAST",
"FB", "FFIV", "FISV", "FOXA", "GILD", "GMCR", "GOOG", "GOOGL",
"GRMN", "HSIC", "ILMN", "INTC", "INTU", "ISRG", "KLAC", "KRFT",
"LBTYA", "LLTC", "LMCA", "LMCK", "LVNTA", "MAR", "MAT", "MDLZ",
"MNST", "MSFT", "MU", "MXIM", "MYL", "NFLX", "NTAP", "NVDA",
"NXPI", "ORLY", "PAYX", "PCAR", "PCLN", "QCOM", "QVCA", "REGN",
"ROST", "SBAC", "SBUX", "SIAL", "SIRI", "SNDK", "SPLS", "SRCL",
"STX", "SYMC", "TRIP", "TSCO", "TSLA", "TXN", "VIAB", "VIP",
"VOD", "VRSK", "VRTX", "WDC", "WFM", "WYNN", "XLNX", "YHOO")
getSymbols(Nasdaq100_Symbols)
## pausing 1 second between requests for more than 5 symbols
...
##   [1] "AAPL"  "ADBE"  "ADI"   "ADP"   "ADSK"  "AKAM"  "ALTR"  "ALXN"
##   [9] "AMAT"  "AMGN"  "AMZN"  "ATVI"  "AVGO"  "BBBY"  "BIDU"  "BIIB"
##  [17] "BRCM"  "CA"    "CELG"  "CERN"  "CHKP"  "CHRW"  "CHTR"  "CMCSA"
##  [25] "COST"  "CSCO"  "CTRX"  "CTSH"  "CTXS"  "DISCA" "DISCK" "DISH"
##  [33] "DLTR"  "DTV"   "EBAY"  "EQIX"  "ESRX"  "EXPD"  "EXPE"  "FAST"
##  [41] "FB"    "FFIV"  "FISV"  "FOXA"  "GILD"  "GMCR"  "GOOG"  "GOOGL"
##  [49] "GRMN"  "HSIC"  "ILMN"  "INTC"  "INTU"  "ISRG"  "KLAC"  "KRFT"
##  [57] "LBTYA" "LLTC"  "LMCA"  "LMCK"  "LVNTA" "MAR"   "MAT"   "MDLZ" 
##  [65] "MNST"  "MSFT"  "MU"    "MXIM"  "MYL"   "NFLX"  "NTAP"  "NVDA" 
##  [73] "NXPI"  "ORLY"  "PAYX"  "PCAR"  "PCLN"  "QCOM"  "QVCA"  "REGN" 
##  [81] "ROST"  "SBAC"  "SBUX"  "SIAL"  "SIRI"  "SNDK"  "SPLS"  "SRCL" 
##  [89] "STX"   "SYMC"  "TRIP"  "TSCO"  "TSLA"  "TXN"   "VIAB"  "VIP"  
##  [97] "VOD"   "VRSK"  "VRTX"  "WDC"   "WFM"   "WYNN"  "XLNX"  "YHOO"
```

Be warned, that this does take a little time as **quantmod** will throttle the download. Each symbol is loaded in memory under the symbol name, therefore we have over 100 new objects loaded in memory each with years of daily market data. As these are independent time series, we have to merge everything together and fill in missing data so everything fits nicely in a data frame. We'll use the `merge.xts` function to merge by time all these objects into one data frame:

```
nasdaq100 <- data.frame(as.xts(merge(AAPL, ADBE, ADI, ADP, ADSK, AKAM,
                ALTR, ALXN, AMAT, AMGN, AMZN, ATVI, AVGO, BBBY, BIDU, BIIB,
                 BRCM, CA, CELG, CERN, CHKP, CHRW, CHTR, CMCSA,
                 COST, CSCO, CTRX, CTSH, CTXS, DISCA, DISCK, DISH,
                 DLTR, DTV, EBAY, EQIX, ESRX, EXPD, EXPE, FAST,
                 FB, FFIV, FISV, FOXA, GILD, GMCR, GOOG, GOOGL,
                 GRMN, HSIC, ILMN, INTC, INTU, ISRG, KLAC, KRFT,
                 LBTYA, LLTC, LMCA, LMCK, LVNTA, MAR, MAT, MDLZ,
                 MNST, MSFT, MU, MXIM, MYL, NFLX, NTAP, NVDA,
                 NXPI, ORLY, PAYX, PCAR, PCLN, QCOM, QVCA, REGN,
                 ROST, SBAC, SBUX, SIAL, SIRI, SNDK, SPLS, SRCL,
                 STX, SYMC, TRIP, TSCO, TSLA, TXN, VIAB, VIP,
                 VOD, VRSK, VRTX, WDC, WFM, WYNN, XLNX, YHOO)))
head(nasdaq100[,1:12],2)
##            AAPL.Open AAPL.High AAPL.Low AAPL.Close AAPL.Volume
## 2007-01-03     86.29     86.58    81.90      83.80   309579900
## 2007-01-04     84.05     85.95    83.82      85.66   211815100
##            AAPL.Adjusted ADBE.Open ADBE.High ADBE.Low ADBE.Close
## 2007-01-03         11.34     40.72     41.32    38.89      39.92
## 2007-01-04         11.59     39.88     41.00    39.43      40.82
##            ADBE.Volume ADBE.Adjusted
## 2007-01-03     7126000         39.92

## 2007-01-04     4503700         40.82
```

Now that we have a handful of years of market data for every stock currently in the **NASDAQ 100 Index**, we need to do something with it. We're going to create a variety of measures between price and volume points. The idea is to quantify stock moves as patterns by subtracting one day versus a previous one. We'll create a series of differences:

- 1 day versus 2 days ago
- 1 day versus 3 days ago
- 1 day versus 5 days ago
- 1 day versus 20 days ago (akin a 20 day moving average)

Creating The Outcome Variable

This is the heart of the system and it's a bit tedious so hold on. What are we trying to predict? Whether the next trading day's volume for a chosen symbol will be higher or lower than the current trading day (this doesn't have to be the `volume` field of **FISV**, it could be the `high` or `close` of any other symbol for which we have data):

```
outcomeSymbol <- 'FISV.Volume'
```

Shift the result we're trying to predict down one trading day using the `lag` function. This will add the `volume` field of our outcome symbol with a lag of 1 trading day so it's on the same line as the predictors. We will rely on this value for training and testing purposes. A value of `1` means the volume went up, and a `0`, that it went down:

```
library(xts)

nasdaq100 <- xts(nasdaq100,order.by=as.Date(rownames(nasdaq100)))

nasdaq100 <- as.data.frame(merge(nasdaq100, lm1=lag(nasdaq100[,outcomeSymbol],-1)))
```

```
nasdaq100$outcome <- ifelse(nasdaq100[,paste0(outcomeSymbol,'.1')] > nasdaq100[,outcomeSymbol]
, 1, 0)
# remove shifted down volume field as we don't care by the value
nasdaq100 <- nasdaq100[,!names(nasdaq100) %in% c(paste0(outcomeSymbol,'.1'))]
```

Cast the date field to type `date` as it currently is of type `character` and sort by decreasing order:

```
nasdaq100$date <- as.Date(row.names(nasdaq100))
nasdaq100 <- nasdaq100[order(as.Date(nasdaq100$date, "%m/%d/%Y"), decreasing = TRUE),]
```

Here is the pattern maker function. This will take our raw market data and `scale` it so that we can compare any symbol with any other symbol. It then subtracts the different day ranges requested by the `days` parameter using the `diff` and `lag` calls and puts them all on the same row along with the outcome. To make things even more compatible, the `roundByScaler` parameter can round results.

```
GetDiffDays <- function(objDF,days=c(10), offLimitsSymbols=c('outcome'), roundByScaler=3) {
        # needs to be sorted by date in decreasing order
        ind <- sapply(objDF, is.numeric)
        for (sym in names(objDF)[ind]) {
                if (!sym %in% offLimitsSymbols) {
                        print(paste('*********', sym))
                        objDF[,sym] <- round(scale(objDF[,sym]),roundByScaler)

                        print(paste('theColName', sym))
                        for (day in days) {
                                objDF[paste0(sym,'_',day)] <- c(diff(objDF[,sym],lag = day),re
p(x=0,day)) * -1
```

```
                        }
                }
        }
        return (objDF)
}
```

Call the function with the following differences and scale it down to 2 decimal points (this takes a little while to run):

```
nasdaq100 <- GetDiffDays(nasdaq100, days=c(1,2,3,4,5,10,20), offLimitsSymbols=c('outcome'), roundByScaler=2)
## [1] "********* AAPL.Open"
## [1] "theColName AAPL.Open"
## [1] "********* AAPL.High"
## [1] "theColName AAPL.High"
## [1] "********* AAPL.Low"
## [1] "theColName AAPL.Low"
## [1] "********* AAPL.Close"
## [1] "theColName AAPL.Close"
## [1] "********* AAPL.Volume"
## [1] "theColName AAPL.Volume"
## [1] "********* AAPL.Adjusted"
## [1] "theColName AAPL.Adjusted"
...
## [1] "********* YHOO.Open"
## [1] "theColName YHOO.Open"
## [1] "********* YHOO.High"
## [1] "theColName YHOO.High"
## [1] "********* YHOO.Low"

## [1] "theColName YHOO.Low"

## [1] "********* YHOO.Close"

## [1] "theColName YHOO.Close"

## [1] "********* YHOO.Volume"

## [1] "theColName YHOO.Volume"

## [1] "********* YHOO.Adjusted"

## [1] "theColName YHOO.Adjusted"
```

We delete the last row from our data frame as it doesn't have an `outcome` variable (that is in the **future**):

```
nasdaq100 <- nasdaq100[2:nrow(nasdaq100),]
```

Let's take a peek at our resulting features for Yahoo:

```
dput(names(nasdaq100)[grepl('YHOO.',names(nasdaq100))])

## c("YHOO.Open", "YHOO.High", "YHOO.Low", "YHOO.Close", "YHOO.Volume",

## "YHOO.Adjusted", "YHOO.Open_1", "YHOO.Open_2", "YHOO.Open_3",

## "YHOO.Open_4", "YHOO.Open_5", "YHOO.Open_10", "YHOO.Open_20",

## "YHOO.High_1", "YHOO.High_2", "YHOO.High_3", "YHOO.High_4", "YHOO.High_5",

## "YHOO.High_10", "YHOO.High_20", "YHOO.Low_1", "YHOO.Low_2", "YHOO.Low_3",

## "YHOO.Low_4", "YHOO.Low_5", "YHOO.Low_10", "YHOO.Low_20", "YHOO.Close_1",

## "YHOO.Close_2", "YHOO.Close_3", "YHOO.Close_4", "YHOO.Close_5",

## "YHOO.Close_10", "YHOO.Close_20", "YHOO.Volume_1", "YHOO.Volume_2",

## "YHOO.Volume_3", "YHOO.Volume_4", "YHOO.Volume_5", "YHOO.Volume_10",
## "YHOO.Volume_20", "YHOO.Adjusted_1", "YHOO.Adjusted_2", "YHOO.Adjusted_3",
## "YHOO.Adjusted_4", "YHOO.Adjusted_5", "YHOO.Adjusted_10", "YHOO.Adjusted_20"
## )
```

We extract the day of the week, day of the month, day of the year as predictors using `POSIXlt`:

```
nasdaq100$wday <- as.POSIXlt(nasdaq100$date)$wday
nasdaq100$yday <- as.POSIXlt(nasdaq100$date)$mday
nasdaq100$mon <- as.POSIXlt(nasdaq100$date)$mon
```

Next we remove the date field as it won't help us as a predictor as they are all unique and we shuffle the data set using the `sample` function:

```
nasdaq100 <- subset(nasdaq100, select=-c(date))
nasdaq100 <- nasdaq100[sample(nrow(nasdaq100)),]
```

Let's Model It!

We'll use a simple **xgboost** model to get an `AUC` score. You can get more details regarding parameter settings for this model at the xgboost wiki:

```
library(xgboost)
predictorNames <- names(nasdaq100)[names(nasdaq100) != 'outcome']

set.seed(1234)
split <- sample(nrow(nasdaq100), floor(0.7*nrow(nasdaq100)))
```

```
train <-nasdaq100[split,]
test <- nasdaq100[-split,]

bst <- xgboost(data = as.matrix(train[,predictorNames]),
               label = train$outcome,
               verbose=0,
               eta = 0.1,
               gamma = 50,
               nround = 50,
               colsample_bytree = 0.1,
               subsample = 8.6,
               objective="binary:logistic")

predictions <- predict(bst, as.matrix(test[,predictorNames]), outputmargin=TRUE)

library(pROC)
auc <- roc(test$outcome, predictions)
print(paste('AUC score:', auc$auc))
## [1] "AUC score: 0.719931972789116"
```

Conclusion

Not bad, right? An **AUC** of **0.719** for very little work (remember that an **AUC** ranges between **0.5** and **1**, where **0.5** is random and **1** is perfect). Hopefully this will pique your imagination with the many possibilities of **quantmod** and **R**.

Full source code (also on GitHub):

```
library(quantmod)

# display a simple bar chart
getSymbols(c("AMZN"))
barChart(AMZN,theme='white.mono',bar.type='hlc')

# display a complex chart
getSymbols(c("^GSPC"))
chartSeries(GSPC, subset='last 3 months')
addBBands(n = 20, sd = 2, ma = "SMA", draw = 'bands', on = -1)

# get market data for all symbols making up the NASDAQ 100 Index
Nasdaq100_Symbols <- c("AAPL", "ADBE", "ADI", "ADP", "ADSK", "AKAM", "ALTR", "ALXN",
"AMAT", "AMGN", "AMZN", "ATVI", "AVGO", "BBBY", "BIDU", "BIIB",
"BRCM", "CA", "CELG", "CERN", "CHKP", "CHRW", "CHTR", "CMCSA",
"COST", "CSCO", "CTRX", "CTSH", "CTXS", "DISCA", "DISCK", "DISH",
"DLTR", "DTV", "EBAY", "EQIX", "ESRX", "EXPD", "EXPE", "FAST",
"FB", "FFIV", "FISV", "FOXA", "GILD", "GMCR", "GOOG", "GOOGL",
"GRMN", "HSIC", "ILMN", "INTC", "INTU", "ISRG", "KLAC", "KRFT",
"LBTYA", "LLTC", "LMCA", "LMCK", "LVNTA", "MAR", "MAT", "MDLZ",
"MNST", "MSFT", "MU", "MXIM", "MYL", "NFLX", "NTAP", "NVDA",
"NXPI", "ORLY", "PAYX", "PCAR", "PCLN", "QCOM", "QVCA", "REGN",
"ROST", "SBAC", "SBUX", "SIAL", "SIRI", "SNDK", "SPLS", "SRCL",
"STX", "SYMC", "TRIP", "TSCO", "TSLA", "TXN", "VIAB", "VIP",
"VOD", "VRSK", "VRTX", "WDC", "WFM", "WYNN", "XLNX", "YHOO")
getSymbols(Nasdaq100_Symbols)

# merge them all together
nasdaq100 <- data.frame(as.xts(merge(AAPL, ADBE, ADI, ADP, ADSK, AKAM,
                ALTR, ALXN,AMAT, AMGN, AMZN, ATVI, AVGO, BBBY, BIDU, BIIB,
                 BRCM, CA, CELG, CERN, CHKP, CHRW, CHTR, CMCSA,
                 COST, CSCO, CTRX, CTSH, CTXS, DISCA, DISCK, DISH,
                 DLTR, DTV, EBAY, EQIX, ESRX, EXPD, EXPE, FAST,
                 FB, FFIV, FISV, FOXA, GILD, GMCR, GOOG, GOOGL,
                 GRMN, HSIC, ILMN, INTC, INTU, ISRG, KLAC, KRFT,
                 LBTYA, LLTC, LMCA, LMCK, LVNTA, MAR, MAT, MDLZ,
                 MNST, MSFT, MU, MXIM, MYL, NFLX, NTAP, NVDA,
                 NXPI, ORLY, PAYX, PCAR, PCLN, QCOM, QVCA, REGN,
                 ROST, SBAC, SBUX, SIAL, SIRI, SNDK, SPLS, SRCL,
                 STX, SYMC, TRIP, TSCO, TSLA, TXN, VIAB, VIP,
                 VOD, VRSK, VRTX, WDC, WFM, WYNN, XLNX, YHOO)))
head(nasdaq100[,1:12],2)

# set outcome variable
outcomeSymbol <- 'FISV.Volume'

# shift outcome value to be on same line as predictors
library(xts)
nasdaq100 <- xts(nasdaq100,order.by=as.Date(rownames(nasdaq100)))
nasdaq100 <- as.data.frame(merge(nasdaq100, lm1=lag(nasdaq100[,outcomeSymbol],-1)))
nasdaq100$outcome <- ifelse(nasdaq100[,paste0(outcomeSymbol,'.1')] > nasdaq100[,outcomeSymbol], 1, 0)

# remove shifted down volume field as we don't care by the value
nasdaq100 <- nasdaq100[,!names(nasdaq100) %in% c(paste0(outcomeSymbol,'.1'))]

# cast date to true date and order in decreasing order
nasdaq100$date <- as.Date(row.names(nasdaq100))
nasdaq100 <- nasdaq100[order(as.Date(nasdaq100$date, "%m/%d/%Y"), decreasing = TRUE),]

# calculate all day differences and populate them on same row
GetDiffDays <- function(objDF,days=c(10), offLimitsSymbols=c('outcome'), roundByScaler=3) {
        # needs to be sorted by date in decreasing order
        ind <- sapply(objDF, is.numeric)
        for (sym in names(objDF)[ind]) {
                if (!sym %in% offLimitsSymbols) {
                        print(paste('*********', sym))
                        objDF[,sym] <- round(scale(objDF[,sym]),roundByScaler)

                        print(paste('theColName', sym))
                        for (day in days) {
                                objDF[paste0(sym,'_',day)] <- c(diff(objDF[,sym],lag = day),rep(x=0,day))
* -1
                        }
                }
        }
        return (objDF)
```

```
}

# call the function with the following differences
nasdaq100 <- GetDiffDays(nasdaq100, days=c(1,2,3,4,5,10,20), offLimitsSymbols=c('outcome'), roundByScaler=
2)

# drop most recent entry as we don't have an outcome
nasdaq100 <- nasdaq100[2:nrow(nasdaq100),]

# take a peek at YHOO features:
dput(names(nasdaq100)[grepl('YHOO.',names(nasdaq100))])

# well use POSIXlt to add day of the week, day of the month, day of the year
nasdaq100$wday <- as.POSIXlt(nasdaq100$date)$wday
nasdaq100$yday <- as.POSIXlt(nasdaq100$date)$mday
nasdaq100$mon<- as.POSIXlt(nasdaq100$date)$mon

# remove date field and shuffle data frame
nasdaq100 <- subset(nasdaq100, select=-c(date))
nasdaq100 <- nasdaq100[sample(nrow(nasdaq100)),]

# let's model
library(xgboost)
predictorNames <- names(nasdaq100)[names(nasdaq100) != 'outcome']

set.seed(1234)
split <- sample(nrow(nasdaq100), floor(0.7*nrow(nasdaq100)))
train <-nasdaq100[split,]
test <- nasdaq100[-split,]

bst <- xgboost(data = as.matrix(train[,predictorNames]),
               label = train$outcome,
               verbose=0,
               eta = 0.1,
               gamma = 50,
               nround = 50,
               colsample_bytree = 0.1,
               subsample = 8.6,
               objective="binary:logistic")
```

```
predictions <- predict(bst, as.matrix(test[,predictorNames]), outputmargin=TRUE)


library(pROC)
auc <- roc(test$outcome, predictions)
print(paste('AUC score:', auc$auc))
```

Predicting Multiple Discrete Values with Multinomials, Neural Networks and the {nnet} Package

November 1, 2014 **Tags:** ***modeling***

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{nnet}** - neural network multinomial modeling
- **{RCurl}** - downloads https data
- **{caret}** - dummyVars and postResample function

So, what is a **multionmial model**?

From Wikipedia:

Multinomial logistic regression is a simple extension of binary logistic regression that allows for more than two categories of the dependent or outcome variable.

And from the `multinom` **{nnet}** help file:

```
library(nnet)

?multinom
```

Fits multinomial log-linear models via neural networks.

In a nutshell, this allows you to predict a factor of multiple levels (more than two) in one shot with the power of neural networks. **Neural networks** are great at working through multiple combinations and also great with linear models, so it's an ideal combination.

If your data is linear in nature, then instead of using multiple models and doing `A` versus `B`, `B` versus `C`, and `C` versus `A`, and finally going through the hassle of concatenating the resulting probabilities, you can let **nnet** do it all in one shot. And this becomes exponentialy more difficult as you predict more than 3 outcome levels!!

The `multinom` function will do all that for you in one shot and allow you to observe the probabilities of each subset to interpret things (now that's really cool).

Let's code!

We're going to use a Hadley Wickham data set to predict how many cylinders a vehicle has. We download the data from Github:

```
library(RCurl)

urlfile <-'https://raw.githubusercontent.com/hadley/fueleconomy/master/data-raw/vehicles.csv'

x <- getURL(urlfile, ssl.verifypeer = FALSE)

vehicles <- read.csv(textConnection(x))
```

Only use the first 24 columns of the data for simplicities sake. Cast all variables to numerics and impute any NAs with `0`.

```
vehicles <- vehicles[names(vehicles)[1:24]]
vehicles <- data.frame(lapply(vehicles, as.character), stringsAsFactors=FALSE)
vehicles <- data.frame(lapply(vehicles, as.numeric))
vehicles[is.na(vehicles)] <- 0
names(vehicles)
##  [1] "barrels08"       "barrelsA08"      "charge120"
##  [4] "charge240"       "city08"          "city08U"
##  [7] "cityA08"         "cityA08U"        "cityCD"
## [10] "cityE"           "cityUF"          "co2"
## [13] "co2A"            "co2TailpipeAGpm" "co2TailpipeGpm"
## [16] "comb08"          "comb08U"         "combA08"
## [19] "combA08U"        "combE"           "combinedCD"
## [22] "combinedUF"      "cylinders"       "displ"
```

Use the `cyclinder` column as the model's outcome and cast it to a factor. Use the `table` function to see how many types of cylinders we are dealing with (BTW a `0` cylinder vehicle is an electric vehicle):

```
vehicles$cylinders <- as.factor(vehicles$cylinders)
table(vehicles$cylinders)
##
##     0     2     3     4     5     6     8    10    12    16
##    66    51   182 13133   757 12101  7715   138   481     7
```

We see that the 4 and 6 cylinder vehicles are the most numerous.

Shuffle the data and split it into two equal data frames so we can have a training and a testing data set:

```
set.seed(1234)
vehicles <- vehicles[sample(nrow(vehicles)),]
split <- floor(nrow(vehicles)/2)
vehiclesTrain <- vehicles[0:split,]
vehiclesTest <- vehicles[(split+1):nrow(vehicles),]
```

Let's put **nnet** to work and predict cyclinders. The `maxiter` variable defaults to 100 when omitted so let's start with a large number during the first round to make sure we find the lowest possible error level (i.e. global minimum - solution with the lowest error possible):

```
library(nnet)
```

```
cylModel <- multinom(cylinders~., data=vehiclesTrain, maxit=500, trace=T)
## # weights:  250 (216 variable)
## initial  value 39869.260885
## iter  10 value 18697.133750
...
## iter 420 value 5217.401201
## final  value 5217.398483
## converged
```

When you see the word **converged** in the log output, you know the model went as far as it could.

Let's find the most influential variables by using **caret's** `varImp` function:

```
library(caret)
mostImportantVariables <- varImp(cylModel)
mostImportantVariables$Variables <- row.names(mostImportantVariables)
mostImportantVariables <- mostImportantVariables[order(-mostImportantVariables$Overall),]
print(head(mostImportantVariables))
## Variables  Overall
## charge240  625.5732
## cityUF     596.4079
## combinedUF 580.1112
## displ      434.8038
## cityE      395.3533
## combA08    322.2910
```

Next we predict `cylinders` using the `predict` function on the testing data set. There are two ways to compute predictions, `class` or `probs`:

```
preds1 <- predict(cylModel, type="probs", newdata=vehiclesTest)
head(preds1)
##               0          2          3         4         5         6
## 30632  1.966e-53  2.770e-38  3.232e-40 3.024e-02 4.728e-02 0.9222608
## 26204  4.310e-33 8.316e-112  5.884e-21 9.297e-01 3.009e-02 0.0401936
## 27378  1.211e-92  7.384e-65  5.948e-75 7.352e-09 2.823e-05 0.2919979
## 12346  2.808e-48  9.065e-29  3.301e-37 4.767e-02 9.357e-02 0.8580807
## 13664  2.428e-27  4.287e-33  8.640e-21 9.643e-01 2.190e-02 0.0137845
## 3357  1.654e-146 2.447e-119 2.731e-111 3.014e-17 2.733e-10 0.0004251
##               8        10        12         16
## 30632 2.198e-04 4.059e-09 2.837e-09  2.457e-89
## 26204 1.523e-06 6.732e-13 3.522e-16  4.859e-95
## 27378 7.019e-01 3.665e-03 2.452e-03  9.845e-46
## 12346 6.820e-04 9.299e-08 1.057e-07  3.385e-87
## 13664 7.224e-08 6.902e-14 7.690e-15 1.406e-111
## 3357  8.994e-01 2.395e-02 7.627e-02  4.517e-45
preds2 <- predict(cylModel, type="class", newdata=vehiclesTest)
head(preds2)
## [1] 6 4 8 6 4 8
## Levels: 0 2 3 4 5 6 8 10 12 16
```

Choosing which of the two predictions will depend on your needs. If you just want your `cylinders` predictions, use `class`, if you need to do anything more complex, like measure the conviction of each prediction, use the `probs` option (every row will add up to 1).

To check the **accuracy** of the model, we call the `postResample` function from **caret**. For numeric vectors, it uses the mean squared error and R-squared and for factors, the overall agreement rate and Kappa:

```
postResample(vehiclesTest$cylinders,preds2)
## Accuracy    Kappa
##   0.9034   0.8566
```

As a bonus, let’s do some simple repeated cross validation to get a more comprehensive mean accuracy score and understand convergence. The code below will iterate through all the data to give every variable a chance of being **test** and **train** data sets. The first time around we set `maxit` to only **50**:

```
totalAccuracy <- c()
```

```
cv <- 10
cvDivider <- floor(nrow(vehicles) / (cv+1))

for (cv in seq(1:cv)) {
  # assign chunk to data test
  dataTestIndex <- c((cv * cvDivider):(cv * cvDivider + cvDivider))
  dataTest <- vehicles[dataTestIndex,]
  # everything else to train
  dataTrain <- vehicles[-dataTestIndex,]

  cylModel <- multinom(cylinders~., data=dataTrain, maxit=50, trace=T)

  pred <- predict(cylModel, newdata=dataTest, type="class")

  #  classification error
  cv_ac <- postResample(dataTest$cylinders, pred)[[1]]
  print(paste('Current Accuracy:',cv_ac,'for CV:',cv))
  totalAccuracy <- c(totalAccuracy, cv_ac)
}
## stopped after 50 iterations
## [1] "Current Accuracy: 0.62559542711972 for CV: 1"
...
## stopped after 50 iterations
## [1] "Current Accuracy: 0.650682756430613 for CV: 2"
...
## stopped after 50 iterations
## [1] "Current Accuracy: 0.613210543029533 for CV: 3"
...
## stopped after 50 iterations
## [1] "Current Accuracy: 0.657669101302001 for CV: 4"
...
## stopped after 50 iterations
## [1] "Current Accuracy: 0.607494442680216 for CV: 5"
...
## stopped after 50 iterations
## [1] "Current Accuracy: 0.644649094950778 for CV: 6"
...
## stopped after 50 iterations
## [1] "Current Accuracy: 0.70593839314068 for CV: 7"
...
## stopped after 50 iterations
## [1] "Current Accuracy: 0.621149571292474 for CV: 8"
...
## stopped after 50 iterations
## [1] "Current Accuracy: 0.59892029215624 for CV: 9"
...
## stopped after 50 iterations
## [1] "Current Accuracy: 0.62432518259765 for CV: 10"
 mean(totalAccuracy)
## [1] 0.635
```

The **mean accuracy** of **0.635** is much lower than the accuracy of **0.9034** that we got with the original simple split. You will notice that the log output never prints the word **converged**. This means the model never reaches the lowest error or global minima and therefore isn't the best fit.

Let's try this again and let the model converge by setting the `maxit` to a large number

```
totalAccuracy <- c()
cv <- 10
cvDivider <- floor(nrow(vehicles) / (cv+1))

for (cv in seq(1:cv)) {
  # assign chunk to data test
  dataTestIndex <- c((cv * cvDivider):(cv * cvDivider + cvDivider))
  dataTest <- vehicles[dataTestIndex,]
  # everything else to train
  dataTrain <- vehicles[-dataTestIndex,]

  cylModel <- multinom(cylinders~., data=dataTrain, maxit=1000, trace=T)

  pred <- predict(cylModel, newdata=dataTest, type="class")

  #  classification error
  cv_ac <- postResample(dataTest$cylinders, pred)[[1]]
  print(paste('Current Accuracy:',cv_ac,'for CV:',cv))
  totalAccuracy <- c(totalAccuracy, cv_ac)
}
## converged
## [1] "Current Accuracy: 0.905366783105748 for CV: 1"


## converged
## [1] "Current Accuracy: 0.89838043823436 for CV: 2"


## converged
## [1] "Current Accuracy: 0.909812638932995 for CV: 3"


## converged
## [1] "Current Accuracy: 0.904096538583677 for CV: 4"


## converged
## [1] "Current Accuracy: 0.906637027627818 for CV: 5"


## converged
## [1] "Current Accuracy: 0.906001905366783 for CV: 6"


## converged
## [1] "Current Accuracy: 0.911400444585583 for CV: 7"


## converged
## [1] "Current Accuracy: 0.902508732931089 for CV: 8"


## converged
## [1] "Current Accuracy: 0.90377897745316 for CV: 9"


## converged
## [1] "Current Accuracy: 0.904414099714195 for CV: 10"
mean(totalAccuracy)

## [1] 0.9052
```

The score using the **repeated cross validation** code is better than the original simple split of **0.9304** and we let each loop converge. The point of using the **repeated cross validation** code isn't that it will return a higher accuracy score (and it doesn't always) but that it will give you a much more accurate score as it uses all of your data.

Full source code (also on GitHub):

```
library(nnet)
?multinom

library(caret)
library(RCurl)
library(Metrics)

######################################################################
# Load data from Hadley Wickham on Github
urlfile <-'https://raw.githubusercontent.com/hadley/fueleconomy/master/data-raw/vehicles.csv'
x <- getURL(urlfile, ssl.verifypeer = FALSE)
vehicles <- read.csv(textConnection(x))

# clean up the data and only use the first 24 columns
vehicles <- vehicles[names(vehicles)[1:24]]
vehicles <- data.frame(lapply(vehicles, as.character), stringsAsFactors=FALSE)
vehicles <- data.frame(lapply(vehicles, as.numeric))
vehicles[is.na(vehicles)] <- 0

# use cyclinder column as outcome and cast to factor
vehicles$cylinders <- as.factor(vehicles$cylinders)
table(vehicles$cylinders)
######################################################################


# shuffle and split
set.seed(1234)
vehicles <- vehicles[sample(nrow(vehicles)),]
split <- floor(nrow(vehicles)/2)
vehiclesTrain <- vehicles[0:split,]
vehiclesTest <- vehicles[(split+1):nrow(vehicles),]


# see how multinom predicts cylinders
# set the maxit to a large number, enough so the neural net can converge to smallest error
cylModel <- multinom(cylinders~., data=vehiclesTrain, maxit=500, trace=T)

# Sort by most influential variables
topModels <- varImp(cylModel)
topModels$Variables <- row.names(topModels)
topModels <- topModels[order(-topModels$Overall),]

# class/probs (best option, second best option?)
preds1 <- predict(cylModel, type="probs", newdata=vehiclesTest)
preds2 <- predict(cylModel, type="class", newdata=vehiclesTest)

# resample for accuracy - the mean squared error and R-squared are calculated of forfactors, the overall agreement rate and Kappa
postResample(vehiclesTest$cylinders,preds2)[[1]]
preds

library(Metrics)
classificationError <- ce(as.numeric(vehiclesTest$cylinders), as.numeric(preds))

# repeat cross validate by iterating through all the data to give every variable a chance of being test and train portions
totalError <- c()
cv <- 10
maxiterations <- 500 # try it again with a lower value and notice the mean error
cvDivider <- floor(nrow(vehiclesTrain) / (cv+1))

for (cv in seq(1:cv)) {
        # assign chunk to data test
        dataTestIndex <- c((cv * cvDivider):(cv * cvDivider + cvDivider))
        dataTest <- vehiclesTrain[dataTestIndex,]
        # everything else to train
        dataTrain <- vehiclesTrain[-dataTestIndex,]

        cylModel <- multinom(cylinders~., data=dataTrain, maxit=maxiterations, trace=T)

        pred <- predict(cylModel, dataTest)

        #  classification error
        err <- ce(as.numeric(dataTest$cylinders), as.numeric(pred))

        totalError <- c(totalError, err)

}

print(paste('Mean error of all repeated cross validations:',mean(totalError)))
```

Modeling 101 - Predicting Binary Outcomes with R, gbm, glmnet, and {caret}

October 22, 2014 **Tags:** ***visualizing, modeling***

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{caret}** - modeling wrapper, functions, commands
- **{pROC}** - Area Under the Curve (AUC) functions

This is an introduction to modeling binary outcomes using the caret library. A binary outcome is a result that has two possible values - true or false, alive or dead, etc.

We're going to use two models: gbm (Generalized Boosted Models) and glmnet (Generalized Linear Models). Approaching a new data set using different models is one way of getting a handle on your data. **Gbm** uses boosted trees while **glmnet** uses regression.

Let's code!

We're going to use the **Titanic** data set from the **University of Colorado Denver**:

```
titanicDF <- read.csv('http://math.ucdenver.edu/RTutorial/titanic.txt',sep='\t')
```

```
print(str(titanicDF))

## 'data.frame':    1313 obs. of  5 variables:

##  $ Name    : Factor w/ 1310 levels "Abbing, Mr Anthony",..: 22 25 26 27 24 31 45 46 50 54 ...

##  $ PClass  : Factor w/ 3 levels "1st","2nd","3rd": 1 1 1 1 1 1 1 1 1 1 ...

##  $ Age     : num  29 2 30 25 0.92 47 63 39 58 71 ...

##  $ Sex     : Factor w/ 2 levels "female","male": 1 1 2 1 2 2 1 2 1 2 ...

##  $ Survived: int  1 0 0 0 1 1 1 0 1 0 ...

## NULL
```

We need to clean up a few things as is customary with any data science project.
The `Name` variable is mostly unique so we're going to extract the title and throw the rest away.

```
titanicDF$Title <- ifelse(grepl('Mr ',titanicDF$Name),'Mr',ifelse(grepl('Mrs ',titanicDF$Name),'Mrs',ifelse(grepl('Miss',titanicDF$Name),'Miss','Nothing')))
```

The `Age` variable has missing data (i.e. `NA`'s) so we're going to impute it with the mean value of all the available ages. There are many ways of imputing missing data - we could delete those rows, set the values to 0, etc. Either way, this will neutralize the missing fields with a common value, and allow the models that can't handle them normally to function (**gbm** can handle `NA`s but **glmnet** cannot):

```
titanicDF$Age[is.na(titanicDF$Age)] <- median(titanicDF$Age, na.rm=T)
```

It is customary to have the **outcome** variable (also known as response variable) located in the last column of a data set:

```
titanicDF <- titanicDF[c('PClass', 'Age',    'Sex',   'Title', 'Survived')]

print(str(titanicDF))

## 'data.frame':    1313 obs. of  5 variables:

##  $ PClass  : Factor w/ 3 levels "1st","2nd","3rd": 1 1 1 1 1 1 1 1 1 1 ...

##  $ Age     : num  29 2 30 25 0.92 47 63 39 58 71 ...

##  $ Sex     : Factor w/ 2 levels "female","male": 1 1 2 1 2 2 1 2 1 2 ...

##  $ Title   : chr  "Miss" "Miss" "Mr" "Mrs" ...

##  $ Survived: int  1 0 0 0 1 1 1 0 1 0 ...

## NULL
```

Our data is starting to look good but we have to fix the **factor** variables as most models only accept **numeric** data. Again, **gbm** can deal with factor variables as it will dummify them internally, but **glmnet** won't. In a nutshell, dummifying factors breaks all the unique values into separate columns (see my post on Brief Walkthrough Of The dummyVars function from {caret}). This is a **caret** function:

```
titanicDF$Title <- as.factor(titanicDF$Title)

titanicDummy <- dummyVars("~.",data=titanicDF, fullRank=F)

titanicDF <- as.data.frame(predict(titanicDummy,titanicDF))

print(names(titanicDF))

##  [1] "PClass.1st"    "PClass.2nd"    "PClass.3rd"    "Age"

##  [5] "Sex.female"    "Sex.male"      "Title.Miss"    "Title.Mr"

##  [9] "Title.Mrs"     "Title.Nothing" "Survived"
```

As you can see, each unique factor is now separated into its own column. Next, we need to understand the proportion of our outcome variable:

```
prop.table(table(titanicDF$Survived))
```

```
##

##      0      1

## 0.6573 0.3427
```

This tells us that **34.27%** of our data contains survivors of the Titanic tragedy. This is an important step because if the proportion was smaller than 15%, it would be considered a **rare event** and would be more challenging to model.

I like generalizing my variables so that I can easily recycle the code for subsequent needs:

```
outcomeName <- 'Survived'

predictorsNames <- names(titanicDF)[names(titanicDF) != outcomeName]
```

Let's model!

Even though we already know the models we're going to use in this walkthrough, **caret** supports a huge number of models. Here is how to get the current list of supported models:

```
names(getModelInfo())
##   [1] "ada"                 "ANFIS"               "avNNet"
##   [4] "bag"                 "bagEarth"            "bagFDA"
##   [7] "bayesglm"            "bdk"                 "blackboost"
##  [10] "Boruta"              "brnn"                "bstLs"
##  [13] "bstSm"               "bstTree"             "C5.0"
##  [16] "C5.0Cost"            "C5.0Rules"           "C5.0Tree"
##  [19] "cforest"             "CSimca"              "ctree"
##  [22] "ctree2"              "cubist"              "DENFIS"
##  [25] "dnn"                 "earth"               "elm"
##  [28] "enet"                "evtree"              "extraTrees"
##  [31] "fda"                 "FH.GBML"             "FIR.DM"
##  [34] "foba"                "FRBCS.CHI"           "FRBCS.W"
##  [37] "FS.HGD"              "gam"                 "gamboost"
##  [40] "gamLoess"            "gamSpline"           "gaussprLinear"
##  [43] "gaussprPoly"         "gaussprRadial"       "gbm"
##  [46] "gcvEarth"            "GFS.FR.MOGAL"        "GFS.GCCL"
##  [49] "GFS.LT.RS"           "GFS.Thrift"          "glm"
##  [52] "glmboost"            "glmnet"              "glmStepAIC"
##  [55] "gpls"                "hda"                 "hdda"
##  [58] "HYFIS"               "icr"                 "J48"
##  [61] "JRip"                "kernelpls"           "kknn"
##  [64] "knn"                 "krlsPoly"            "krlsRadial"
##  [67] "lars"                "lars2"               "lasso"
##  [70] "lda"                 "lda2"                "leapBackward"
##  [73] "leapForward"         "leapSeq"             "Linda"
##  [76] "lm"                  "lmStepAIC"           "LMT"
##  [79] "logicBag"            "LogitBoost"          "logreg"
##  [82] "lssvmLinear"         "lssvmPoly"           "lssvmRadial"
##  [85] "lvq"                 "M5"                  "M5Rules"
##  [88] "mda"                 "Mlda"                "mlp"
##  [91] "mlpWeightDecay"      "multinom"            "nb"
##  [94] "neuralnet"           "nnet"                "nodeHarvest"
##  [97] "oblique.tree"        "OneR"                "ORFlog"
## [100] "ORFpls"              "ORFridge"            "ORFsvm"
## [103] "pam"                 "parRF"               "PART"
## [106] "partDSA"             "pcaNNet"             "pcr"
## [109] "pda"                 "pda2"                "penalized"
## [112] "PenalizedLDA"        "plr"                 "pls"
## [115] "plsRglm"             "ppr"                 "protoclass"
## [118] "qda"                 "QdaCov"              "qrf"
## [121] "qrnn"                "rbf"                 "rbfDDA"
## [124] "rda"                 "relaxo"              "rf"
## [127] "rFerns"              "RFlda"               "ridge"
## [130] "rknn"                "rknnBel"             "rlm"
## [133] "rocc"                "rpart"               "rpart2"
## [136] "rpartCost"           "RRF"                 "RRFglobal"
## [139] "rrlda"               "RSimca"              "rvmLinear"
## [142] "rvmPoly"             "rvmRadial"           "SBC"
## [145] "sda"                 "sddaLDA"             "sddaQDA"
## [148] "simpls"              "SLAVE"               "slda"
## [151] "smda"                "sparseLDA"           "spls"
## [154] "stepLDA"             "stepQDA"             "superpc"
## [157] "svmBoundrangeString" "svmExpoString"       "svmLinear"
## [160] "svmPoly"             "svmRadial"           "svmRadialCost"
## [163] "svmRadialWeights"    "svmSpectrumString"   "treebag"
## [166] "vbmpRadial"          "widekernelpls"       "WM"
## [169] "xyf"
```

Plenty to satisfy most needs!!

Gbm Modeling

It is important to know what type of modeling a particular model supports. This can be done using the **caret** function `getModelInfo`:

```
getModelInfo()$gbm$type

## [1] "Regression"     "Classification"
```

This tells us that `gbm` supports both **regression** and **classification**. As this is a binary classification, we need to force **gbm** into using the classification mode. We do this by changing the **outcome** variable to a factor (we use a copy of the outcome as we'll need the original one for our next model):

```
titanicDF$Survived2 <- ifelse(titanicDF$Survived==1,'yes','nope')

titanicDF$Survived2 <- as.factor(titanicDF$Survived2)

outcomeName <- 'Survived2'
```

As with most modeling projects, we need to split our data into two portions: a **training** and a **testing** portion. By doing this, we can use one portion to teach the model how to recognize survivors on the Titanic and the other portion to evaluate the model. Setting the **seed** is paramount for reproducibility as `createDataPartition` will shuffle the data randomly before splitting it. By using the same seed you will always get the same split in subsequent runs:

```
set.seed(1234)

splitIndex <- createDataPartition(titanicDF[,outcomeName], p = .75, list = FALSE, times = 1)

trainDF <- titanicDF[ splitIndex,]

testDF  <- titanicDF[-splitIndex,]
```

Caret offers many tuning functions to help you get as much as possible out of your models; the trainControl function allows you to control the resampling of your data. This will split the training data set internally and do its own train/test runs to figure out the best settings for your model. In this case, we're going to cross-validate the data 3 times, therefore training it 3 times on different portions of the data before settling on the best tuning parameters (for **gbm** it is `trees`, `shrinkage`, and `interaction depth`). You can also set these values yourself if you don't trust the function.

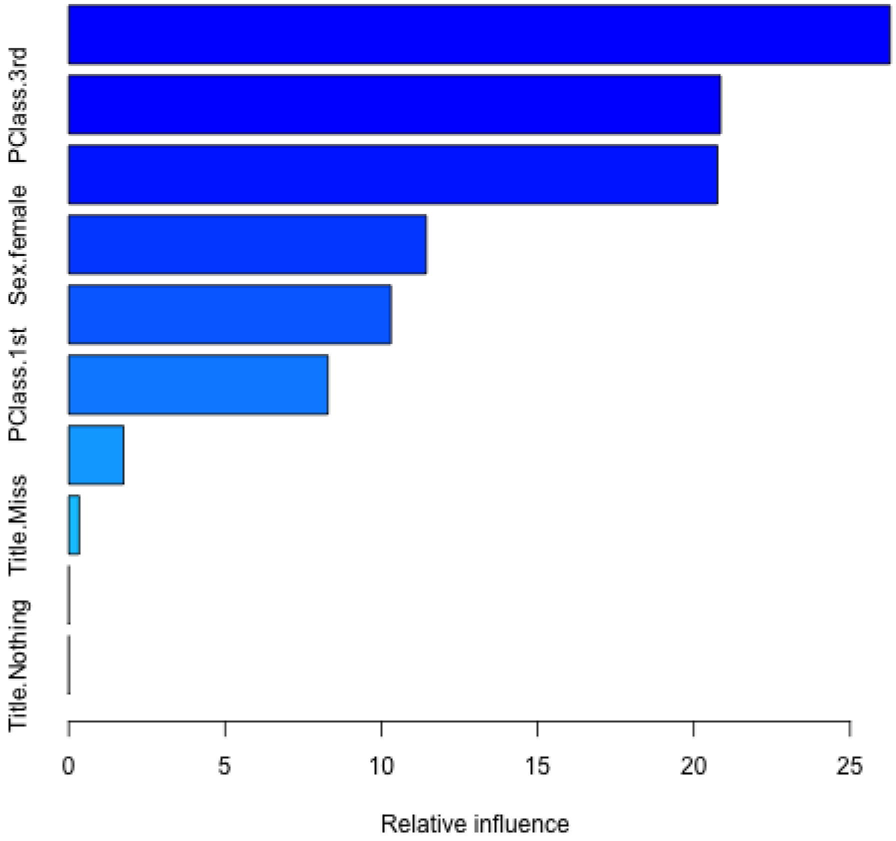
```
objControl <- trainControl(method='cv', number=3, returnResamp='none', summaryFunction = twoClassSummary, classProbs = TRUE)
```

This is the heart of our modeling adventure, time to teach our model how to recognize Titanic survivors. Because this is a classification model, we're requesting that our metrics useROC instead of the default **RMSE**:

```
objModel <- train(trainDF[,predictorsNames], trainDF[,outcomeName],

                  method='gbm',

                  trControl=objControl,

                  metric = "ROC",

                  preProc = c("center", "scale"))

## Iter   TrainDeviance   ValidDeviance   StepSize   Improve

##      1        1.2314             nan     0.1000    0.0245

##      2        1.1948             nan     0.1000    0.0192

##      3        1.1594             nan     0.1000    0.0158

...
```

I truncated most of the lines from the training process but you get the idea. We then can call `summary()` function on our model to find out what variables were most important:

```
summary(objModel)
```



```
##                          var rel.inf
## Title.Mr            Title.Mr 26.2756
## PClass.3rd        PClass.3rd 20.8523
## Sex.male           Sex.male 20.7569

## Sex.female       Sex.female 11.4357

## Age                     Age 10.3042

## PClass.1st       PClass.1st  8.2905

## Title.Mrs         Title.Mrs  1.7515

## Title.Miss       Title.Miss  0.3332

## PClass.2nd       PClass.2nd  0.0000

## Title.Nothing Title.Nothing  0.0000
```

We can find out what tuning parameters were most important to the model (notice the last lines about `trees`, `shrinkage` and `interaction depth`:

```
print(objModel)

## Stochastic Gradient Boosting

##

## 986 samples

##  10 predictor

##   2 classes: 'nope', 'yes'

##

## Pre-processing: centered, scaled

## Resampling: Cross-Validated (3 fold)

...

## Tuning parameter 'shrinkage' was held constant at a value of 0.1

## ROC was used to select the optimal model using  the largest value.

## The final values used for the model were n.trees = 100,

##  interaction.depth = 1 and shrinkage = 0.1.
```

Evaluate gbm model

There are two types of evaluation we can do here, `raw` or `prob`. **Raw** gives you a class prediction, in our case `yes` and `nope`, while **prob** gives you the probability on how sure the model is about its choice. I always use **prob**, as I like to be in control of the threshold and also like to use AUC score which requires probabilities, not classes. There are situations where having class values can come in handy, such as with multinomial models where you're predicting more than two values.

We now call the `predict` function and pass it our trained model and our testing data. Let's start by looking at class predictions and using the **caret** `postResample` function to get an accuracy score:

```
predictions <- predict(object=objModel, testDF[,predictorsNames], type='raw')
head(predictions)
## [1] yes  nope yes  nope nope nope
## Levels: nope yes
print(postResample(pred=predictions, obs=as.factor(testDF[,outcomeName])))
## Accuracy    Kappa
##   0.8135   0.5644
```

The accuracy tells us that our model is correct **81.35%** of the time - not bad...

Now let's look at probabilities:

```
# probabilites
library(pROC)
predictions <- predict(object=objModel, testDF[,predictorsNames], type='prob')
head(predictions)
##      nope    yes
## 1 0.07292 0.9271
## 2 0.76058 0.2394
## 3 0.43309 0.5669
## 4 0.67279 0.3272
## 5 0.67279 0.3272
## 6 0.54616 0.4538
```

To get the **AUC** score, you need to pass the `yes` column to the `roc` function (each row adds up to 1 but we're interested in the `yes`, the **survivors**):

```
auc <- roc(ifelse(testDF[,outcomeName]=="yes",1,0), predictions[[2]])
print(auc$auc)
## Area under the curve: 0.825
```

The **AUC** is telling us that our model has a **0.825 AUC** score (remember that an **AUC** ranges between **0.5** and **1**, where **0.5** is random and **1** is perfect).

Glmnet Modeling

Let's change gears and try this out on a regression model. Let's look at what modeling types **glmnet** supports and reset our outcome variable as we're going to be using the numerical outcome instead of the factor.

```
getModelInfo()$glmnet$type
## [1] "Regression"     "Classification"
outcomeName <- 'Survived'

set.seed(1234)
splitIndex <- createDataPartition(titanicDF[,outcomeName], p = .75, list = FALSE, times = 1)
trainDF <- titanicDF[ splitIndex,]
testDF  <- titanicDF[-splitIndex,]
```

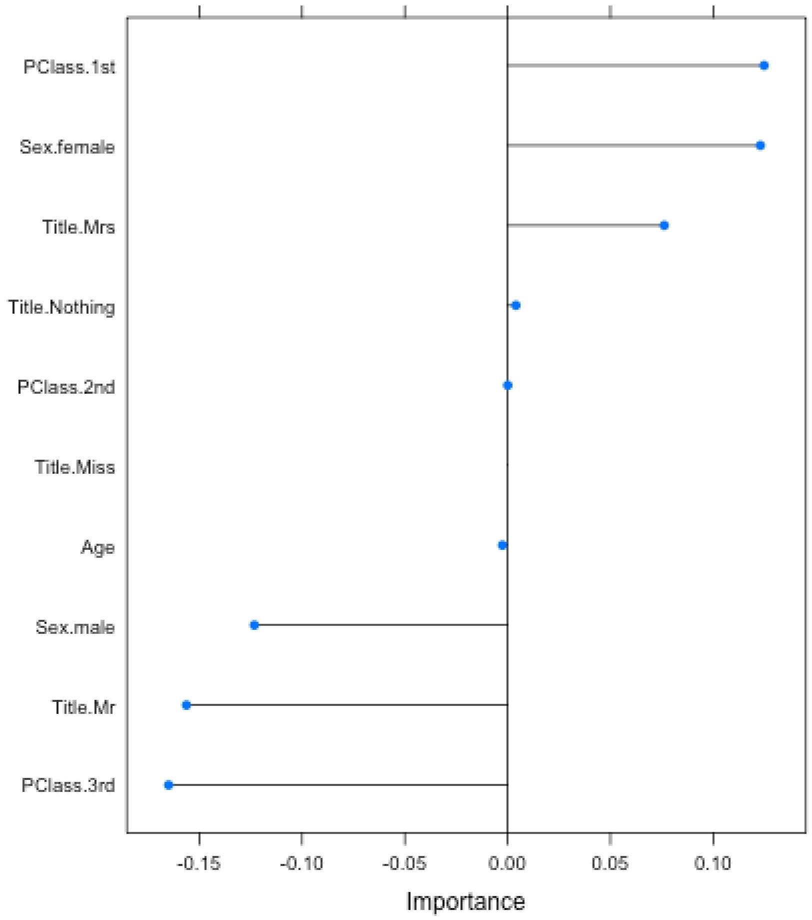
We re-run some of the basic training and prediction functions with some slight changes:

```
objControl <- trainControl(method='cv', number=3, returnResamp='none')

objModel <- train(trainDF[,predictorsNames], trainDF[,outcomeName], method='glmnet',  metric =
 "RMSE")

predictions <- predict(object=objModel, testDF[,predictorsNames])

auc <- roc(testDF[,outcomeName], predictions)

print(auc$auc)

## Area under the curve: 0.857
```

This is a stronger **AUC** score than our previous **gbm** model. Testing with different types of models does pay off (take it with a grain of salt as we didn't tune our models much).

You can also call the **caret** function `varImp` to figure out the variables that were important to the model. And this is one great feature of the **glmnet** model; it returns positive and negative variable importance unlike most models. This helps deepens your understanding about your variables, such that being in `PClass.1st` leans the probabilities in the survivor's favor while `PClass.3rd` does the opposite (make sure you set `scale` to False):

```
plot(varImp(objModel,scale=F))
```



Full source code (also on GitHub):

```
# load libraries
library(caret)
library(pROC)


#################################################
# data prep
#################################################


# load data
titanicDF <- read.csv('http://math.ucdenver.edu/RTutorial/titanic.txt',sep='\t')
titanicDF$Title <- ifelse(grepl('Mr ',titanicDF$Name),'Mr',ifelse(grepl('Mrs ',titanicDF$Name),'Mrs',ifelse(grepl('Miss',titanicDF$Name),'Miss','Nothing')))
titanicDF$Age[is.na(titanicDF$Age)] <- median(titanicDF$Age, na.rm=T)
```

```
# miso format
titanicDF <- titanicDF[c('PClass', 'Age',    'Sex',   'Title', 'Survived')]

# dummy variables for factors/characters
titanicDF$Title <- as.factor(titanicDF$Title)
titanicDummy <- dummyVars("~.",data=titanicDF, fullRank=F)
titanicDF <- as.data.frame(predict(titanicDummy,titanicDF))
print(names(titanicDF))


# what is the proportion of your outcome variable?
prop.table(table(titanicDF$Survived))

# save the outcome for the glmnet model
tempOutcome <- titanicDF$Survived

# generalize outcome and predictor variables
outcomeName <- 'Survived'
predictorsNames <- names(titanicDF)[names(titanicDF) != outcomeName]

#################################################
# model it
#################################################
# get names of all caret supported models
names(getModelInfo())

titanicDF$Survived <- ifelse(titanicDF$Survived==1,'yes','nope')

# pick model gbm and find out what type of model it is
getModelInfo()$gbm$type

# split data into training and testing chunks
set.seed(1234)
splitIndex <- createDataPartition(titanicDF[,outcomeName], p = .75, list = FALSE, times = 1)
trainDF <- titanicDF[ splitIndex,]
testDF  <- titanicDF[-splitIndex,]

# create caret trainControl object to control the number of cross-validations performed
```

```
objControl <- trainControl(method='cv', number=3, returnResamp='none', summaryFunction = twoClassSummary,
classProbs = TRUE)



# run model

objModel <- train(trainDF[,predictorsNames], as.factor(trainDF[,outcomeName]),

                  method='gbm',

                  trControl=objControl,

                  metric = "ROC",

                  preProc = c("center", "scale"))

)


# find out variable importance

summary(objModel)


# find out model details

objModel


#################################################

# evalute mdoel

#################################################

# get predictions on your testing data


# class prediction

predictions <- predict(object=objModel, testDF[,predictorsNames], type='raw')

head(predictions)

postResample(pred=predictions, obs=as.factor(testDF[,outcomeName]))


# probabilites

predictions <- predict(object=objModel, testDF[,predictorsNames], type='prob')

head(predictions)

postResample(pred=predictions, obs=testDF[,outcomeName])


auc <- roc(ifelse(testDF[,outcomeName]=="yes",1,0), predictions[[2]])

print(auc$auc)



################################################

# glmnet model
```

```
################################################

# pick model gbm and find out what type of model it is
getModelInfo()$glmnet$type

# save the outcome for the glmnet model
titanicDF$Survived  <- tempOutcome

# split data into training and testing chunks
set.seed(1234)
splitIndex <- createDataPartition(titanicDF[,outcomeName], p = .75, list = FALSE, times = 1)
trainDF <- titanicDF[ splitIndex,]
testDF  <- titanicDF[-splitIndex,]

# create caret trainControl object to control the number of cross-validations performed
objControl <- trainControl(method='cv', number=3, returnResamp='none')

# run model
objModel <- train(trainDF[,predictorsNames], trainDF[,outcomeName], method='glmnet',  metric = "RMSE")

# get predictions on your testing data
predictions <- predict(object=objModel, testDF[,predictorsNames])

library(pROC)
auc <- roc(testDF[,outcomeName], predictions)
print(auc$auc)

postResample(pred=predictions, obs=testDF[,outcomeName])

# find out variable importance
summary(objModel)
plot(varImp(objModel,scale=F))

# find out model details
objModel

# display variable importance on a +/- scale
vimp <- varImp(objModel, scale=F)
```

```
results <- data.frame(row.names(vimp$importance),vimp$importance$Overall)
results$VariableName <- rownames(vimp)
colnames(results) <- c('VariableName','Weight')
results <- results[order(results$Weight),]
results <- results[(results$Weight != 0),]

par(mar=c(5,15,4,2)) # increase y-axis margin.
xx <- barplot(results$Weight, width = 0.85,
              main = paste("Variable Importance -",outcomeName), horiz = T,
              xlab = "< (-) importance >  < neutral >  < importance (+) >", axes = FALSE,
              col = ifelse((results$Weight > 0), 'blue', 'red'))
axis(2, at=xx, labels=results$VariableName, tick=FALSE, las=2, line=-0.3, cex.axis=0.6)



################################################
# advanced stuff
################################################

# boosted tree model (gbm) adjust learning rate and and trees
gbmGrid <-  expand.grid(interaction.depth =  c(1, 5, 9),
                        n.trees = 50,
                        shrinkage = 0.01)

# run model
objModel <- train(trainDF[,predictorsNames], trainDF[,outcomeName], method='gbm', trControl=objControl, tu
neGrid = gbmGrid, verbose=F)

# get predictions on your testing data
predictions <- predict(object=objModel, testDF[,predictorsNames])

library(pROC)
auc <- roc(testDF[,outcomeName], predictions)
print(auc$auc)
```

Reducing High Dimensional Data with Principle Component Analysis (PCA) and prcomp

October 13, 2014 **Tags:** ***modeling***

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{stats}** - prcomp and PCA
- **{xgboost}** - fast modeling algorithm
- **{Metrics}** - measuring error & AUC
- **{caret}** - reducing zero/near-zero variance

I can't remember the last time I worked on a data set with less than **500** features. This isn't a big deal with today's computing power, but it can become unwieldy when you need to use certain forest-based models, heavy cross-validation, grid tuning, or any ensemble work. *Note: the term variables, features, predictors are used throughout and mean the same thing.*

The 3 common ways of dealing with **high-dimensionality data** (i.e. having too many variables) are:

1. get more computing muscle (like RStudio on an Amazon Web Services EC2 instance),
2. prune your data set using feature selection (measure variables effectiveness and keeps only the best - built-in feature selection - see fscaret),
3. and finally, the subject of this walkthrough, use **feature reduction** (also refereed as feature extraction) to create new variables made of bits and pieces of the original variables.

According to wikipedia:

> "Principal component analysis (PCA) is a statistical procedure that uses an orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables called principal components."

You'll find reams of explanations on the web, but, in a nutshell, **PCA** looks for the set of related variables in your data that explain most of the variance and creates a new feature out of it. This becomes your first component. It will then keep doing so on the next set of variables unrelated to the first, and that becomes your next component, and so on and so forth. This is done in an unsupervised manner so it doesn't care what your response variable/outcome is (but you should exclude it from your data before feeding it into **PCA**). *As a side note, this is the basis of a lot of compression software – it is that good.*

Let's code!

To get started, we need a data set with a lot of columns. We're going to borrow a data set from NIPS (Neural Information Processing Systems) from a completed, 2013 competition where the data is still available. The meaning of the data is immaterial for our needs. We download the GISETTE and associated labels data sets (**warning:** this is a large file):

```
temp <- tempfile()

download.file("http://www.nipsfsc.ecs.soton.ac.uk/datasets/GISETTE.zip",temp, mode="wb")

unzip(temp, "GISETTE/gisette_train.data")

gisetteRaw <- read.table("GISETTE/gisette_train.data", sep=" ",skip=0, header=F)

unzip(temp, "GISETTE/gisette_train.labels")

g_labels <- read.table("GISETTE/gisette_train.labels", sep=" ",skip=0, header=F)


print(dim(gisetteRaw))

## [1] 6000 5001
```

The **gisetteRaw** data frame has **5001** columns and that's the kind of size we're looking for. Before we can start the **PCA** transformation process, we need to remove the extreme near-zero variance as it won't help us much and risks crashing the script. We load the **caret** package and call `nearZeroVar` function with `saveMetrics` parameter set to **true**. This will return a data frame with the degree of zero variance for each feature:

```
library(caret)

nzv <- nearZeroVar(gisetteRaw, saveMetrics = TRUE)
```

```
print(paste('Range:',range(nzv$percentUnique)))
## [1] "Range: 0"   "Range: 8.6"
print(head(nzv))
##    freqRatio percentUnique zeroVar  nzv
## V1     48.25        5.2167   FALSE TRUE
## V2   1180.80        1.3667   FALSE TRUE
## V3     41.32        6.1500   FALSE TRUE
## V4   5991.00        0.1667   FALSE TRUE
## V5    980.00        1.5333   FALSE TRUE
## V6    140.00        3.5167   FALSE TRUE
```

We remove features with less than 0.1% variance:

```
print(paste('Column count before cutoff:',ncol(gisetteRaw)))
## [1] "Column count before cutoff: 5001"
dim(nzv[nzv$percentUnique > 0.1,])
## [1] 4639    4
gisette_nzv <- gisetteRaw[c(rownames(nzv[nzv$percentUnique > 0.1,])) ]
print(paste('Column count after cutoff:',ncol(gisette_nzv)))
## [1] "Column count before cutoff: 4639"
```

The data is cleaned up and ready to go. Let's see how well it performs without any **PCA** transformation. We bind the labels (response/outcome variables) to the set:

```
dfEvaluate <- cbind(as.data.frame(sapply(gisette_nzv, as.numeric)),
              cluster=g_labels$V1)
```

We're going to feed the data into the following cross-validation function using the `zxgboost` model. This is a fast model and does great with large data sets. The repeated cross-validation will run the data 5 times, each time assigning a new chunk of data as training and testing. This not only allows us to use all the data as both train and test sets, but also stabilizes our AUC (Area Under the Curve) score.

```
EvaluateAUC <- function(dfEvaluate) {
        require(xgboost)
        require(Metrics)
        CVs <- 5
        cvDivider <- floor(nrow(dfEvaluate) / (CVs+1))
        indexCount <- 1
        outcomeName <- c('cluster')
        predictors <- names(dfEvaluate)[!names(dfEvaluate) %in% outcomeName]
        lsErr <- c()
        lsAUC <- c()
        for (cv in seq(1:CVs)) {
                print(paste('cv',cv))
                dataTestIndex <- c((cv * cvDivider):(cv * cvDivider + cvDivider))
                dataTest <- dfEvaluate[dataTestIndex,]
                dataTrain <- dfEvaluate[-dataTestIndex,]

                bst <- xgboost(data = as.matrix(dataTrain[,predictors]),
                               label = dataTrain[,outcomeName],
                               max.depth=6, eta = 1, verbose=0,
                               nround=5, nthread=4,
                               objective = "reg:linear")

                predictions <- predict(bst, as.matrix(dataTest[,predictors]), outputmargin=TRUE)
                err <- rmse(dataTest[,outcomeName], predictions)
                auc <- auc(dataTest[,outcomeName],predictions)

                lsErr <- c(lsErr, err)
                lsAUC <- c(lsAUC, auc)
                gc()
        }
        print(paste('Mean Error:',mean(lsErr)))
        print(paste('Mean AUC:',mean(lsAUC)))
}
```

```
EvaluateAUC(dfEvaluate)


## [1] "cv 1"

## [1] "cv 2"

## [1] "cv 3"

## [1] "cv 4"

## [1] "cv 5"


## [1] 0.9659
```

This yields a great **AUC score of 0.9659** (remember, AUC of 0.5 is random, and 1.0 is perfect). But we don't really care how well the model did; we just want to use that AUC score as a basis of comparison against the transformed PCA variables.

So, let's use the same data and run it through `prcomp`. This will transform all the related variables that account for most of the variation - meaning that the first component variable will be the most powerful variable (**Warning:** this can be a very slow to process depending on your machine - it took 20 minutes on my MacBook - so do it once and store the resulting data set for later use):

```
pmatrix <- scale(gisette_nzv)

princ <- prcomp(pmatrix)
```

Let's start by running the same cross-validation code with just the **first PCA component** (remember, this holds most of the variation of our data set). We need to use our princ result set and call the `predict` function to get our data.frame:

```
nComp <- 1

dfComponents <- predict(princ, newdata=pmatrix)[,1:nComp]


dfEvaluate <- cbind(as.data.frame(dfComponents),

              cluster=g_labels$V1)


EvaluateAUC(dfEvaluate)
```

```
## [1] "cv 1"

## [1] "cv 2"

## [1] "cv 3"

## [1] "cv 4"

## [1] "cv 5"


## [1] 0.719
```

The resulting **AUC of 0.719** isn't that good compared to the orginal, non-transformed data set. But we have to remember that this is one variable against almost 5000!! Let's try this again with 2 components:

```
nComp <- 2

...

print(mean(lsAUC))

## [1] 0.7228
```

Two components give an **AUC score of 0.7228**, still some ways to go. Let's jump to **5** components:

```
nComp <- 5

...

print(mean(lsAUC))

## [1] 0.9279
```

Now we're talking, **0.9279**!!! Let's try **10** components:

```
nComp <- 10

...

print(mean(lsAUC))

## [1] 0.9651
```

Yowza!! **0.9651**!! Let's try **20** components:

```
nComp <- 20

...

print(mean(lsAUC))

## [1] 0.9641
```

Hmmm, going back down... Let's stop here and stick with the first **10 PCA** components. So, 10 PCA columns versus 4639 columns - not bad, right? Keep in mind that you should be able to get closer to the **AUC** of the original data set by adding more **PCA** components as `prcomp` accounts for all variations in the data. On the other hand, by following the steps in this walkthrough, you can get a great **AUC** score with very little effort and an absurdly smaller resulting data set.

Additional Stuff

A common critique about **PCA** is that it is hard to analyze once transformed as many of variables get clumped together under a nondescript name. One way around this is top plot your **PCA** data on top of you discrete variables, see FactoMineR for more information.

Though out of scope for this hands-on post, there are many ways of finding the perfect amount of components to use - check out Eigen angles and vectors and check out also clusterboot.

Full source code (also on GitHub):

```
require(ROCR)
require(caret)
require(ggplot2)

EvaluateAUC <- function(dfEvaluate) {
        require(xgboost)
        require(Metrics)
        CVs <- 5
        cvDivider <- floor(nrow(dfEvaluate) / (CVs+1))
        indexCount <- 1
        outcomeName <- c('cluster')
        predictors <- names(dfEvaluate)[!names(dfEvaluate) %in% outcomeName]
        lsErr <- c()
        lsAUC <- c()
        for (cv in seq(1:CVs)) {
                print(paste('cv',cv))
                dataTestIndex <- c((cv * cvDivider):(cv * cvDivider + cvDivider))
                dataTest <- dfEvaluate[dataTestIndex,]
                dataTrain <- dfEvaluate[-dataTestIndex,]

                bst <- xgboost(data = as.matrix(dataTrain[,predictors]),
                               label = dataTrain[,outcomeName],
                               max.depth=6, eta = 1, verbose=0,
                               nround=5, nthread=4,
                               objective = "reg:linear")

                predictions <- predict(bst, as.matrix(dataTest[,predictors]), outputmargin=TRUE)
                err <- rmse(dataTest[,outcomeName], predictions)
                auc <- auc(dataTest[,outcomeName],predictions)

                lsErr <- c(lsErr, err)
                lsAUC <- c(lsAUC, auc)
                gc()
        }
        print(paste('Mean Error:',mean(lsErr)))
        print(paste('Mean AUC:',mean(lsAUC)))
}
```

```
##########################################################################################
## Download data
##########################################################################################

# http://www.nipsfsc.ecs.soton.ac.uk/datasets/GISETTE.zip
# http://stat.ethz.ch/R-manual/R-devel/library/stats/html/princomp.html
temp <- tempfile()

# word of warning, this is 20mb - slow
download.file("http://www.nipsfsc.ecs.soton.ac.uk/datasets/GISETTE.zip",temp, mode="wb")
dir(tempdir())

unzip(temp, "GISETTE/gisette_train.data")
gisetteRaw <- read.table("GISETTE/gisette_train.data", sep=" ",skip=0, header=F)
unzip(temp, "GISETTE/gisette_train.labels")
g_labels <- read.table("GISETTE/gisette_train.labels", sep=" ",skip=0, header=F)

##########################################################################################
## Remove zero and close to zero variance
##########################################################################################

nzv <- nearZeroVar(gisetteRaw, saveMetrics = TRUE)
range(nzv$percentUnique)

# how many have no variation at all
print(length(nzv[nzv$zeroVar==T,]))

print(paste('Column count before cutoff:',ncol(gisetteRaw)))

# how many have less than 0.1 percent variance
dim(nzv[nzv$percentUnique > 0.1,])

# remove zero & near-zero variance from original data set
gisette_nzv <- gisetteRaw[c(rownames(nzv[nzv$percentUnique > 0.1,])) ]
print(paste('Column count after cutoff:',ncol(gisette_nzv)))

##########################################################################################
# Run model on original data set
```

```
##########################################################################################

dfEvaluate <- cbind(as.data.frame(sapply(gisette_nzv, as.numeric)),
                    cluster=g_labels$V1)

EvaluateAUC(dfEvaluate)

##########################################################################################
# Run prcomp on the data set
##########################################################################################

pmatrix <- scale(gisette_nzv)
princ <- prcomp(pmatrix)

# plot the first two components
ggplot(dfEvaluate, aes(x=PC1, y=PC2, colour=as.factor(g_labels$V1+1))) +
        geom_point(aes(shape=as.factor(g_labels$V1))) + scale_colour_hue()

# full - 0.965910574495451
nComp <- 5
nComp <- 10
nComp <- 90
nComp <- 20
nComp <- 50
nComp <- 100

# change nComp to try different numbers of component variables (10 works great)
nComp <- 10  # 0.9650
dfComponents <- predict(princ, newdata=pmatrix)[,1:nComp]
dfEvaluate <- cbind(as.data.frame(dfComponents),
                    cluster=g_labels$V1)

EvaluateAUC(dfEvaluate)
```

The Sparse Matrix and {glmnet}

October 8, 2014 **Tags:** ***modeling***

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{Matrix}** - creates sparse/dense matrices
- **{glmnet}** - generalized linear models
- **{pROC}** - ROC tools

In this walkthough, I am going to show how sparse matrices work in R and how to use them with the GLMNET package.

For those that aren't familiar with sparse matrices, or the sparse matrix, as the name implies, it is a large but ideally hollow data set. If your data contains lots of zeros then a sparse matrix is a very memory-efficient way of holding that data. For example, if you have a lot of dummy variables, most of that data will be zeros, and a sparse matrix will only hold non-zero data and ignore the zeros, thus using a lot less memory and allowing the memorization of much larger data sets than traditional data frames.

Wikipedia has great write-up on the sparse matrix and related theories if you want to dive into this in more details.

Unfortunately the sparse matrix in R doesn't accept **NAs**, **NaNs** and **Infinites**… Also, normalization functions, such as centering or scaling, could affect the zero values and render the data set into a non-sparse matrix and defeating any memory-efficient advantages.

Let's start with a simple data set called `some_dataframe`:

```
print(some_dataframe)

##    c1 c2 c3 c4 c5 c6 c7 c8 c9 c10 outcome

## 1   2  7  0  0  0  0  0  0  0   0       0

## 2   0  0  3  0  0  0  0  0  0   0       0
## 3   0  0  0  6  1  0  0  0  0   0       0
## 4   0  0  0  2  0  0  0  0  0   0       0
## 5   0  0  0  0  0  0  0  0 12   0       1
## 6   0  0  0  0  0 25  0  0  0   0       1
## 7   1  0  0  0  2  0  0  0  0   0       0
## 8   0  0  0  2  0  0  0  0  0   0       0
## 9   0  0  0  0  0  0  0  0 14   0       1
## 10  0  0  0  0  0 21  0  0  0   0       1
## 11  0  0  0  0  0  0 28  0  0   0       1
## 12  0  0  0  0  0  0  0 35  0   0       1
## 13  0  0  0  0  0  0  0  0 42   0       1
## 14  0  0  0  0  0  0  0  0  0  49       1
```

To see the above **data.frame** in a **matrix** format simply requires casting it to a matrix:

```
some_matrix <- data.matrix(some_dataframe[1:10])
print(some_matrix)
##       c1 c2 c3 c4 c5 c6 c7 c8 c9 c10
##  [1,]  2  7  0  0  0  0  0  0  0   0
##  [2,]  0  0  3  0  0  0  0  0  0   0
##  [3,]  0  0  0  6  1  0  0  0  0   0
##  [4,]  0  0  0  2  0  0  0  0  0   0
##  [5,]  0  0  0  0  0  0  0  0 12   0
##  [6,]  0  0  0  0  0 25  0  0  0   0
##  [7,]  1  0  0  0  2  0  0  0  0   0
##  [8,]  0  0  0  2  0  0  0  0  0   0
##  [9,]  0  0  0  0  0  0  0  0 14   0
## [10,]  0  0  0  0  0 21  0  0  0   0
## [11,]  0  0  0  0  0  0 28  0  0   0
## [12,]  0  0  0  0  0  0  0 35  0   0

## [13,]  0  0  0  0  0  0  0  0 42   0

## [14,]  0  0  0  0  0  0  0  0  0  49
```

Visually, it isn't much different than the data frame (internally, a matrix is restricted to one data type only). In order to transform it into a sparse matrix, we load the **Matrix library**, call the `Matrix` function with the `sparse` flag set to true:

```
library(Matrix)

print(Matrix(some_matrix, sparse=TRUE))

## 14 x 10 sparse Matrix of class "dgCMatrix"

##    [[ suppressing 10 column names 'c1', 'c2', 'c3' ... ]]

##

##  [1,] 2 7 . . .  .  .  .  .  .

##  [2,] . . 3 . .  .  .  .  .  .

##  [3,] . . . 6 1  .  .  .  .  .

##  [4,] . . . 2 .  .  .  .  .  .

##  [5,] . . . . .  .  .  . 12  .

##  [6,] . . . . . 25  .  .  .  .

##  [7,] 1 . . . 2  .  .  .  .  .

##  [8,] . . . 2 .  .  .  .  .  .

##  [9,] . . . . .  .  .  . 14  .

## [10,] . . . . . 21  .  .  .  .

## [11,] . . . . .  . 28  .  .  .

## [12,] . . . . .  .  . 35  .  .

## [13,] . . . . .  .  .  . 42  .

## [14,] . . . . .  .  .  .  . 49
```

And here we finally get a sense of its efficiency, it only retains the non-zero values!

GLMNET package

The help files describes the `GLMNET` package as a package containing 'extremely efficient procedures for fitting lasso or elastic-net regularization for linear regression, logistic and multinomial regression models, poisson regression and the Cox model and more (from the help files).

Unfortunately in R, few models support sparse matrices besides **GLMNET** (that I know of) therefore in conversations about modeling with R, when the subject of sparse matrices comes up, it is usually followed by the `glmnet` model.

Let's start by splitting our `some_dataframe` in two parts: a 2/3 portion that will become our training data set and a 1/3 portion for our testing data set (always set the `seed` so random draws are reproducible):

```
set.seed(2)

split <- sample(nrow(some_dataframe), floor(0.7*nrow(some_dataframe)))

train <-some_dataframe[split,]

test <- some_dataframe[-split,]
```

We then construct a sparse model matrix using the typical **R** formula:

```
library(glmnet)

train_sparse <- sparse.model.matrix(~.,train[1:10])

test_sparse <- sparse.model.matrix(~.,test[1:10])

print(train_sparse)

## 9 x 11 sparse Matrix of class "dgCMatrix"

##    [[ suppressing 11 column names '(Intercept)', 'c1', 'c2' ... ]]

##

## 3  1 . . . 6 1  .  . .  .  .

## 10 1 . . . . . 21  . .  .  .

## 7  1 1 . . . 2  .  . .  .  .

## 2  1 . . 3 . .  .  . .  .  .

## 13 1 . . . . .  .  . . 42  .

## 9  1 . . . . .  .  . . 14  .
## 11 1 . . . . .  . 28 .  .  .
## 6  1 . . . . . 25  . .  .  .
## 14 1 . . . . .  .  . .  . 49
print(test_sparse)
## 5 x 11 sparse Matrix of class "dgCMatrix"
##    [[ suppressing 11 column names '(Intercept)', 'c1', 'c2' ... ]]
##
## 1  1 2 7 . . . . .  .  . .
## 4  1 . . . 2 . . .  .  . .
## 5  1 . . . . . . .  . 12 .
## 8  1 . . . 2 . . .  .  . .
## 12 1 . . . . . . . 35  . .
```

We call the `glmnet` model and get our fit:

```
fit <- glmnet(train_sparse,train[,11])
```

And call the `predict` function to get our probabilities:

```
pred <- predict(fit, test_sparse, type="class")
print(head(pred[,1:5]))
##        s0     s1     s2     s3    s4
## 1  0.6667 0.6742 0.6815 0.6884 0.695
## 4  0.6667 0.6742 0.6815 0.6884 0.695
## 5  0.6667 0.6742 0.6815 0.6884 0.695
## 8  0.6667 0.6742 0.6815 0.6884 0.695
## 12 0.6667 0.6742 0.6815 0.6884 0.695
```

The reason it returns many probability sets is because `glmnet` fits the model for different regularization parameters at the same time. To help us choose the best prediction set, we can use the function `cv.glmnet`. This will use cross validation to find the fit with the smallest error. Let's call cv.glmnet and pass the results to the `s` paramter (the prenalty parameter) of the `predict` function:

```
# use cv.glmnet to find best lambda/penalty - choosing small nfolds for cv due to…

# s is the penalty parameter

cv <- cv.glmnet(train_sparse,train[,11],nfolds=3)

pred <- predict(fit, test_sparse,type="response", s=cv$lambda.min)
```

NOTE: the `cv.glmnet` returns various values that may be important for your modeling needs. In particular `lambda.min` and `lambda.1se`. One is the smallest error and the other is the simplest error. Refer to the help files to find the best one for your needs.

```
print(names(cv))

##  [1] "lambda"     "cvm"        "cvsd"       "cvup"       "cvlo"

##  [6] "nzero"      "name"       "glmnet.fit" "lambda.min" "lambda.1se"
```

As the data is made up, let's not read too deeply into these predictions:

```
print(pred)

##          1

## 1  0.9898

## 4  0.8306

## 5  0.9898

## 8  0.8306

## 12 0.9898
```

Let's see how the `sparse.model.matrix` function of the **Matrix** package handles discreet values. I added a factor variable `mood` with two levels: `happy` and `sad`:

```
print(cat_dataframe)

##    c1 c2 c3 c4 c5 c6 c7 c8 c9 c10  mood outcome

## 1   2  7  0  0  0  0  0  0  0   0 happy       0

## 2   0  0  3  0  0  0  0  0  0   0 happy       0

## 3   0  0  0  6  1  0  0  0  0   0 happy       0

## 4   0  0  0  2  0  0  0  0  0   0 happy       0

## 5   0  0  0  0  0  0  0  0 12   0   sad       1

## 6   0  0  0  0  0 25  0  0  0   0   sad       1

## 7   1  0  0  0  2  0  0  0  0   0 happy       0

## 8   0  0  0  2  0  0  0  0  0   0 happy       0

## 9   0  0  0  0  0  0  0  0 14   0   sad       1

## 10  0  0  0  0  0 21  0  0  0   0   sad       1

## 11  0  0  0  0  0  0 28  0  0   0   sad       1

## 12  0  0  0  0  0  0  0 35  0   0   sad       1

## 13  0  0  0  0  0  0  0  0 42   0   sad       1

## 14  0  0  0  0  0  0  0  0  0  49   sad       1
```

We call the `sparse.model.matrix` function and we notice that it turned `happy` into 0's and `sad` into 1's. Thus, we only see the 1's:

```
sparse.model.matrix(~.,cat_dataframe)

## 14 x 13 sparse Matrix of class "dgCMatrix"

##    [[ suppressing 13 column names '(Intercept)', 'c1', 'c2' ... ]]

##

## 1  1 2 7 . . .  .  .  .  .  . . .

## 2  1 . . 3 . .  .  .  .  .  . . .

## 3  1 . . . 6 1  .  .  .  .  . . .

## 4  1 . . . 2 .  .  .  .  .  . . .
## 5  1 . . . . .  .  .  . 12  . 1 1
## 6  1 . . . . . 25  .  .  .  . 1 1
## 7  1 1 . . . 2  .  .  .  .  . . .
## 8  1 . . . 2 .  .  .  .  .  . . .
## 9  1 . . . . .  .  .  . 14  . 1 1
## 10 1 . . . . . 21  .  .  .  . 1 1
## 11 1 . . . . .  . 28  .  .  . 1 1
## 12 1 . . . . .  .  . 35  .  . 1 1
## 13 1 . . . . .  .  .  . 42  . 1 1
## 14 1 . . . . .  .  .  .  . 49 1 1
```

Let's complicate things by adding a few more levels to our factor:

```
print(cat_dataframe)
##    c1 c2 c3 c4 c5 c6 c7 c8 c9 c10    mood outcome
## 1   2  7  0  0  0  0  0  0  0   0   angry       0
## 2   0  0  3  0  0  0  0  0  0   0 neutral       0
## 3   0  0  0  6  1  0  0  0  0   0   happy       0
## 4   0  0  0  2  0  0  0  0  0   0   happy       0
## 5   0  0  0  0  0  0  0  0 12   0     sad       1
## 6   0  0  0  0  0 25  0  0  0   0     sad       1
## 7   1  0  0  0  2  0  0  0  0   0   happy       0
## 8   0  0  0  2  0  0  0  0  0   0   happy       0
## 9   0  0  0  0  0  0  0  0 14   0     sad       1
## 10  0  0  0  0  0 21  0  0  0   0 neutral       1
## 11  0  0  0  0  0  0 28  0  0   0     sad       1
## 12  0  0  0  0  0  0  0 35  0   0     sad       1
## 13  0  0  0  0  0  0  0  0 42   0     sad       1
## 14  0  0  0  0  0  0  0  0  0  49     sad       1
```

```
print(levels(cat_dataframe$mood))

## [1] "angry"   "happy"   "neutral" "sad"
```

We can compare the dimensions of both data sets:

```
dim(cat_dataframe)

## [1] 14 12

dim(sparse.model.matrix(~.,cat_dataframe))

## [1] 14 15
```

The sparse model has 3 extra columns. It broke out the 4 levels into 3. This is because it applied `Full Rank` to the set - you're either one of the 3 moods, if you're neither of the 3, then you're assumed to be the 4th or `angry`. Check out my walkthrough on dummy variables for more details:

```
print(sparse.model.matrix(~.,cat_dataframe))

## 14 x 15 sparse Matrix of class "dgCMatrix"

##    [[ suppressing 15 column names '(Intercept)', 'c1', 'c2' ... ]]

##

## 1  1 2 7 . . .  .  .  .  .  . . . . .

## 2  1 . . 3 . .  .  .  .  .  . 1 . .

## 3  1 . . . 6 1  .  .  .  .  . 1 . . .

## 4  1 . . . 2 .  .  .  .  .  . 1 . . .

## 5  1 . . . . . .  .  .  . 12  . . . 1 1

## 6  1 . . . . . 25  .  .  .  . . . 1 1

## 7  1 1 . . . 2  .  .  .  .  . 1 . . .

## 8  1 . . . 2 .  .  .  .  .  . 1 . . .

## 9  1 . . . . . .  .  .  . 14  . . . 1 1

## 10 1 . . . . . 21  .  .  .  . . 1 . 1

## 11 1 . . . . .  . 28  .  .  . . . 1 1
## 12 1 . . . . .  .  . 35  .  . . . 1 1

## 13 1 . . . . .  .  .  . 42  . . . 1 1

## 14 1 . . . . .  .  .  .  . 49 . . 1 1
```

Full source code (also on GitHub):

```
some_dataframe <- read.table(text="c1       c2      c3      c4      c5      c6      c7      c8      c9      c10     outcome

2     7     0     0     0     0     0     0     0     0     0

0     0     3     0     0     0     0     0     0     0     0

0     0     0     6     1     0     0     0     0     0     0

0     0     0     2     0     0     0     0     0     0     0

0     0     0     0     0     0     0     0     12     0     1

0     0     0     0     0     25     0     0     0     0     1

1     0     0     0     2     0     0     0     0     0     0

0     0     0     2     0     0     0     0     0     0     0

0     0     0     0     0     0     0     0     14     0     1

0     0     0     0     0     21     0     0     0     0     1

0     0     0     0     0     0     28     0     0     0     1

0     0     0     0     0     0     0     35     0     0     1

0     0     0     0     0     0     0     0     42     0     1

0     0     0     0     0     0     0     0     0     49     1", header=T, sep="")


library(Matrix)

some_matrix <- data.matrix(some_dataframe[1:10])


# show matrix representation of data set

Matrix(some_matrix, sparse=TRUE)


# split data set into a train and test portion

set.seed(2)

split <- sample(nrow(some_dataframe), floor(0.7*nrow(some_dataframe)))

train <-some_dataframe[split,]

test <- some_dataframe[-split,]
```

```
# transform both sets into sparse matrices using the sparse.model.matrix
train_sparse <- sparse.model.matrix(~.,train[1:10])
test_sparse <- sparse.model.matrix(~.,test[1:10])

# model the sparse sets using glmnet
library(glmnet)
fit <- glmnet(train_sparse,train[,11])

# use cv.glmnet to find best lambda/penalty
# s is the penalty parameter
cv <- cv.glmnet(train_sparse,train[,11],nfolds=3)
pred <- predict(fit, test_sparse,type="response", s=cv$lambda.min)

#  receiver operating characteristic (ROC curves)
library(pROC)
auc = roc(test[,11], pred)
print(auc$auc)

# how does sparse deal with categorical data (adding mood feature with two levels)?
cat_dataframe<- read.table(text="c1     c2     c3     c4     c5     c6     c7     c8     c9     c10     mo
od     outcome
2     7     0     0     0     0     0     0     0     0     happy     0
0     0     3     0     0     0     0     0     0     0     happy     0
0     0     0     6     1     0     0     0     0     0     happy     0
0     0     0     2     0     0     0     0     0     0     happy     0
0     0     0     0     0     0     0     0     12     0     sad     1
0     0     0     0     0     25     0     0     0     0     sad     1
1     0     0     0     2     0     0     0     0     0     happy     0
0     0     0     2     0     0     0     0     0     0     happy     0
0     0     0     0     0     0     0     0     14     0     sad     1
0     0     0     0     0     21     0     0     0     0     sad     1
0     0     0     0     0     0     28     0     0     0     sad     1
0     0     0     0     0     0     0     35     0     0     sad     1
0     0     0     0     0     0     0     0     42     0     sad     1
0     0     0     0     0     0     0     0     0     49     sad     1", header=T, sep="")
print(sparse.model.matrix(~.,cat_dataframe))

# increasing the number of levels in the mood variable)
```

```
cat_dataframe <- read.table(text="c1     c2     c3     c4     c5     c6     c7     c8     c9     c10     m
ood     outcome

2     7     0     0     0     0     0     0     0     0     angry     0

0     0     3     0     0     0     0     0     0     0     neutral     0

0     0     0     6     1     0     0     0     0     0     happy     0

0     0     0     2     0     0     0     0     0     0     happy     0

0     0     0     0     0     0     0     0     12     0     sad     1

0     0     0     0     0     25     0     0     0     0     sad     1

1     0     0     0     2     0     0     0     0     0     happy     0

0     0     0     2     0     0     0     0     0     0     happy     0

0     0     0     0     0     0     0     0     14     0     sad     1

0     0     0     0     0     21     0     0     0     0     neutral     1

0     0     0     0     0     0     28     0     0     0     sad     1

0     0     0     0     0     0     0     35     0     0     sad     1

0     0     0     0     0     0     0     0     42     0     sad     1

0     0     0     0     0     0     0     0     0     49     sad     1", header=T, sep="")

print(levels(cat_dataframe$mood))

dim(cat_dataframe)

# sparse added extra columns when in binarized mood

dim(sparse.model.matrix(~.,cat_dataframe))
```

Modeling Ensembles with R and {caret}

October 18, 2014 **Tags:** ***modeling***

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{caret}** - modeling wrapper, functions, commands
- **{RCurl}** - web data downloading functions
- **{pROC}** - Area Under the Curve (AUC) functions

There are many reasons to ensemble models but it usually comes down to capturing a deeper understanding of high dimensionality data. The more complex a data set, the more it benefits from additional models, just like additional eyes, to capture more nuances scattered around high dimensionality data.

Let's code!

This walkthrough leverages the **caret** package for ease of coding but the concept applies to any model in any statistical programming language. **Caret** allows you to easily switch models in a script without having to change much of the code. You can easily write a loop and have it run through the almost 170 models that the package currently supports (Max Kuhn keeps adding new ones) by only changing one variable.

To get a complete list of the models supported by **caret** use the `getModelInfo` function:

```
library(caret)
names(getModelInfo())
##   [1] "ada"                 "ANFIS"               "avNNet"
##   [4] "bag"                 "bagEarth"            "bagFDA"
##   [7] "bayesglm"            "bdk"                 "blackboost"
##  [10] "Boruta"              "brnn"                "bstLs"
##  [13] "bstSm"               "bstTree"             "C5.0"
##  [16] "C5.0Cost"            "C5.0Rules"           "C5.0Tree"
##  [19] "cforest"             "CSimca"              "ctree"
##  [22] "ctree2"              "cubist"              "DENFIS"
##  [25] "dnn"                 "earth"               "elm"
##  [28] "enet"                "evtree"              "extraTrees"
##  [31] "fda"                 "FH.GBML"             "FIR.DM"
##  [34] "foba"                "FRBCS.CHI"           "FRBCS.W"
##  [37] "FS.HGD"              "gam"                 "gamboost"
##  [40] "gamLoess"            "gamSpline"           "gaussprLinear"
##  [43] "gaussprPoly"         "gaussprRadial"       "gbm"
##  [46] "gcvEarth"            "GFS.FR.MOGAL"        "GFS.GCCL"
##  [49] "GFS.LT.RS"           "GFS.Thrift"          "glm"
##  [52] "glmboost"            "glmnet"              "glmStepAIC"
##  [55] "gpls"                "hda"                 "hdda"
##  [58] "HYFIS"               "icr"                 "J48"
##  [61] "JRip"                "kernelpls"           "kknn"
##  [64] "knn"                 "krlsPoly"            "krlsRadial"
##  [67] "lars"                "lars2"               "lasso"
##  [70] "lda"                 "lda2"                "leapBackward"
##  [73] "leapForward"         "leapSeq"             "Linda"
##  [76] "lm"                  "lmStepAIC"           "LMT"
##  [79] "logicBag"            "LogitBoost"          "logreg"
##  [82] "lssvmLinear"         "lssvmPoly"           "lssvmRadial"
##  [85] "lvq"                 "M5"                  "M5Rules"
##  [88] "mda"                 "Mlda"                "mlp"
##  [91] "mlpWeightDecay"      "multinom"            "nb"
##  [94] "neuralnet"           "nnet"                "nodeHarvest"
##  [97] "oblique.tree"        "OneR"                "ORFlog"
## [100] "ORFpls"              "ORFridge"            "ORFsvm"
## [103] "pam"                 "parRF"               "PART"
## [106] "partDSA"             "pcaNNet"             "pcr"
## [109] "pda"                 "pda2"                "penalized"
## [112] "PenalizedLDA"        "plr"                 "pls"
## [115] "plsRglm"             "ppr"                 "protoclass"
## [118] "qda"                 "QdaCov"              "qrf"
## [121] "qrnn"                "rbf"                 "rbfDDA"
## [124] "rda"                 "relaxo"              "rf"
## [127] "rFerns"              "RFlda"               "ridge"
## [130] "rknn"                "rknnBel"             "rlm"
## [133] "rocc"                "rpart"               "rpart2"
## [136] "rpartCost"           "RRF"                 "RRFglobal"
## [139] "rrlda"               "RSimca"              "rvmLinear"
## [142] "rvmPoly"             "rvmRadial"           "SBC"
## [145] "sda"                 "sddaLDA"             "sddaQDA"
## [148] "simpls"              "SLAVE"               "slda"
## [151] "smda"                "sparseLDA"           "spls"
## [154] "stepLDA"             "stepQDA"             "superpc"
## [157] "svmBoundrangeString" "svmExpoString"       "svmLinear"
## [160] "svmPoly"             "svmRadial"           "svmRadialCost"
## [163] "svmRadialWeights"    "svmSpectrumString"   "treebag"
## [166] "vbmpRadial"          "widekernelpls"       "WM"
## [169] "xyf"
```

As you can see, there are plenty of models available to satisfy most needs. Some support either **dual use**, while others are either **classification** or**regression** only. You can test for the type a model supports by using the same `getModelInfo` function:

```
getModelInfo()$glm$type
#  "Regression"     "Classification"
```

In the above snippet, `glm` supports both **regression** and **classification**.

We download the **vehicles** data set from Hadley Wickham hosted on Github. To keep this simple, we attempt to predict whether a vehicle has 6 cylinders using only the first 24 columns of the data set:

```
library(RCurl)
urlfile <-'https://raw.githubusercontent.com/hadley/fueleconomy/master/data-raw/vehicles.csv'
x <- getURL(urlfile, ssl.verifypeer = FALSE)
vehicles <- read.csv(textConnection(x))

# alternative way of getting the data if the above snippet doesn't work:
```

```
# urlData <- getURL('https://raw.githubusercontent.com/hadley/fueleconomy/master/data-raw/vehicles.csv')

# vehicles <- read.csv(text = urlData)
```

We clean the outcome variable `cyclinders` by assigning it `1` for 6 cyclinders and `0` for everything else:

```
vehicles <- vehicles[names(vehicles)[1:24]]

vehicles <- data.frame(lapply(vehicles, as.character), stringsAsFactors=FALSE)

vehicles <- data.frame(lapply(vehicles, as.numeric))

vehicles[is.na(vehicles)] <- 0

vehicles$cylinders <- ifelse(vehicles$cylinders == 6, 1,0)
```

We call `prop.table` to understand the proportions of our outcome variable:

```
prop.table(table(vehicles$cylinders))

##      0      1

## 0.6506 0.3494
```

This tells us that 35% of the data represents a vehicle with 6 cylinders. So our data isn't perfectly balanced but it certainly isn't skewed or considered a rare event.

Here is the one complicated part, instead of splitting the data into 2 parts of `train` and `test`, we split the data into 3 parts: `ensembleData`, `blenderData`, and `testingData`:

```
set.seed(1234)

vehicles <- vehicles[sample(nrow(vehicles)),]

split <- floor(nrow(vehicles)/3)
```

```
ensembleData <- vehicles[0:split,]

blenderData <- vehicles[(split+1):(split*2),]

testingData <- vehicles[(split*2+1):nrow(vehicles),]
```

We assign the outcome name to `labelName` and the predictor variables to `predictors`:

```
labelName <- 'cylinders'

predictors <- names(ensembleData)[names(ensembleData) != labelName]
```

We create a **caret** `trainControl` object to control the number of cross-validations performed (the more the better but for breivity we only require 3):

```
myControl <- trainControl(method='cv', number=3, returnResamp='none')
```

Benchmark Model

We run the data on a `gbm` model without any enembling to use as a comparative benchmark:

```
test_model <- train(blenderData[,predictors], blenderData[,labelName], method='gbm', trControl
=myControl)

## Iter   TrainDeviance   ValidDeviance   StepSize   Improve

##      1        0.2147             nan     0.1000    0.0128

##      2        0.2044             nan     0.1000    0.0104

##      3        0.1962             nan     0.1000    0.0084

...
```

You'll see a series of the lines (as shown above) as it trains the `gbm` model.

We then use the model to predict 6-cylinder vehicles using the `testingData` data set and **pROC**'s `auc` function to get the Area Under the Curve (AUC):

```
preds <- predict(object=test_model, testingData[,predictors])

library(pROC)

auc <- roc(testingData[,labelName], preds)

print(auc$auc)

## Area under the curve: 0.99
```

It gives a fairly strong **AUC** score of 0.99 (remember that 0.5 is random and 1 is perfect). Hard to believe we can improve this score using an ensemble of models...

Ensembles

But we're going to try. We now use 3 models - `gbm`, `rpart`, and `treebag` as part of our **ensembles** of models and `train` them with the `ensembleData` data set:

```
model_gbm <- train(ensembleData[,predictors], ensembleData[,labelName], method='gbm', trControl=myControl)


model_rpart <- train(ensembleData[,predictors], ensembleData[,labelName], method='rpart', trControl=myControl)


model_treebag <- train(ensembleData[,predictors], ensembleData[,labelName], method='treebag', trControl=myControl)
```

After our 3 models are trained, we use them to predict **6 cylinder vehicles** on the other two data sets: `blenderData` and `testingData` - **yes, both!!** We need to do this to harvest the predictions from both data sets as we're going to add those predictions as new features to the same data sets. So, as we have 3 models, we're going to add three new columns to both `blenderData` and `testingData`:

```
blenderData$gbm_PROB <- predict(object=model_gbm, blenderData[,predictors])

blenderData$rf_PROB <- predict(object=model_rpart, blenderData[,predictors])

blenderData$treebag_PROB <- predict(object=model_treebag, blenderData[,predictors])


testingData$gbm_PROB <- predict(object=model_gbm, testingData[,predictors])

testingData$rf_PROB <- predict(object=model_rpart, testingData[,predictors])

testingData$treebag_PROB <- predict(object=model_treebag, testingData[,predictors])
```

Please note how easy it is to add those values back to the original data set (follow where we assign the resulting predictions above).

Now we train a final **blending** model on the old data and the new predictions (we use `gbm` but that is completely arbitrary):

```
predictors <- names(blenderData)[names(blenderData) != labelName]

final_blender_model <- train(blenderData[,predictors], blenderData[,labelName], method='gbm', trControl=myControl)
```

And we call `predict` and `roc`/`auc` functions to see how our blended ensemble model fared:

```
preds <- predict(object=final_blender_model, testingData[,predictors])

auc <- roc(testingData[,labelName], preds)

print(auc$auc)

## Area under the curve: 0.993
```

There you have it, an **AUC** bump of 0.003. This may not seem like much (and there are many ways of improving that score) but it should easily give you that extra edge on your next data science competition!!

Full source code (also on GitHub):

```
library(caret)
names(getModelInfo())

# Load data from Hadley Wickham on Github - Vehicle data set and predict 6 cylinder vehicles
library(RCurl)
#urlData <- getURL('https://raw.githubusercontent.com/hadley/fueleconomy/master/data-raw/vehicles.csv')
#vehicles <- read.csv(text = urlData)

# alternative way of getting the data
urlfile <-'https://raw.githubusercontent.com/hadley/fueleconomy/master/data-raw/vehicles.csv'
x <- getURL(urlfile, ssl.verifypeer = FALSE)
vehicles <- read.csv(textConnection(x))

# clean up the data and only use the first 24 columns
vehicles <- vehicles[names(vehicles)[1:24]]
vehicles <- data.frame(lapply(vehicles, as.character), stringsAsFactors=FALSE)
vehicles <- data.frame(lapply(vehicles, as.numeric))
vehicles[is.na(vehicles)] <- 0
vehicles$cylinders <- ifelse(vehicles$cylinders == 6, 1,0)

prop.table(table(vehicles$cylinders))

# shuffle and split the data into three parts
set.seed(1234)
vehicles <- vehicles[sample(nrow(vehicles)),]
split <- floor(nrow(vehicles)/3)
ensembleData <- vehicles[0:split,]
blenderData <- vehicles[(split+1):(split*2),]
testingData <- vehicles[(split*2+1):nrow(vehicles),]

# set label name and predictors
labelName <- 'cylinders'
predictors <- names(ensembleData)[names(ensembleData) != labelName]
```

```
library(caret)
# create a caret control object to control the number of cross-validations performed
myControl <- trainControl(method='cv', number=3, returnResamp='none')

# quick benchmark model
test_model <- train(blenderData[,predictors], blenderData[,labelName], method='gbm', trControl=myControl)
preds <- predict(object=test_model, testingData[,predictors])

library(pROC)
auc <- roc(testingData[,labelName], preds)
print(auc$auc) # Area under the curve: 0.9896

# train all the ensemble models with ensembleData
model_gbm <- train(ensembleData[,predictors], ensembleData[,labelName], method='gbm', trControl=myControl)
model_rpart <- train(ensembleData[,predictors], ensembleData[,labelName], method='rpart', trControl=myControl)
model_treebag <- train(ensembleData[,predictors], ensembleData[,labelName], method='treebag', trControl=myControl)

# get predictions for each ensemble model for two last data sets
# and add them back to themselves
blenderData$gbm_PROB <- predict(object=model_gbm, blenderData[,predictors])
blenderData$rf_PROB <- predict(object=model_rpart, blenderData[,predictors])
blenderData$treebag_PROB <- predict(object=model_treebag, blenderData[,predictors])
testingData$gbm_PROB <- predict(object=model_gbm, testingData[,predictors])
testingData$rf_PROB <- predict(object=model_rpart, testingData[,predictors])
testingData$treebag_PROB <- predict(object=model_treebag, testingData[,predictors])

# see how each individual model performed on its own
auc <- roc(testingData[,labelName], testingData$gbm_PROB )
print(auc$auc) # Area under the curve: 0.9893

auc <- roc(testingData[,labelName], testingData$rf_PROB )
print(auc$auc) # Area under the curve: 0.958

auc <- roc(testingData[,labelName], testingData$treebag_PROB )
print(auc$auc) # Area under the curve: 0.9734
```

```
# run a final model to blend all the probabilities together

predictors <- names(blenderData)[names(blenderData) != labelName]

final_blender_model <- train(blenderData[,predictors], blenderData[,labelName], method='gbm', trControl=myControl)


# See final prediction and AUC of blended ensemble

preds <- predict(object=final_blender_model, testingData[,predictors])

auc <- roc(testingData[,labelName], preds)

print(auc$auc)  # Area under the curve: 0.9922
```

Ensemble Feature Selection On Steroids: {fscaret} Package

October 1, 2014 **Tags:** ***modeling***

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{fscaret}** - feature selection for ensembles
- **{caret}** - machine learning tools

The fscaret package, as its name implies, is closely related to the caret package. It relies on **caret**, and its numerous functions, to get its job done.

So what does this package do?
Well, you give it a data set and a list of models and, in return, **fscaret** will scale and return the importance of each variable for each model and for the ensemble of models. The tool extracts the importance of each variable by using the selected models' VarImp or similar measuring function. For example, linear models use the absolute value of the t-statistic for each parameter and decision-tree models, total the importance of the individual trees, etc. It returns individual and combined MSEs and RMSEs:

- **MSE (Mean Squared Error)**: the variance of the estimator
- **RMSE (Root Mean Squared Error)**: the standard deviation of the sample

Now, **caret** is itself a wrapper sitting atop 170 models and offering many machine learning and tuning functions. I will assume you are familiar with caret, but even if you aren't, **fscaret** is intuitive enough to provide lots of use on its own as an ensemble variable selection tool (but do get to know caret, you won't regret it).

To get a full list of all the models available with the **fscaret** package (which is over a third of the models supported in **caret**), install and load the fscaret package then type in the following:

```
library(fscaret)
data(funcRegPred)
print(funcRegPred)
##  [1] "glm"            "glmStepAIC"     "gam"            "gamLoess"
##  [5] "gamSpline"      "rpart"          "rpart2"         "ctree"
##  [9] "ctree2"         "evtree"         "obliqueTree"    "gbm"
## [13] "blackboost"     "bstTree"        "glmboost"       "gamboost"
## [17] "bstLs"          "bstSm"          "rf"             "parRF"
## [21] "cforest"        "Boruta"         "RRFglobal"      "RRF"
## [25] "treebag"        "bag"            "logicBag"       "bagEarth"
## [29] "nodeHarvest"    "partDSA"        "earth"          "gcvEarth"
## [33] "logreg"         "glmnet"         "nnet"           "mlp"
## [37] "mlpWeightDecay" "pcaNNet"        "avNNet"         "rbf"
## [41] "pls"            "kernelpls"      "simpls"         "widekernelpls"
## [45] "spls"           "svmLinear"      "svmRadial"      "svmRadialCost"
## [49] "svmPoly"        "gaussprLinear"  "gaussprRadial"  "gaussprPoly"
## [53] "knn"            "xyf"            "bdk"            "lm"
## [57] "lmStepAIC"      "leapForward"    "leapBackward"   "leapSeq"
## [61] "pcr"            "icr"            "rlm"            "neuralnet"
## [65] "qrf"            "qrnn"           "M5Rules"        "M5"
## [69] "cubist"         "ppr"            "penalized"      "ridge"
## [73] "lars"           "lars2"          "enet"           "lasso"
## [77] "relaxo"         "foba"           "krlsRadial"     "krlsPoly"
## [81] "rvmLinear"      "rvmRadial"      "rvmPoly"        "superpc"
```

And compare that with the models supported in **caret** (install and load the **caret** package):

```
library(caret)
print(paste('Total models in caret:', length(getModelInfo())))
## [1] "Total models in caret: 169"
# to get full list of supported models type:
```

```
# names(getModelInfo())
```

Even though **caret** supports a lot more than those 84 models, it should be plenty of fire power to select the best variables possible for your needs. When installing **fscaret**, it is recommended to install it with all its dependencies:

```
install.packages("fscaret", dependencies = c("Depends", "Suggests"))
```

This will speed things up tremendously on subsequent runs but you will have to suffer during the installation process. If I remember correctly, it took me over an hour to get all the models installed (some of the models require user confirmations).

The input data needs to be formatted in a particular way: **MISO**. (Multiple Ins, Single Out). The output needs be the last column in the data frame. So you can't have it anywhere else, nor can it predict multiple response columns at once.

As with anything **ensemble** related, if you're going to run 50 models in one shot, you better have the computing muscle to do so - there's no free lunch. Start with a single or small set of models. If you're going to run a large ensemble of models, fire it up before going to bed and see what you get the next day.

For the demo, we'll use a Titanic dataset from the University of Colorado Denver. This is a classic data set often seen in online courses and walkthroughs. The outcome is passenger survivorship (i.e. can you predict who will survive based on various features). We drop the passenger names as they are all unique but keep the passenger titles. We also impute the missing 'Age' variables with the mean:

```
titanicDF <- read.csv('http://math.ucdenver.edu/RTutorial/titanic.txt',sep='\t')

titanicDF$Title <- ifelse(grepl('Mr ',titanicDF$Name),'Mr',ifelse(grepl('Mrs ',titanicDF$Name)
,'Mrs',ifelse(grepl('Miss',titanicDF$Name),'Miss','Nothing')))

titanicDF$Age[is.na(titanicDF$Age)] <- median(titanicDF$Age, na.rm=T)
```

We move the 'Survived' outcome variable to the end of the data frame to be **MISO** compliant:

```
# miso format
titanicDF <- titanicDF[c('PClass', 'Age',    'Sex',   'Title', 'Survived')]
```

To help us process this data, we're going to use some of **caret** functions. First, we call the **dummyVars** function to dummify the 'Title' variable:

```
titanicDF$Title <- as.factor(titanicDF$Title)
titanicDummy <- dummyVars("~.",data=titanicDF, fullRank=F)
titanicDF <- as.data.frame(predict(titanicDummy,titanicDF))
print(names(titanicDF))
##  [1] "PClass.1st"    "PClass.2nd"    "PClass.3rd"    "Age"
##  [5] "Sex.female"    "Sex.male"      "Title.Miss"    "Title.Mr"
##  [9] "Title.Mrs"     "Title.Nothing" "Survived"
```

The other **caret** function we need is the **createDataPartition** to split the data set randomly using a 0.75 split. With this split we allocate three-quarters of the data to the training data set and one quarter to the testing data set:

```
set.seed(1234)
splitIndex <- createDataPartition(titanicDF$Survived, p = .75, list = FALSE, times = 1)
trainDF <- titanicDF[ splitIndex,]
testDF  <- titanicDF[-splitIndex,]
```

Finally, we select five models to process our data and call the meat-and-potatoes function of the **fscaret** package, named as its package, **fscaret**:

```
fsModels <- c("glm", "gbm", "treebag", "ridge", "lasso")
```

```
myFS<-fscaret(trainDF, testDF, myTimeLimit = 40, preprocessData=TRUE,

              Used.funcRegPred = 'gbm', with.labels=TRUE,

              supress.output=FALSE, no.cores=2)
```

If the above code ran successfully, you will see a series of log outputs (unless you set **supress.output** to false). Each model will run through its paces and the final **fscaret** output will list the number of variables each model processed:

```
----Processing files:----

[1] "9in_default_REGControl_VarImp_gbm.txt"     "9in_default_REGControl_VarImp_glm.txt"     "9
in_default_REGControl_VarImp_lasso.txt"

[4] "9in_default_REGControl_VarImp_ridge.txt"   "9in_default_REGControl_VarImp_treebag.txt"
```

The **myFS** holds a lot of information. One of the most interesting result set is the `$VarImp$matrixVarImp.MSE`. This returns the top variables from the perspective of all models involved (the **MSE** is scaled to compare each model equally):

```
myFS$VarImp$matrixVarImp.MSE

##       gbm     SUM    SUM% ImpGrad Input_no

## 5 32.8042 32.8042 100.000    0.00        5

## 3 25.8817 25.8817  78.898   21.10        3

## 7 21.5655 21.5655  65.740   16.68        7

## 4 12.0336 12.0336  36.683   44.20        4

## 1  5.4330  5.4330  16.562   54.85        1

## 6  1.0265  1.0265   3.129   81.11        6

## 9  0.8386  0.8386   2.556   18.30        9

## 8  0.4168  0.4168   1.271   50.29        8

## 2  0.0000  0.0000   0.000  100.00        2
```

We need to do a little wrangling in order to clean this up and get a nicely ordered list with the actual variable names attached:

```
results <- myFS$VarImp$matrixVarImp.MSE

results$Input_no <- as.numeric(results$Input_no)

results <- results[c("SUM","SUM%","ImpGrad","Input_no")]

myFS$PPlabels$Input_no <-  as.numeric(rownames(myFS$PPlabels))

results <- merge(x=results, y=myFS$PPlabels, by="Input_no", all.x=T)

results <- results[c('Labels', 'SUM')]

results <- subset(results,results$SUM !=0)

results <- results[order(-results$SUM),]

print(results)

##          Labels     SUM

## 5    Sex.female 32.8042

## 3    PClass.3rd 25.8817

## 7      Title.Mr 21.5655

## 4           Age 12.0336

## 1    PClass.1st  5.4330

## 6    Title.Miss  1.0265

## 9 Title.Nothing  0.8386

## 8     Title.Mrs  0.4168
```

So, according to the models chosen, 'Sex.female' is the most important variable to predict survivorship in the Titanic dataset, followed by 'PClass.3rd' and 'Title.Mr'.

Full source code (also on GitHub):

```
# warning: could take over an hour to install all models the first time you install the fscaret package

# install.packages("fscaret", dependencies = c("Depends", "Suggests"))
```

```
library(fscaret)

# list of models fscaret supports:
data(funcRegPred)
funcRegPred

library(caret)
# list of models caret supports:
names(getModelInfo())

# using dataset from the UCI Machine Learning Repository (http://archive.ics.uci.edu/ml/)
titanicDF <- read.csv('http://math.ucdenver.edu/RTutorial/titanic.txt',sep='\t')

# creating new title feature
titanicDF$Title <- ifelse(grepl('Mr ',titanicDF$Name),'Mr',ifelse(grepl('Mrs ',titanicDF$Name),'Mrs',ifelse(grepl('Miss',titanicDF$Name),'Miss','Nothing')))
titanicDF$Title <- as.factor(titanicDF$Title)

# impute age to remove NAs
titanicDF$Age[is.na(titanicDF$Age)] <- median(titanicDF$Age, na.rm=T)

# reorder data set so target is last column
titanicDF <- titanicDF[c('PClass', 'Age',    'Sex',   'Title', 'Survived')]

# binarize all factors
titanicDummy <- dummyVars("~.",data=titanicDF, fullRank=F)
titanicDF <- as.data.frame(predict(titanicDummy,titanicDF))

# split data set into train and test portion
set.seed(1234)
splitIndex <- createDataPartition(titanicDF$Survived, p = .75, list = FALSE, times = 1)
trainDF <- titanicDF[ splitIndex,]
testDF  <- titanicDF[-splitIndex,]

# limit models to use in ensemble and run fscaret
fsModels <- c("glm", "gbm", "treebag", "ridge", "lasso")
myFS<-fscaret(trainDF, testDF, myTimeLimit = 40, preprocessData=TRUE,
              Used.funcRegPred = fsModels, with.labels=TRUE,
              supress.output=FALSE, no.cores=2)
```

```
# analyze results
print(myFS$VarImp)
print(myFS$PPlabels)
```

Mapping The United States Census With {ggmap}

September 29, 2014 **Tags:** ***visualizing***

Resources

- YouTube Companion Video
- Full Source Code

Packages Used in this Walkthrough

- **{RCurl}** - downloads https data
- **{xlsx}** - Excel reader
- **{zipcode}** - US zipcode tools and data
- **{ggmap}** - map visualization
- **{ggplot2}** - graphics

If you haven't played with the ggmap package then you're in for a treat! It will map your data on any location around the world as long as you give it proper geographical coordinates.

Even though **ggmap** supports different map providers, I have only used it with Google Maps and that is what I will demonstrate in this article. We're going to download the median household income for the United States from the 2006 to 2010 census. Normally you would need to download a shape file from the Census.gov site but the **University of Michigan's Institute for Social Research** graciously provides an Excel file for the national numbers. The file is limited to the mean and median household numbers for every zip code in the United States.

We're going to load two packages in order to download the data: RCurl to handle HTTP protocols to download the file directly from the Internet andxlsx to read the Excel file and load the sheet named 'Median' into our data.frame:

```
urlfile <-'http://www.psc.isr.umich.edu/dis/census/Features/tract2zip/MedianZIP-3.xlsx'

destfile <- "census20062010.xlsx"

download.file(urlfile, destfile, mode="wb")

census <- read.xlsx2(destfile, sheetName = "Median")
```

```
# NOTE: if you can't download the file automatically, download it manually at:
# 'http://www.psc.isr.umich.edu/dis/census/Features/tract2zip/'
```

We clean the file by keeping only the zip code and median household income variables and casting the median figures from factor to numbers:

```
census <- census[c('Zip','Median..')]
names(census) <- c('Zip','Median')
census$Median <- as.character(census$Median)
census$Median <- as.numeric(gsub(',','',census$Median))
print(head(census,5))
##    Zip Median
## 1 1001  56663
## 2 1002  49853
## 3 1003  28462
## 4 1005  75423
## 5 1007  79076
```

You will notice that the zip codes above only have 4 digits. We leverage another package called zipcode to not only clean our zip codes by removing any '+4' data and padding with zeros where needed, but more importantly, to give us the central latitude and longitude coordinate for our zip codes (this requires downloading the zipcode data file):

```
data(zipcode)
census$Zip <- clean.zipcodes(census$Zip)
```

We merge our census data with the zipcode data on zip codes:

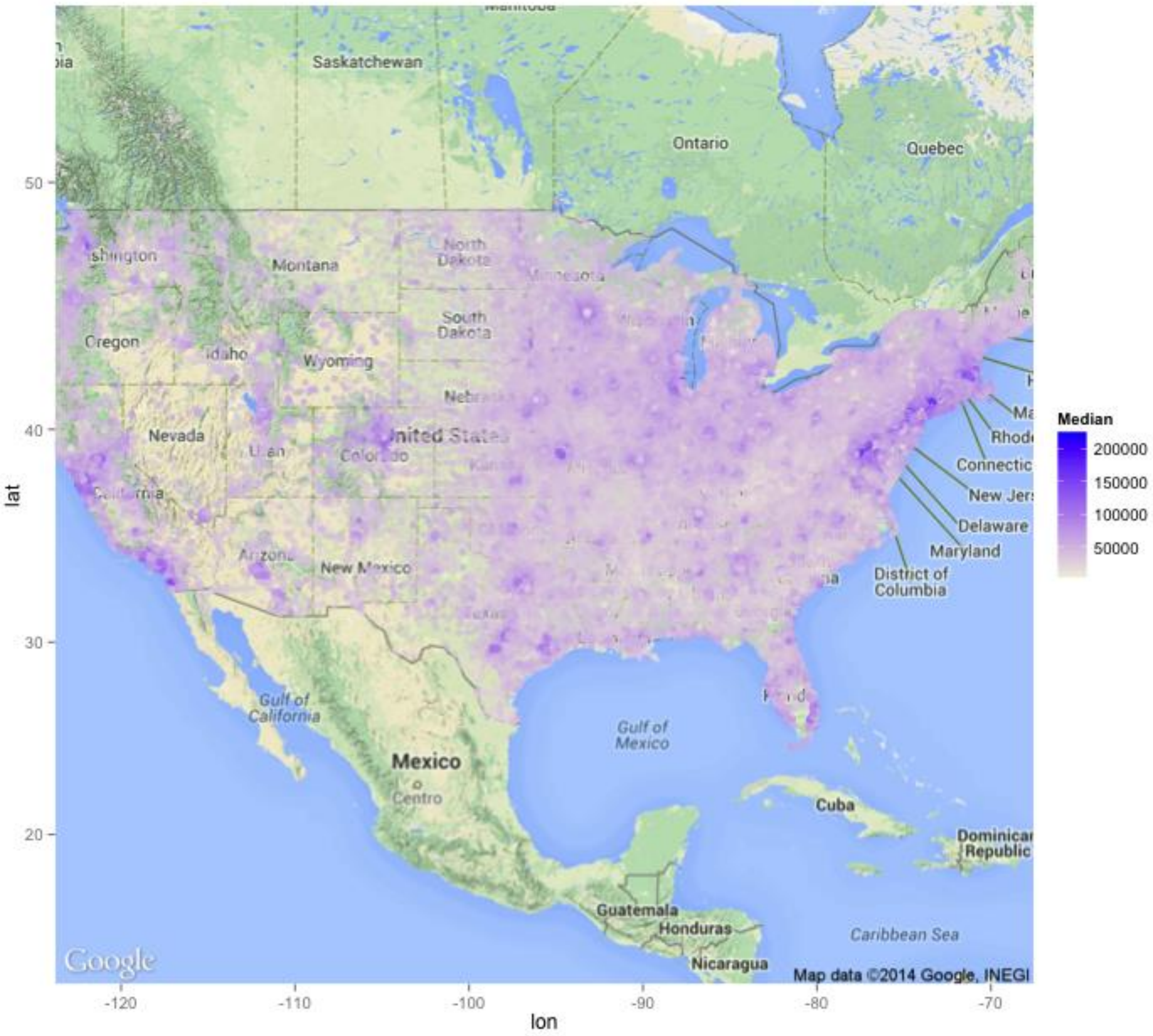
```
census <- merge(census, zipcode, by.x='Zip', by.y='zip')
```

Finally, we reach the heart of our mapping goal, we download a map of the United States using **ggmap**. For a more detailed introduction to ggmap, check out this article written by the authors of the package. The **get_map** function downloads the map as an image. Amongst the available parameters, we opt for zoom level 4 (which works well to cover the US), and request a colored, terrain-type map (versus satellite or black and white, amongst many otherr options):

```
map<-get_map(location='united states', zoom=4, maptype = "terrain",

             source='google',color='color')

## Map from URL : http://maps.googleapis.com/maps/api/staticmap?center=united+states&zoom=4&si
ze=%20640x640&scale=%202&maptype=terrain&sensor=false

## Google Maps API Terms of Service : http://developers.google.com/maps/terms

## Information from URL : http://maps.googleapis.com/maps/api/geocode/json?address=united+stat
es&sensor=false

## Google Maps API Terms of Service : http://developers.google.com/maps/terms
```

And ggplot2 that will handle the graphics (based on the grammar of graphics) where we pass our census data with the geographical coordinates:

```
ggmap(map) + geom_point(

        aes(x=longitude, y=latitude, show_guide = TRUE, colour=Median),

        data=census, alpha=.5, na.rm = T)  +

        scale_color_gradient(low="beige", high="blue")
```



And there you have it, the median household income from 2006 to 2010 mapped onto a Google Map of the US in just a few lines of code! You can play around with the alpha setting to increase or decrease the transparency of the census data on the map.

Full source code (also on GitHub):

```
require(RCurl)
require(xlsx)

# NOTE if you can't download the file automatically, download it manually at:
# 'http://www.psc.isr.umich.edu/dis/census/Features/tract2zip/'
urlfile <-'http://www.psc.isr.umich.edu/dis/census/Features/tract2zip/MedianZIP-3.xlsx'
destfile <- "census20062010.xlsx"
download.file(urlfile, destfile, mode="wb")
census <- read.xlsx2(destfile, sheetName = "Median")

# clean up data
census <- census[c('Zip','Median..')]
names(census) <- c('Zip','Median')
census$Median <- as.character(census$Median)
census$Median <- as.numeric(gsub(',','',census$Median))
print(head(census,5))

# get geographical coordinates from zipcode
require(zipcode)
data(zipcode)
census$Zip <- clean.zipcodes(census$Zip)
census <- merge(census, zipcode, by.x='Zip', by.y='zip')

# get a Google map
require(ggmap)
map<-get_map(location='united states', zoom=4, maptype = "terrain",
             source='google',color='color')

# plot it with ggplot2
require("ggplot2")
ggmap(map) + geom_point(
        aes(x=longitude, y=latitude, show_guide = TRUE, colour=Median),
        data=census, alpha=.8, na.rm = T)  +
        scale_color_gradient(low="beige", high="blue")
```

Brief Guide on Running RStudio Server On Amazon Web Services

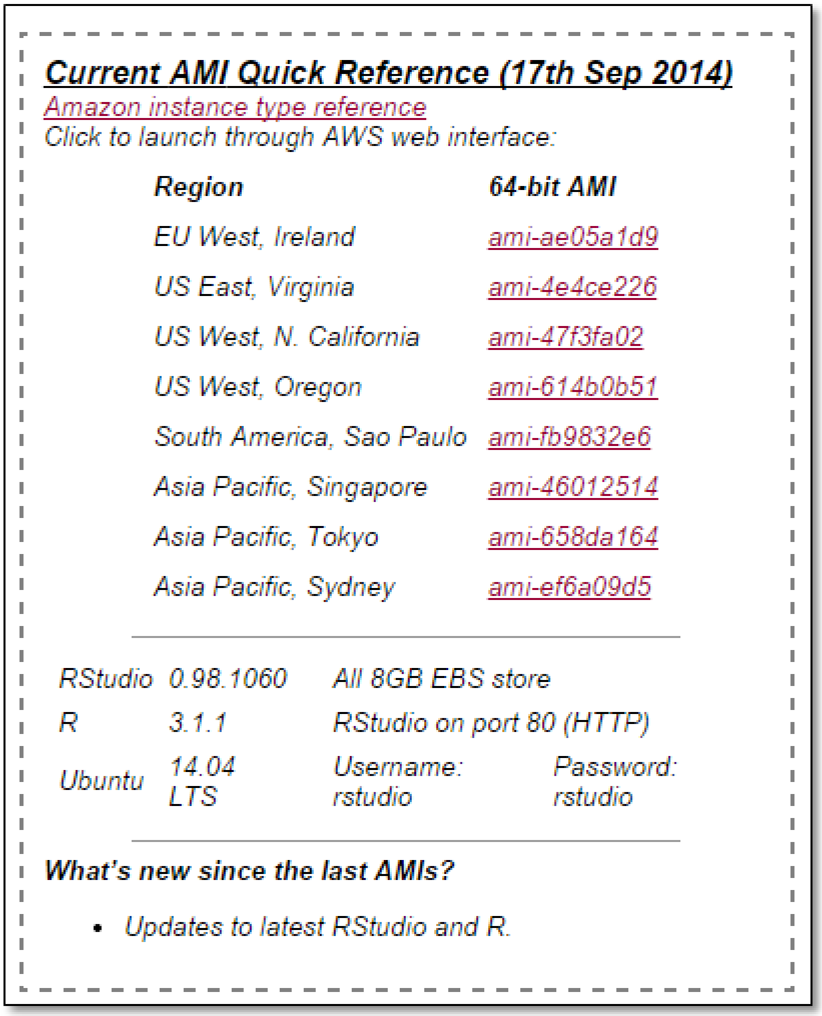
May 15, 2014 **Tags: *rstudioserver***

Resources

- YouTube Companion Video

There are a lot of great resources on the web, but I didn't find one that covered my needs from end to end, and figured others could benefit from a walkthrough. Here, I'll show how to select, install and run RStudio Server, customize security settings to use the RStudio's web interface, and upload and download data between your local machine and the server.

If you like Kaggle competitions, like I do, then this is a great way of quickly adding all sorts of computing configurations to satiate your needs. Our first stop is at Louis Aslet's web page. Louis curates a series of Amazon Machine Images (referred as AMIs):



Current AMI Quick Reference (17th Sep 2014)

Amazon instance type reference

Click to launch through AWS web interface:

Region	64-bit AMI
EU West, Ireland	ami-ae05a1d9
US East, Virginia	ami-4e4ce226
US West, N. California	ami-47f3fa02
US West, Oregon	ami-614b0b51
South America, Sao Paulo	ami-fb9832e6
Asia Pacific, Singapore	ami-46012514
Asia Pacific, Tokyo	ami-658da164
Asia Pacific, Sydney	ami-ef6a09d5

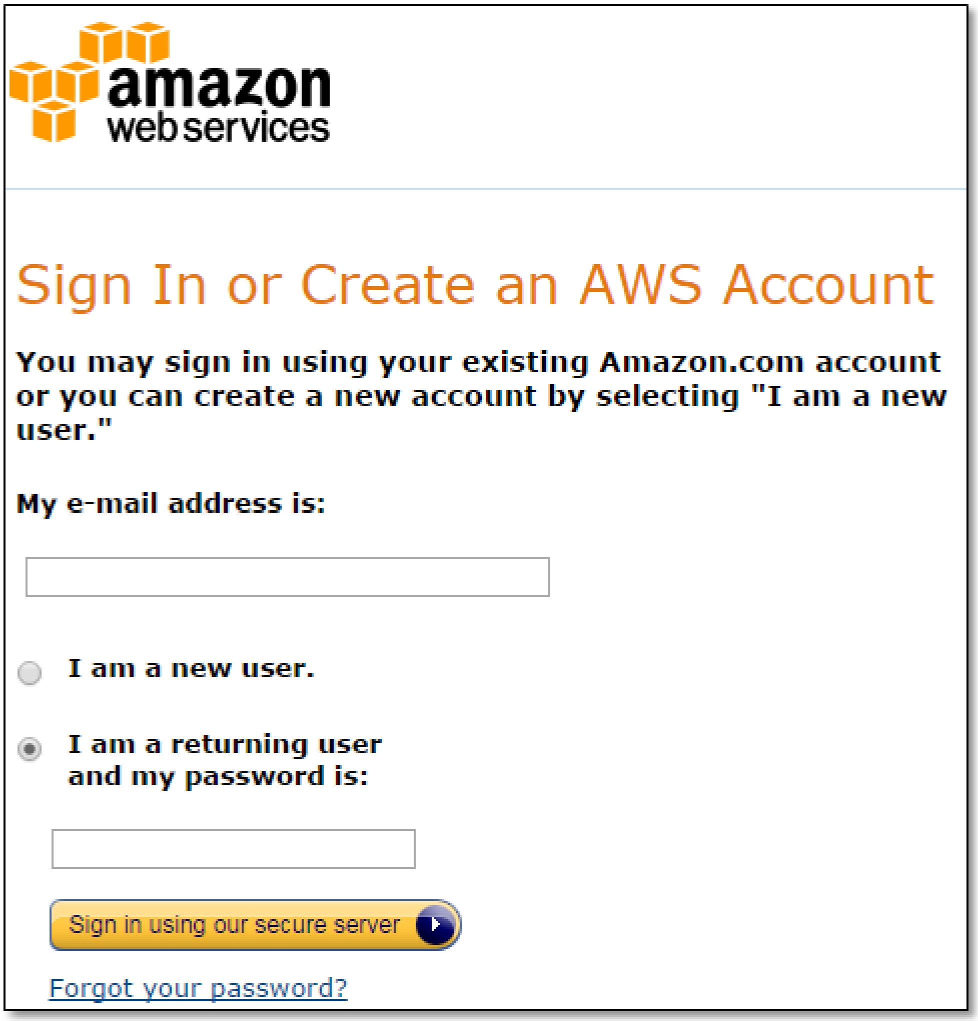
RStudio	0.98.1060	All 8GB EBS store	
R	3.1.1	RStudio on port 80 (HTTP)	
Ubuntu	14.04 LTS	Username: rstudio	Password: rstudio

What's new since the last AMIs?

- Updates to latest RStudio and R.

These are pre-configured images that will install RStudio and common R packages onto a computing instance. This is a huge time and money saver as it automatically installs a whole slew of software under two minutes - and when you're charged by the minute, every minute counts.

Louis also has a video, albeit short, on how to setup RStudio and lots of additional resources. So explore them if you have unanswered questions. We're interested in the upper right-hand box where you need to select the AMI closest to your location and click on it. This will take you to the Amazon Web Services page. If you do not have an AWS account, it will prompt you to set one up:



Sign In or Create an AWS Account

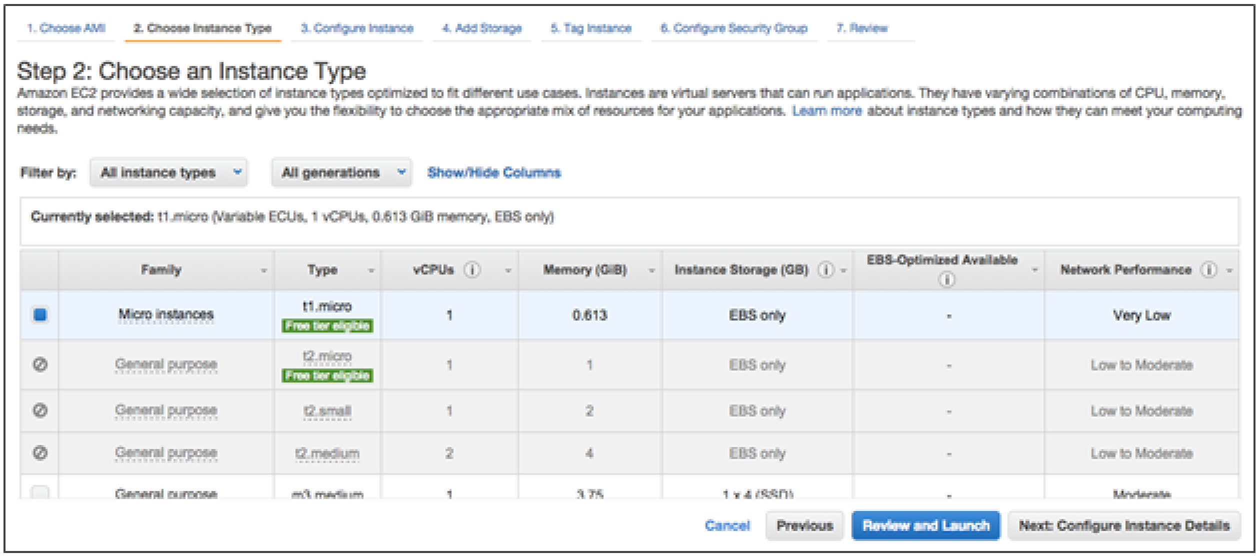
You may sign in using your existing Amazon.com account or you can create a new account by selecting "I am a new user."

My e-mail address is:

○ I am a new user.

● I am a returning user and my password is:

Otherwise it will take you to **Step 2**. This is the fun part. Its like going to the store and picking up a brand new computer. Here you get to choose how much computing muscle you want. The AMI image you selected earlier will get applied to whatever setup you choose. You can go for more GPU, memory, storage, etc. Unfortunately, throwing more memory at a problem is not a guarantee to make it go away - and I'm talking from personal experience here.



Step 2: Choose an Instance Type

Amazon EC2 provides a wide selection of instance types optimized to fit different use cases. Instances are virtual servers that can run applications. They have varying combinations of CPU, memory, storage, and networking capacity, and give you the flexibility to choose the appropriate mix of resources for your applications. Learn more about instance types and how they can meet your computing needs.

Filter by: All instance types | All generations | Show/Hide Columns

Currently selected: t1.micro (Variable ECUs, 1 vCPUs, 0.613 GiB memory, EBS only)

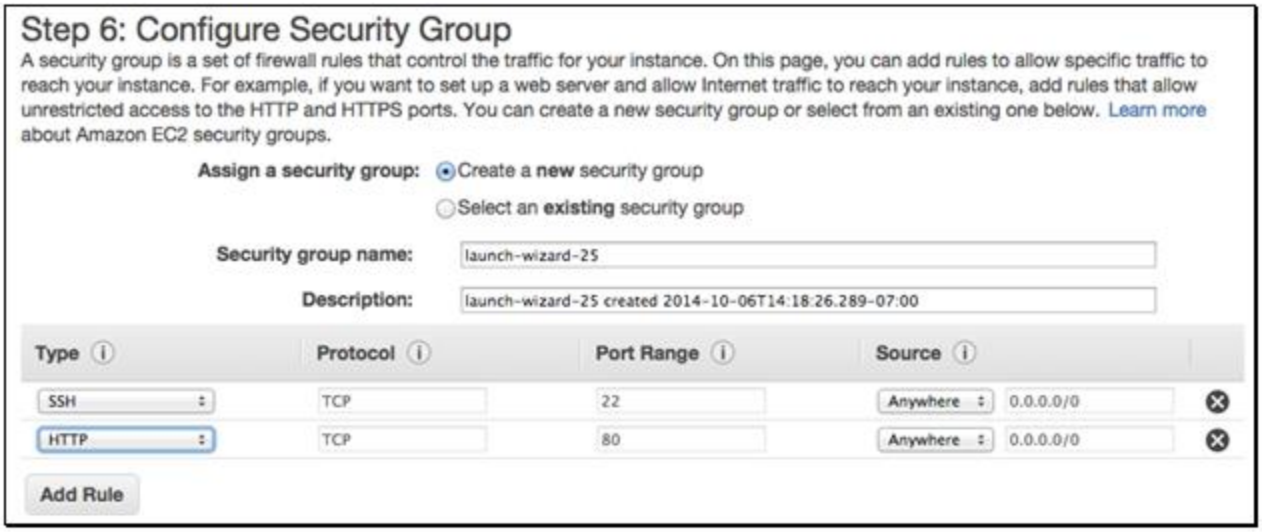
	Family	Type	vCPUs	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
■	Micro instances	t1.micro Free tier eligible	1	0.613	EBS only	-	Very Low
⊘	General purpose	t2.micro Free tier eligible	1	1	EBS only	-	Low to Moderate
⊘	General purpose	t2.small	1	2	EBS only	-	Low to Moderate
⊘	General purpose	t2.medium	2	4	EBS only	-	Low to Moderate
□	General purpose	m3.medium	1	3.75	1 x 4 (SSD)	-	Moderate

I recommend starting small as it is easy to upgrade an existing instance to something bigger.

Security

You need to have two open ports and a key-pair to communicate with your instance. `Port 22` should be opened by default and you'll need to add `port 80`.

`Port 22` is used to connect a command line terminal tool using SSH. I will not be showing that today. Instead, we'll be using `port 80` which allows access to the web interface of RStudio. So add another rule, choose `HTTP`, enter the value **80**, and leave the rest as is:



Step 6: Configure Security Group

A security group is a set of firewall rules that control the traffic for your instance. On this page, you can add rules to allow specific traffic to reach your instance. For example, if you want to set up a web server and allow Internet traffic to reach your instance, add rules that allow unrestricted access to the HTTP and HTTPS ports. You can create a new security group or select from an existing one below. Learn more about Amazon EC2 security groups.

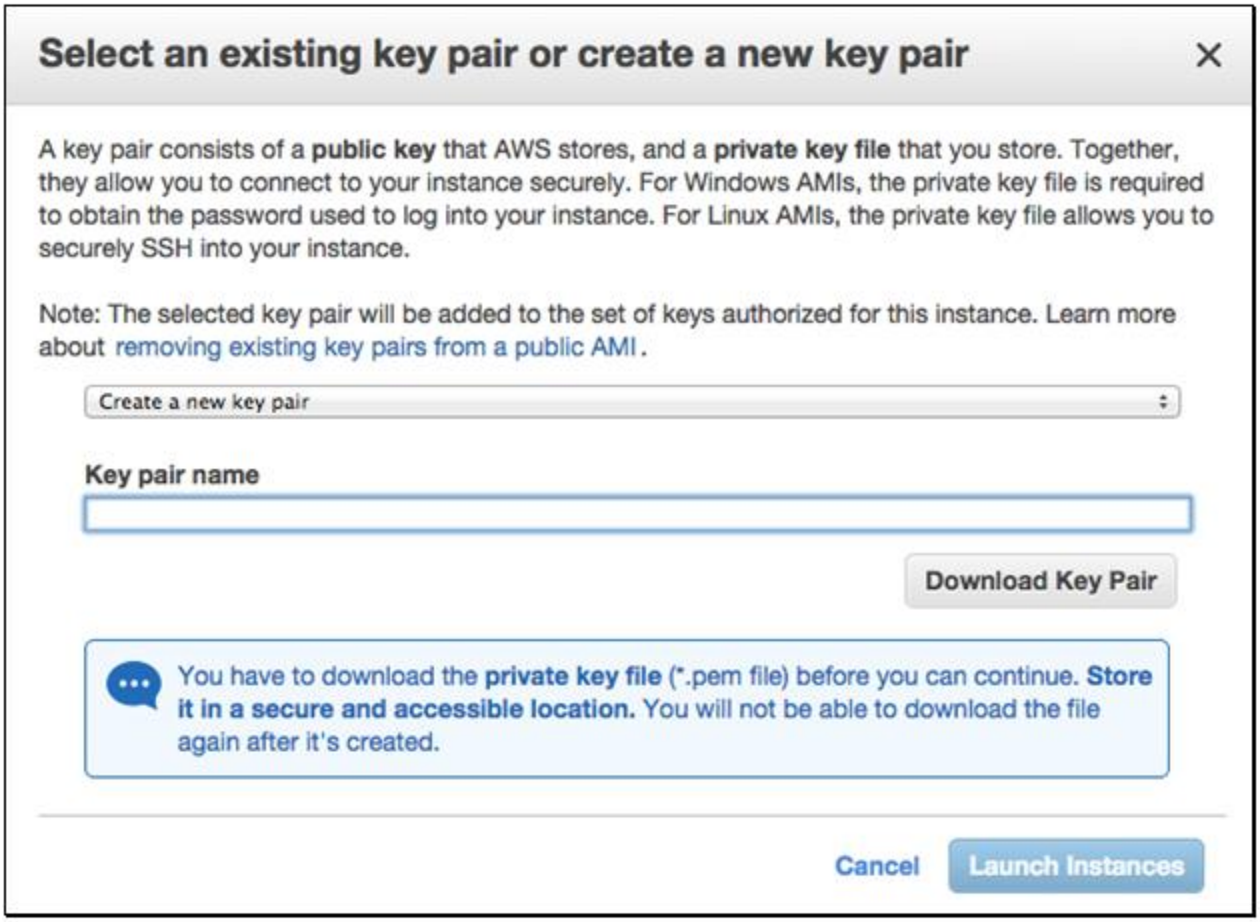
Assign a security group: ● Create a **new** security group
○ Select an **existing** security group

Security group name: launch-wizard-25

Description: launch-wizard-25 created 2014-10-06T14:18:26.289-07:00

Type	Protocol	Port Range	Source	
SSH	TCP	22	Anywhere	0.0.0.0/0
HTTP	TCP	80	Anywhere	0.0.0.0/0

After you hit **Launch**, a key-pair pop-up box will appear. This is what authenticates your computer's identity and allows you to communicate securely to your AWS instance. If this is your first time using EC2 you'll want to create and download a new key pair:



Select an existing key pair or create a new key pair

A key pair consists of a **public key** that AWS stores, and a **private key file** that you store. Together, they allow you to connect to your instance securely. For Windows AMIs, the private key file is required to obtain the password used to log into your instance. For Linux AMIs, the private key file allows you to securely SSH into your instance.

Note: The selected key pair will be added to the set of keys authorized for this instance. Learn more about removing existing key pairs from a public AMI .

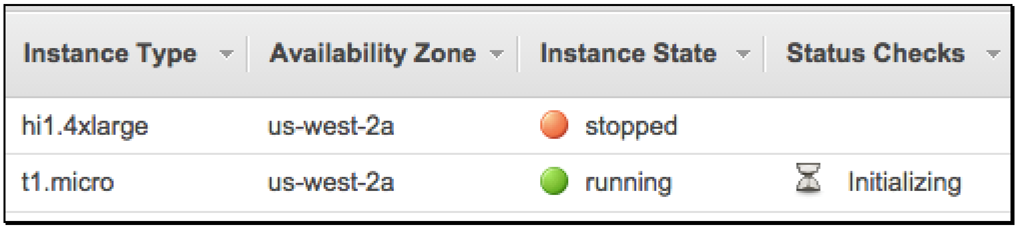
Create a new key pair

Key pair name

You have to download the **private key file** (*.pem file) before you can continue. **Store it in a secure and accessible location.** You will not be able to download the file again after it's created.

View Instance

After launching your instance, once the **instance state** goes from `initializing` to `running`, the public DNS string is the official link to your instance's RStudio web interface:

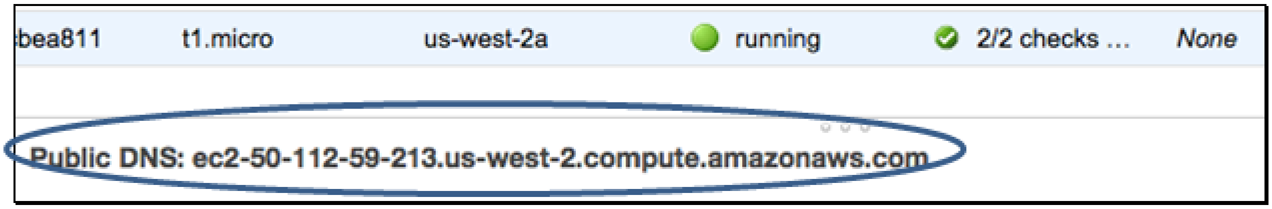


Instance Type	Availability Zone	Instance State	Status Checks
hi1.4xlarge	us-west-2a	stopped	
t1.micro	us-west-2a	running	Initializing

- Remember where you save this key-pair as you cannot communicate to your instance without it.

- If security is a big concern, there are plenty of additional settings and customizations available, please refer to the online RStudio server and Amazon EC2 help files regarding your options.

Once your instance is running (green light), click on it, copy the Public DNS URL and paste it in your browser:

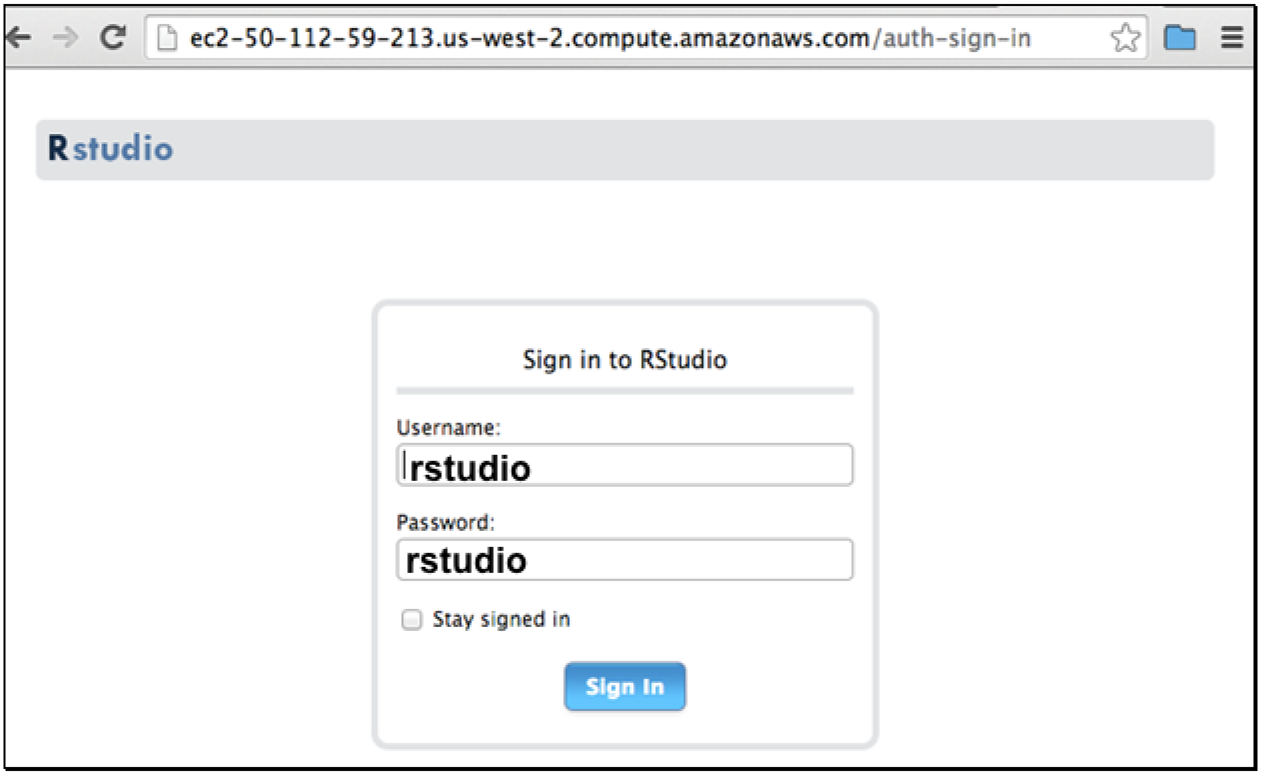


bea811 t1.micro us-west-2a ● running ✓ 2/2 checks ... *None*

Public DNS: ec2-50-112-59-213.us-west-2.compute.amazonaws.com

Using RStudio Server

You will be prompted for your credentials. By default, the initial **account** and **password** for these AMIs is `rstudio`, `rstudio`, all lower case:



ec2-50-112-59-213.us-west-2.compute.amazonaws.com/auth-sign-in

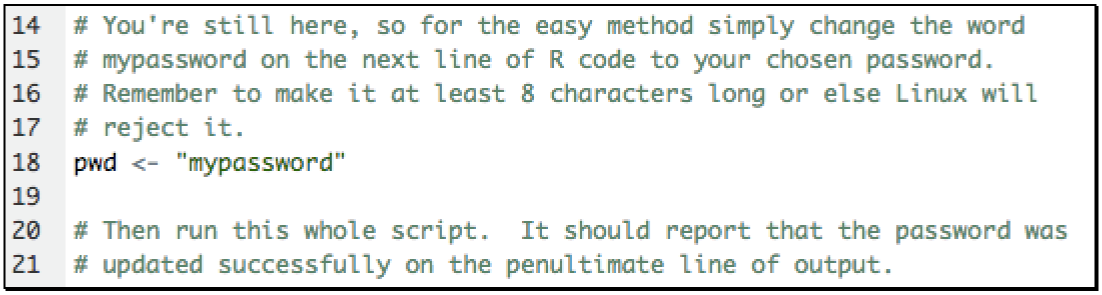
Sign in to RStudio

Username: rstudio

Password: rstudio

☐ Stay signed in

Once in RStudio, the first thing you need to do is run the loaded script and change the password (minimum length required is 8 characters). Replace the **mypassword** with your new password and hit the `run` script button. Then log out and back in with the new password:



```
# You're still here, so for the easy method simply change the word
# mypassword on the next line of R code to your chosen password.
# Remember to make it at least 8 characters long or else Linux will
# reject it.
pwd <- "mypassword"

# Then run this whole script.  It should report that the password was
# updated successfully on the penultimate line of output.
```

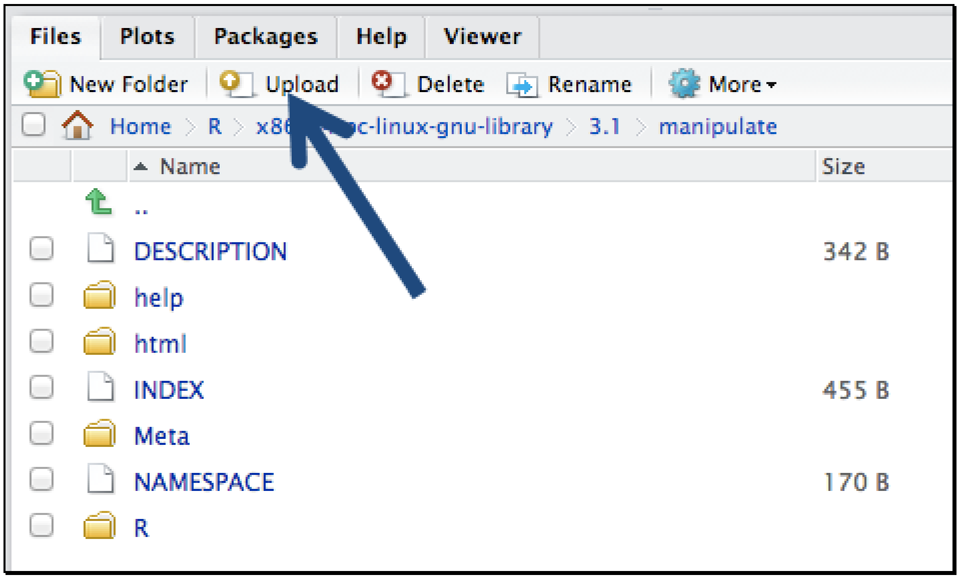
Uploading and Downloading Files

The last thing I want to cover is how to upload and download files to and from your EC2 instance.

Uploading Files (i.e. copying files from your local machine to your EC2 Instance):

- Switch to the **Files** pane
- Navigate to the directory you wish to upload files into
- Click the **Upload** toolbar button. A menu box will open and select the file you want to upload
- Choose the file you wish to upload and press **OK**

Note: you can upload several files or even an entire folder at once, you just need to compress everything into a zip file and upload it (when RStudio receives an uploaded zip file it will automatically uncompresses it).

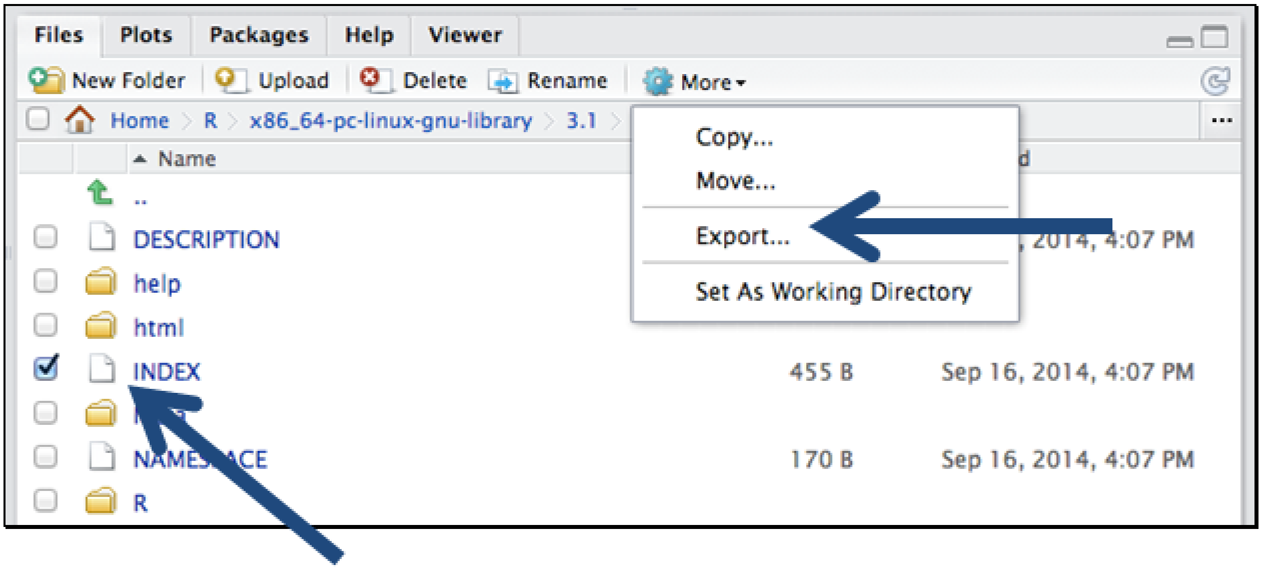


	Name	Size
	..	
□	DESCRIPTION	342 B
□	help	
□	html	
□	INDEX	455 B
□	Meta	
□	NAMESPACE	170 B
□	R	

Downloading Files (i.e. copying files from your EC2 instance to your local machine):

- Switch to the directory you want to download files from the **Files** pane

- Select the file(s) and/or folder(s) you want to download
- Click **More** and **Export** on the toolbar
- You'll then be prompted with a default file name for the download. Either accept the default or specify a custom name then press **OK**



Important Don't forget to shut down the server or terminate it completely - otherwise the meter will keep running and you will keep being charged!

Additional Resources (PDFs)

RStudio Server Administrator's Guide

Using RStudio on Amazon EC2 under the Free Usage Tier