Session 4 - Statistical Modeling

Jongbin Jung

January 10, 2016

Dependencies

- Latest version ($\geq$ `3.1.2`) of `R` (*free* from `https://www.r-project.org/`)
- Latest version of `Rstudio` (also *free* from `https://www.rstudio.com/`)
- Packages

```
lm, glm # loaded with R by default
install.packages('glmnet')
install.packages('randomForest')
install.packages('ROCR')
install.packages('ggplot2')
install.packages('dplyr')
install.packages('caret')  # optional
```

- We'll use the `diamonds` dataset included in `ggplot2` for demonstration

Statistical Modeling

- In this session, I'll demonstrate some common statistical modeling techniques used in R
- But I *won't*
  - Go into the theory/details of each (or any) method/model; I assume you know all that already
  - Introduce the whole `caret` package, which is a **great** package, but I won't focus on it because:
    1. it's pretty well documented here (`http://topepo.github.io/caret/index.html`), and
    2. I (very personally) don't think 'teaching people how to use the `caret` package' is equivalent to 'teaching people statistical modeling in R'
- At the very least, you'll end up with a set of (hopefully useful) R snippets

Basic Framework

- Just to refresh, a basic framework of the statistical modelling process looks something like,
    1. Split the data: Train/(Validation/Test) or CV
    2. Format the data appropriately (pre-processing)
    3. Train a model on the training data
    4. Evaluate the performance on validation/test data
- Would you agree with the order of steps 1 and 2?
- Some considerations in each step, let's discuss a few

Data Splitting

Random Sampling

- Most simple, but often sufficient way of splitting data is to generate random samples
- Think of generating a sample of row numbers, and then using the row numbers to actually create each dataset, e.g., for 50/50 split

```
ind <- 1:nrow(diamonds)
train_p <- .5  # proportion of training data
train_ind <- sample(ind, train_p * nrow(diamonds))
diamond_train <- diamonds[train_ind, ]
diamond_test <- diamonds[-train_ind, ]
```

Stratified Sampling

- Split data while maintaining proportion of certain subgroups
- Use `group_by()` and `sample_frac()` in `dplyr` to select a subset of the data that satisfies the criteria
- Use `setdiff()` in `dplyr` to creat the complement subset

Stratified Sampling: Example

```
diamond_train <- diamonds %>% group_by(cut) %>%
    sample_frac(.5) %>% ungroup()
diamond_test <- setdiff(diamonds, diamond_train)

# check the proportions
cbind(test=summary(diamond_train$cut)/nrow(diamonds),
      train=summary(diamond_test$cut)/nrow(diamonds))
```

```
##             test  train
## Fair      0.0149 0.0147
## Good      0.0455 0.0453
## Very Good 0.1120 0.1118
## Premium   0.1278 0.1272
## Ideal     0.1998 0.1989
```

More than One Split

- Often, you'll want more than one split, e.g., train/validate/test, cross validation
- One obvious way is to use the previous method recursively
- Let's try this as an **Exercise**!

1. From the `diamonds` data, create a 50:30:20 split of train:validate:test data. Name the data frames `dia_train`, `dia_valid`, and `dia_test`, respectively.

(solution script is on the next slide)

More than One Split: Exercise Solution

```r
ind <- 1:nrow(diamonds)
train_p <- .5  # proportion of training data
valid_p <- .3  # proportion of validation data
train_ind <- sample(ind, train_p * nrow(diamonds))
dia_train <- diamonds[train_ind, ]
dia_tmp <- diamonds[-train_ind, ]
ind <- 1:nrow(dia_tmp)
valid_ind <- sample(ind, valid_p * nrow(diamonds))
dia_valid <- dia_tmp[valid_ind, ]
dia_test <- dia_tmp[-valid_ind, ]
```

- We'll use these three datasets in the following exercises

More than One Split (cont'd)

- As you can imagine, this starts getting messy for more than two splits
- A good alternative is `createFolds()` from `caret` (I know I said I wouldn't cover `caret`, but this is one exception . . .)
- Also, for more than 3 splits, you might want to manage each split with labels, rather than creating multiple data frames

```
nsplits <- 10  # the number of splits you want
split_ind <- createFolds(diamonds$carat, k=nsplits)

diamonds_split <- diamonds
for (x in 1:nsplits) {
    ind <- split_ind[[x]]  # indexing a list
    diamonds_split[ind, 'split_id'] <- x
}
```

Pre-processing

`scale()`

- Use `scale()` to center/scale variables (columns) of your dataset
- `scale()` only works on numerical columns
- It's up to you to give `scale()` just the variables you want to manipulate
- The general idea is
  1. Extract the variable(s) you want to center/scale
  2. Use `scale()` to manipulate those variables
  3. Create a copy of your original data with the desired variables manipulated
- Remember to center/scale **all** partitions of your data, **but be aware of where the centering/scaling parameters come from!**

scale(): Example

```r
# Create a copy of the data
train_std <- diamond_train
test_std <- diamond_test

# extract numerical columns and their names
train_num <- train_std[, sapply(train_std, is.numeric)]
test_num <- test_std[, sapply(test_std, is.numeric)]
numcol_names <- names(train_num)

# apply scale() to train data and save parameters
train_num <- scale(train_num)
param_center <- attr(train_num, 'scaled:center')
param_scale <- attr(train_num, 'scaled:scale')
```

scale(): Example (cont'd)

```
# apply to numeric test data columns
test_num <- scale(test_num, center=param_center,
                  scale = param_scale)

# replace numeric columns with scaled ones
train_std[, numcol_names] <- train_num
test_std[, numcol_names] <- test_num
```

- Some notes:
  - Be careful about how you choose 'numeric' variables: binary variables?
  - There are other ways to do this, but this seems to be the best I've found so far

model.matrix()

- While many models work just fine with data frames, some models require that you provide data in the form of a purely numeric matrix (aka model matrix)
- This means converting `factor` variables into multiple binary variables (variables that only have 0 or 1 as values)
- The `model.matrix()` function in `R` does a good job of generating model matrices catered to the formula of your model
- The `R` representation of a model formula such as

$$y_{\texttt{carat}} = \beta_0 + \beta_{\texttt{cut}} x_{\texttt{cut}} + + \beta_{\texttt{depth}} x_{\texttt{depth}}$$

would be

```
carat ~ cut + depth
```

model.matrix(): Example

- To construct a model matrix for the formula

$$y_{\texttt{carat}} = \beta_0 + \beta_{\texttt{cut}} x_{\texttt{cut}} + +\beta_{\texttt{depth}} x_{\texttt{depth}}$$

```
train_mm <- model.matrix(carat ~ cut + depth,
                         train_std)
head(train_mm, 2)
```

```
##   (Intercept)  cut.L cut.Q  cut.C cut^4 depth
## 1           1 -0.632 0.535 -0.316  0.12 -2.99
## 2           1 -0.632 0.535 -0.316  0.12  2.84
```

```
levels(train_std$cut)
```

```
## [1] "Fair"      "Good"      "Very Good"
## [4] "Premium"   "Ideal"
```

`model.matrix()`: Example (cont'd)

- Note that
  - Orthogonal polynomial coding is used for ordinal variable `cut`, where `.L`, `.Q`, `.C`, and `^4` stand for Linear, Quadratic, Cubic, and 4th power
  - `model.matrix()` drops one level as the 'base case', c.f., `cut` has five levels but only four orders in the `model.matrix`
- Some shortcuts in formula
  - "`.`" is used to include all variables (except the target, i.e., variable to the left of ~)
  - "`:`" is used to indicate interaction terms
  - "`-`" (as opposed to +) can be used to exclude certain variables

Exercise

1. With the datasets `dia_train` and `dia_test`, creat an additional variable `expensive`, which is a binary variable with value `yes` if `price` is greater than the median of `price` from `dia_train`, and `no` otherwise.
2. Standardize (scale and center) all numeric columns of the `dia_train` and `dia_test` datasets and call them `train_std` and `test_std`, respectively.
3. Generate model matrices that uses all variables except `expensive` to predict `price` for both datasets. Use variable names `train_mm` and `test_mm`. Note we can use these datasets to train/test a model to predict `expensive` as well!

Solution 1

```
medprice <- median(dia_train$price)
dia_train <- dia_train %>%
    mutate(expensive =
               ifelse(price > medprice, 'yes', 'no'))
dia_test <- dia_test %>%
    mutate(expensive =
               ifelse(price > medprice, 'yes', 'no'))
```

Solution 2

```
train_std <- dia_train
test_std <- dia_test

train_num <- train_std[, sapply(train_std, is.numeric)]
test_num <- test_std[, sapply(test_std, is.numeric)]
numcol_names <- names(train_num)

train_num <- scale(train_num)
param_center <- attr(train_num, 'scaled:center')
param_scale <- attr(train_num, 'scaled:scale')

test_num <- scale(test_num, center=param_center,
                  scale = param_scale)

train_std[, numcol_names] <- train_num
test_std[, numcol_names] <- test_num
```

Solution 3

```
train_mm <-
    model.matrix(price ~ . - expensive, train_std)
test_mm <-
    model.matrix(price ~ . - expensive, train_std)
```

Training models

(OLS) Linear Regression

- Linear regression models can be fitted in `R` using `lm`

Logistic Regression with `glm`

Regularized Linear Models with glmnet

Random Forests

Prediction/Evaluation

Bootstrap