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('ggplot2')  # just for dataset
install.packages('dplyr')
install.packages('caret')  # optional
# for evaluating classification models
install.packages('ROCR')
```

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

Statistical Modeling

- In this session, I'll share some statistical modeling techniques I've picked up using `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 0.0148
## Good       0.0455 0.0452
## Very Good 0.1120 0.1118
## Premium    0.1278 0.1272
## Ideal      0.1998 0.1990
```

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

```
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  2.26
## 2           1 -0.632 0.535 -0.316  0.12  4.77
```

```
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, test_std)
```

Training models

(OLS) Linear Regression

- Linear regression models can be fitted in `R` using `lm`, with the syntax

```
my_model <- lm(formula, data)
```

- The `data` should be a data frame, and the formula should refer to the column names of `data` as variables
- Explore the model with generic functions `summary()` and `coef()`, once trained

```
summary(my_model)
coef(my_model)
```

- Values (e.g., 'risiduals') of the model can be indexed with the $ operator

(OLS) Linear Regression: Example

- Let's build a linear regression model of `price` against `carat` and `cut`, using the `train_std` we created earlier

```
fm <- lm(price ~ carat + cut, train_std)
```

- Explore the model with

```
coef(fm)
summary(fm)
```

- Note how `cut` is automatically transformed to an appropriate form

Logistic Regression with glm

- We can use glm for generalized linear models, just like we use lm for OLS models

```
my_model <- glm(formula, data, family)
```

- family is a description of the link function to be used
- Recall, a logistic regression is a generalized linear model that uses a logit link function
- The logit link function is described in a binomial family object in R (see ?family for other link functions)
- So, to fit a logistic regression model, we write

```
my_model <- glm(formula, data, family='binomial')
```

- Keep in mind, the target variable in the formula must numeric between 0 and 1

Logistic Regression with glm: Example

- With our test_std dataset, let's fit a logistic regression model of expensive against carat and cut
- Since expensive is not numeric, we must convert it first

```
train_std$exp_numeric <-
    ifelse(train_std$expensive == 'yes', 1, 0)
fm <- glm(exp_numeric ~ carat + cut,
          train_std, family='binomial')
```

- Explore the model with

```
coef(fm)
summary(fm)
```

Regularized Linear Models with `glmnet`

- There are many packages that deal with regularized linear models
- I (personally) find `glmnet` to be most useful and consistent
- The objective function for `glmnet` is

$$\texttt{glm objective} - \lambda \times \texttt{penalty}$$

- Where the `penalty` is defined as

$$(1-\alpha)/2\|\beta\|_2^2 + \alpha\|\beta\|_1$$

- Note that $\alpha = 1$ is the L1 (lasso) penalty, and $\alpha = 0$ is the L2 (ridge) penalty

Regularized Linear Models with `glmnet` (cont'd)

- The `glmnet` syntax is

```
my_model <- glmnet(x, y, family, alpha, nlambda)
```

- **`x`** is the `model matrix`
- **`y`** is the target variable vector
- **`family`** determines the link function (Gaussian by default)
- **`alpha`** determines lasso (1), ridge (0), or a mix of the two
- **`nlambda`** controls the number of λ values to try
- Note that `glmnet` automatically computes its own sequence of λ values based on `nlambda`, and using a specific value of λ is discouraged (see `?glmnet` for details)

Regularized Linear Models with `glmnet`: Choosing λ

- The penalty weight λ is a free parameter in regularized linear models, and must be determined somehow
- A quick and dirty, but pretty reasonable way to choose λ with `glmnet` is to use `cv.glmnet()`
- `cv.glmnet()` does k-fold cross-validation for `glmnet`, and returns values/models for λ
- **In addition to `glmnet` arguments**, in `cv.glmnet()` we can set the number of folds k to use in k-fold cross validation by supplying the argument `nfolds`. Default is 10.
- The `plot()` function for `cv.glmnet()` provides some insight on the value of λ and cross-validated model performance

Regularized Linear Models: Example

- For L1-regularized model of `price` against `carat` and `cut`

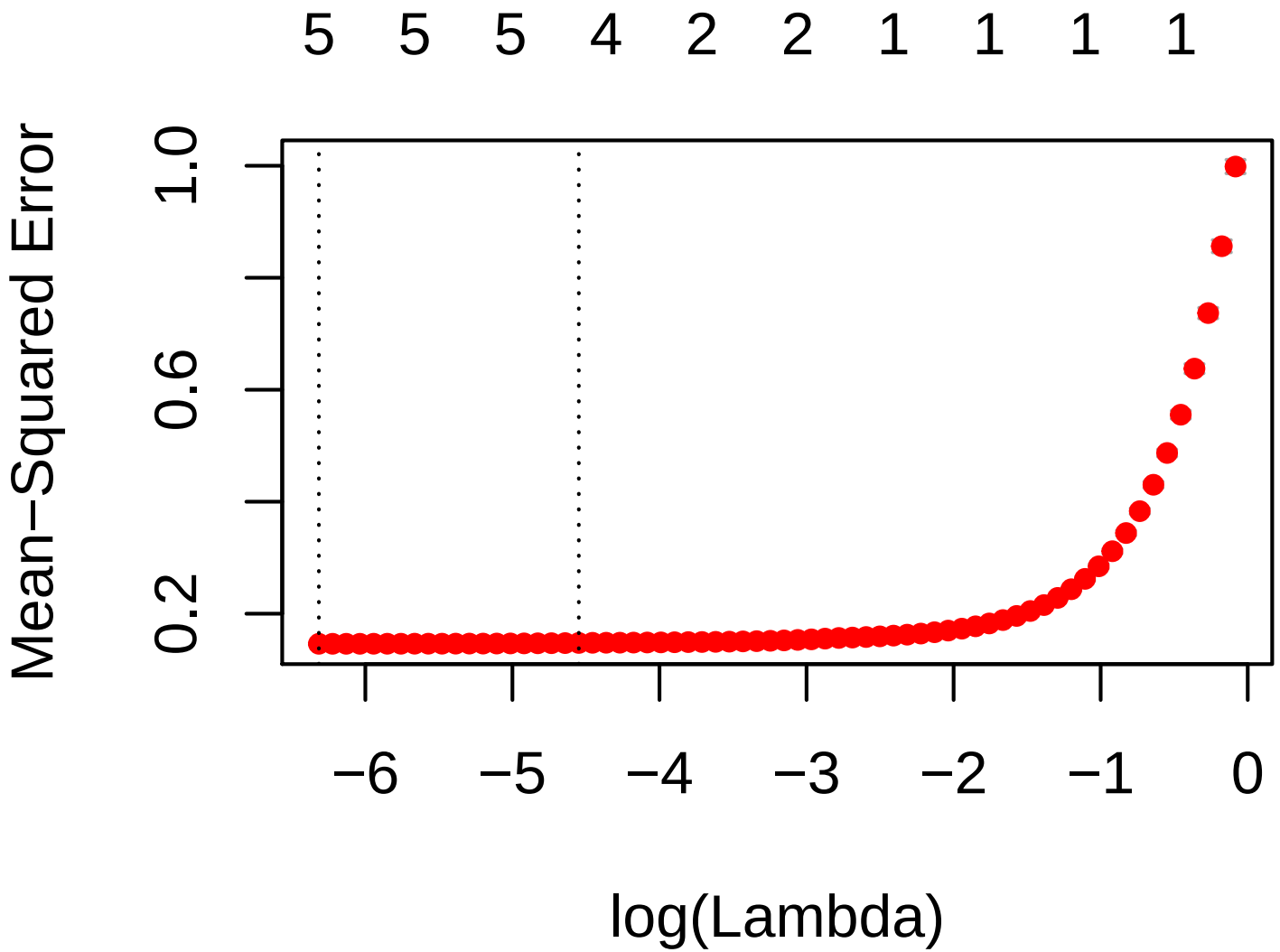
```
y <- train_std$price
x <- model.matrix(price ~ carat + cut, train_std)
price_l1 <- cv.glmnet(x, y, alpha=1)
```

- For L2-regularized model of `expensive` against `carat` and `cut`

```
y <- ifelse(train_std$expensive == 'yes', 1, 0)
x <- model.matrix(expensive ~ carat + cut, train_std)
expensive_l2 <-
    cv.glmnet(x, y, alpha=0, family='binomial')
```

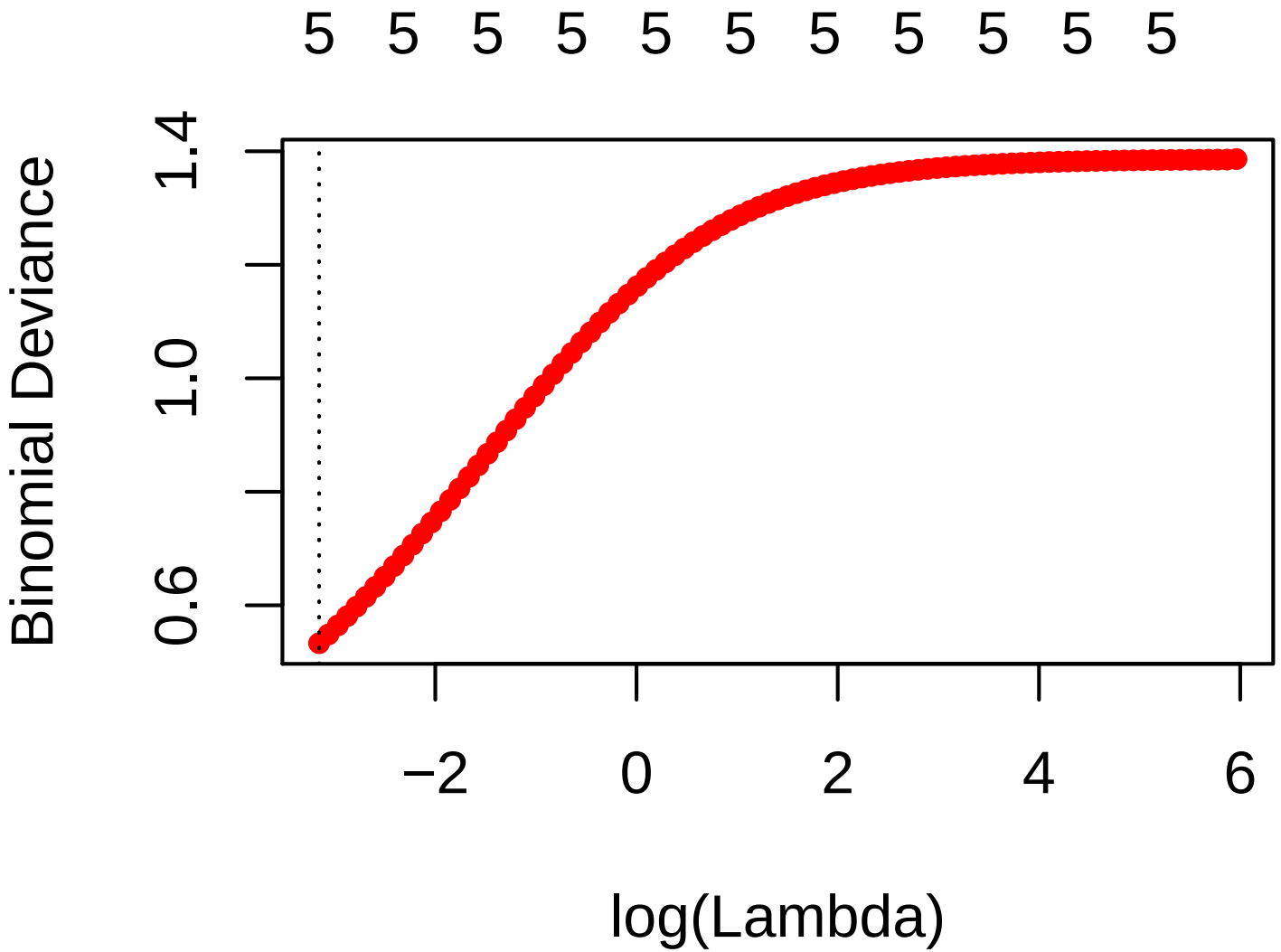
Regularized Linear Models: Example (plots)

```
plot(price_l1)
```



Regularized Linear Models: Example (plots)

```
plot(expensive_l2)
```



Random Forests

- The `randomForest` package in `R` implements Breiman's random forest algorithm, based on Breiman and Cutler's original Fortran code
- It can be used for classification, regression, or unsepervised learning of proximity between data points
- The basic syntax is

```
randomForest(x, y, ntree)
```

- `x`: the model matrix
- `y`: target variable vector. *Classification* if `y` is a factor, *regression* if `y` is otherwise, and *unsupervised* learning if `y` is omitted.
- **ntree**: number of trees to include in the forest
- While `randomForest` will also work for the `formula`/data frame syntax, a model matrix is considered more efficient

Random Forests: Examples

- To fit random forest of `price` against `carat` and `cut`

```
y <- train_std$price
x <- model.matrix(price ~ carat + cut, train_std)
rf_price <- randomForest(x, y, ntree=100)
```

- To fit a random forest `expensive` against `carat` and `cut`

```
y <- factor(train_std$expensive)
x <- model.matrix(expensive ~ carat + cut, train_std)
rf_expensive <- randomForest(x, y, ntree=100)
```

- `randomForest` is quite memory intensive
- `randomForest` classification aggregates *votes* of individual trees (as opposed to probability assessments), hence the floating-point precision is strictly determined by the order of `ntree`, i.e., $1/$`ntree`

Exercises

- Fit the following models, with `train_std` and `train_mm` where appropriate, from previous exercises
  - `lm_price`: OLS model of `price` against everything
  - `logit_exp`: Logistic regression of `expensive` against everything
  - `l1_price`: L1 regularized logistic regression of `price` against everything
  - `rf_exp`: Random forest classification model with 50 trees that uses `cut`, `color`, and `clarity` to predict `expensive`
- When I say "everything", I mean "everything but `expensive`" for `price`, and vice versa

Solution

```
lm_price <- lm(price ~ . - expensive, train_std)

train_std$expensive_num <-
    ifelse(train_std$expensive == 'yes', 1, 0)
logit_exp <-
    glm(expensive_num ~ . - price - expensive,
        train_std, family='binomial')

l1_price <-
    cv.glmnet(x=train_mm, y=train_std$price, alpha=1,
              standardize=FALSE, intercept=FALSE)

y <- factor(train_std$expensive)
x <- model.matrix(expensive ~ cut + color + clarity, train_
rf_exp <- randomForest(x, y, ntree=50)
```

Prediction/Evaluation

The Generic `predict()`

- Any decent modeling package in `R` will provide its own version of the generic `predict()` function
- The generic syntax is

```
predict(model_object, newdata, type)
```

- The `type` argument is often optional, and specifies the 'scale' of the predictions, e.g., probabilities v. log-odds for logistic regression models
- It's best (but not always necessary) for the `newdata` to have the same variables (columns) as the data that was used to train the model

The Generic predict(): Examples

- Using the models from the previous exercise, we can generate predictions for the test_std data

```
lm_price_pred <- predict(lm_price, test_std)

test_std$expensive_num <-
    ifelse(test_std$expensive == 'yes', 1, 0)

logit_exp_pred <- predict(logit_exp, test_std,
                          type='response')

l1_price_pred <- predict(l1_price, test_mm,
                         s='lambda.min')

rf_exp_pred <- predict(rf_exp, test_mm,
                       type='prob')
```

Example Notes

- The `s` argument for `predict.cv.glmnet` specifies the criteria for selecting a λ value from those that were fitted (see `?predict.cv.glmnet` for details)
- Beware of the column ordering for `rf_exp_pred` !

```
head(rf_exp_pred, n=2)
```

```
##      no  yes
## 1 0.06 0.94
## 2 0.00 1.00
```

Manual Predictions

- For linear models, generating predictions from new data should be as simple as matrix multiplication (with the help of `coef()`), e.g.,

```
manual_l1_price <-
    cbind(1, test_mm) %*%
    coef(l1_price, s='lambda.min')
all(manual_l1_price == l1_price_pred)
```

```
## [1] TRUE
```

- What might *not* be simple is creating the right matrix
- It's probably safe to stick with `predict()`, but make **sure** you read the documentation for new packages!!!!

Evaluation

- Once the predictions on the test set are generated, you should be able to compute any performance measure you're interested in

```
# RMSE for lm_price
sqrt(mean((test_std$price - lm_price_pred)^2))
```

```
## [1] 0.275
```

```
# RMSE for l1
sqrt(mean((test_std$price - l1_price_pred)^2))
```

```
## [1] 0.281
```

Evaluation (cont'd)

```
# Accuracy for logit_exp (for threshold 0.5)
logit_exp_vote <- ifelse(logit_exp_pred >= .5, 1, 0)
mean(logit_exp_vote == test_std$expensive_num)
```

```
## [1] 0.98
```

```
# Accuracy for rf_exp (for threshold 0.5)
rf_exp_vote <- ifelse(rf_exp_pred[,2] >= .5, 1, 0)
mean(rf_exp_vote == test_std$expensive_num)
```

```
## [1] 0.648
```

Evaluating Classifications

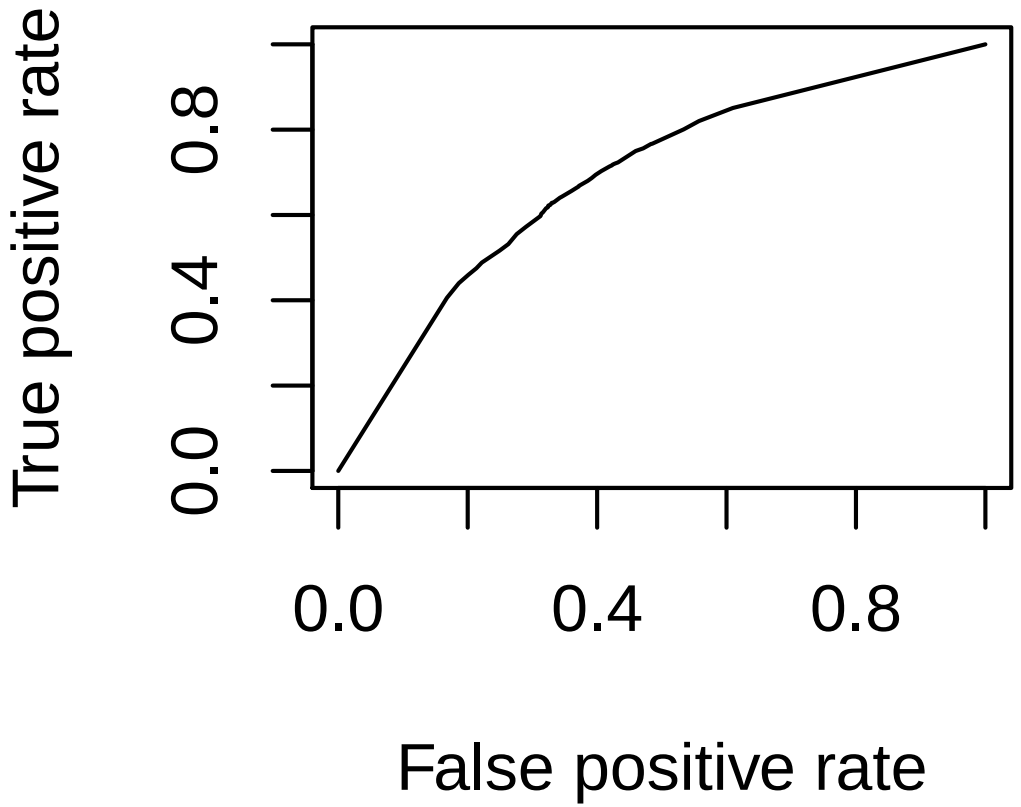
- The `ROCR` package can come in handy when evaluating classification models

```
library(ROCR)
pred <- prediction(rf_exp_pred[, 2],
                   test_std$expensive_num)
auc <- performance(pred, 'auc')
unlist(slot(auc, 'y.values'))
```

```
## [1] 0.689
```

Evaluating Classifications (cont'd)

```
perf <- performance(pred, 'tpr', 'fpr')
plot(perf)
```



- See website `https://rocr.bioinf.mpi-sb.mpg.de/` for details and more examples

Bootstrap

The Sledgehammer

- The actual implementation of generating bootstrap sampling distributions is super simple
- What might not be as simple is
  1. Truly understanding the theory
  2. Finding the right way of constructing measure of interest, e.g., confidence interval
  3. Implementing in a concise, reusable, and efficient way
- There are probably entire courses covering the first two
- People tend to gloss over the third
- I'll **only** focus on the third
- But keep in mind, knowing how to do it is very different from doing it right

Framework

1. Write a dummy function that
   - takes a single (dummy) argument
   - creates a single bootstrap sample from the original sample
   - returns the statistic of interest of the boostrap sample (as a scalar)
2. Use `boot <- sapply(1:B, dummy_function)` to create a bootstrap estimation of the sampling distribution
3. Compute the measure of interest (e.g., confidence interval) using the original sample and the `boot` vector created in (2)

- To save computation time, let's use a subsample of `diamonds` for the following examples

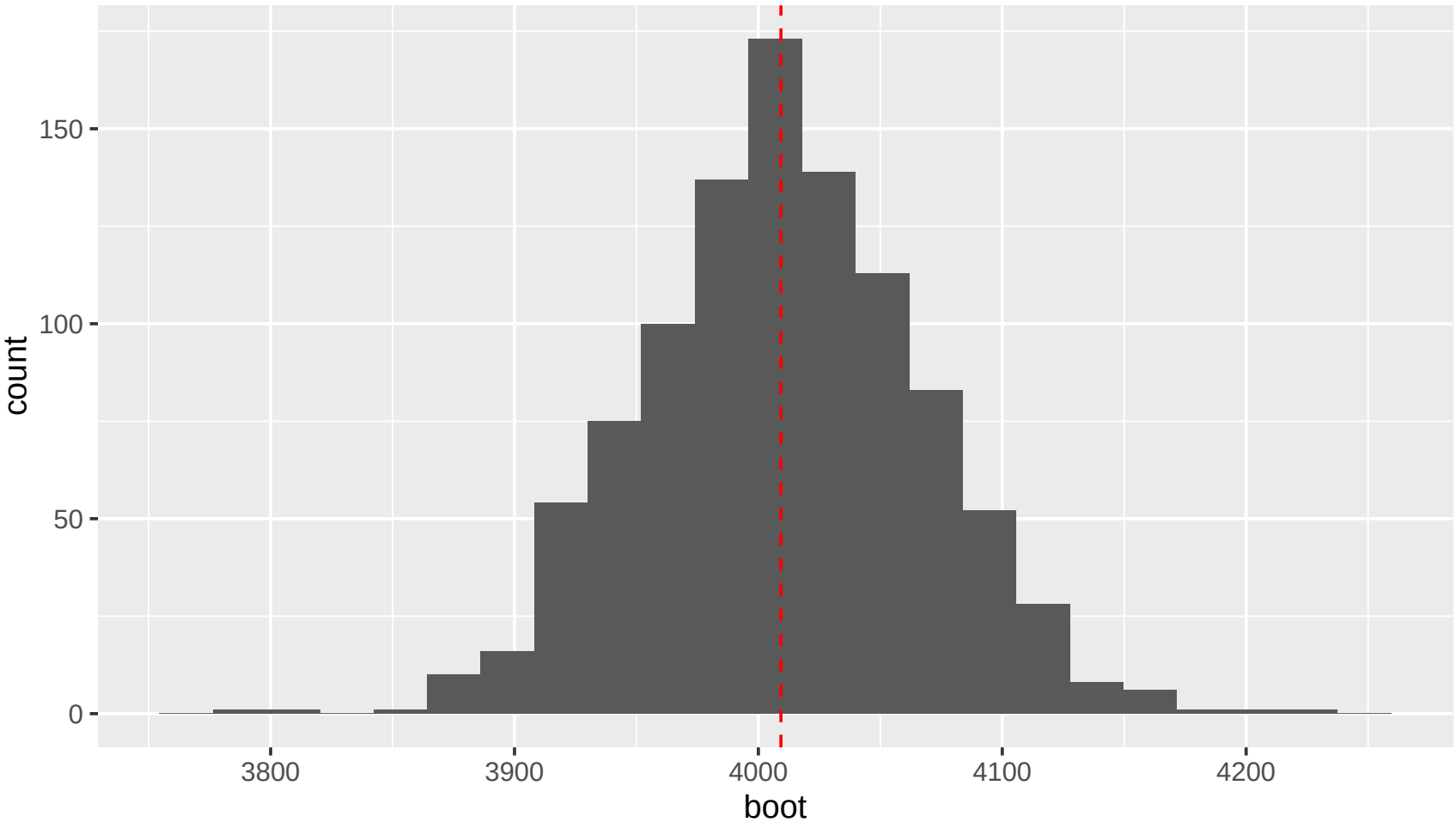
```
diamonds_small <- sample_n(diamonds, 5000)
```

Simple Example

- Construct a bootstrap estimate of the sampling distribution for the mean of `price` in the `diamonds` data, with $B = 1,000$

```
# 1. A dummy function
get_mean <- function(x) {
    mean(sample(diamonds_small$price,
                nrow(diamonds_small),
                replace=TRUE))
}
# 2. Creating the boot vector
boot <- sapply(1:1000, get_mean)
```

Histogram of boot and Original Point Estimate



Less Simple Example

- Construct a bootstrap estimate of the sampling distribution for the coefficient of `carat` for a OLS model fitting `price` to everything

```
# 1. A dummy function
get_coef <- function(x) {
    df <- sample_frac(diamonds_small, size=1,
                      replace=TRUE)
    fm <- lm(price ~ ., df)
    as.numeric(coef(fm)['carat'])
}
# 2. Creating the boot vector
boot <- sapply(1:1000, get_coef)
```

Histogram of boot and Original Point Estimate