SSCC Training Number 2 for ECON 690 Fall 2018

Mark Banghart

September 28, 2018

Preparations for today's training

- Set up a HW2 RStudio project in your econ 690 folder.
- Open a new Rmarkdown file (or R script) in the HW2 project to follow along with today's training. If you use an .Rmd file, you can write notes to go along with the examples from this training.
- Add library() calls for Tidyverse, glmnet, knitr, and Lahman. Note, you may have to install some of these packages.

```
library(tidyverse)
library(glmnet)
library(knitr)
library(Lahman)
```

- The examples will use the Batting, Salary, and Appearances data sets from the Lahman R package. These contain data on baseball players. Only data from the American and National leagues for the year 2000 and later will be used.

  The following code imports the data sets.

```
batting <-
  Batting %>%
  filter(yearID >= 2000, lgID %in% c("AL", "NL"))

salary <- Salaries

appear <-
  Appearances %>%
  select(yearID:playerID, G_p:G_rf)
```

  Note the use of the range of columns function (:) in the above code. The reference yearID:playerID is all the columns starting at yearID through and including playerID.

RMarkdown

- The source() function can be used to run code from an R script in a Rmarkdown file. Chunk options can be used to turn off the display of results. The results that are needed for the document can then be displayed where they are needed in the document with separate code chunks.
- Chunk options
  - **echo** controls the display of the R source code. When set to FALSE, the R source code is not included in the document. The formatting of R's results is not affected by this option.
  - **comment** controls what characters are displayed in front of R text results. When set to "", no characters are displayed.
  - **results** controls if text results are display and if they are formatted. When set to "hide", text results are not included in the document. When set to "asis", no formatting is done by knitr. This is useful for tables which have been formatted by R.

  - **message** controls the display of messages. When set to FALSE, messages are not included in the document.
  - **warning** controls the display of warning messages. When set to FALSE, warning messages are not included in the document.
  - **fig.show** controls if graphics are displayed. When set to "hide", graphics are not included in the document.

  The following is an example of a code chunck that does not display anything in the document.

  ```
  ```{r, echo=FALSE, results='hide', message=FALSE, warning=FALSE, fig.show='hide'}
  ```

- Use separate code block for each table and graph. This will make your document look a little cleaner.

  This is also allows for separate chunk options to be applied to each table and graph.

Manipulating data

Data sets often require preparation work to get them ready for use in your study. An example of this from the first homework assignment is the requirement that flights needed to be removed from markets that had fewer than 20 passengers a day. This could be done using the group_by() function to create market level groups. These groups could then be filtered (filter()) to remove markets with less than 20 passengers a day. The groups could then be removed (ungroup()) to restore the flight level organization of the data set. This is an example of what is known as the "split-apply-combine" paradigm. This is a common approach to solving programming problems.

This data preparation typically requires manipulating information across rows, columns, or some combination of the two.

- Working across rows (working on columns)

  R's native mode of operation is to work across the rows of a column. This is called vectorization in R. We saw a few examples of Tidyverse functions that work across rows in our first training class, mutate(), group_by(), and summarize().

  A few additional column oriented functions

  - arrange()

    Orders a data set based on the row or rows selected.

  - transmute()

    Creates new variables (columns) like mutate(). Unlike mutate, transmute discards all prior columns.

- Working across columns (working on rows)

  One approach to Working across columns is done by reshaping the data to put multiple columns into a single column. The data can then be grouped by the prior row identifier. The row wise manipulation can now be done as a column manipulation. All of R's native column functions can now be used. If needed, the data can be reshaped back to their original columns.

  - gather()

    Tidyverse function to combine multiple columns into a single column. The stacked columns organization is called long or tall form.

  - spread()

    Tidyverse function to divide a single column into multiple columns. The separate columns organization is called wide form.

  Example: create a variable that identifies the number of positions a players played in each year they played. The number of games a position was played is in the appear data set.
  ```

```
appear <-
  appear %>%
  gather(key = "var_name", value = "value", G_p:G_rf) %>%
  group_by(playerID, yearID) %>%
  mutate(multi_position = sum(value != 0)) %>%
  ungroup %>%
  spread(key = "var_name", value = "value")
```

Another approach is to use the pmap() set of functions (pmap, pmap_dbl, etc.).

- pmap(.l, .f)

  The .l parameter is a list of lists. Each row of the .l lists is used for a single call to .f function. The pmap() function returns a list of the results from the calls to .f.

- pmap_dbl(.l, .f)

  Works the same as pmap(). Returns a vector of type numeric.

Note, some functions may need the parameters from the .l list to be passed as a vector. The lift_vd() can be used to do this.

Example: Create a variable for the number of games a player played in each year. The number of games played at each positions is in the appear data set. Assume that a player played no more than one position in a game.

```
appear <-
  appear %>%
  mutate(total_games = pmap_dbl(select(., G_p:G_rf), sum))
```

Relational data

This section provides an introduction to working with related data sets. The data sets used in this section are two dimensional and will be referred to as tables to emphases the two dimensional nature of the data sets.

Data from different tables often needs to be combined into one table. Each table will have data which identifies how the data in the tables connect. These are called keys. The names of keys may be different in the tables.

The data between tables can be related in a variety of ways.

- One of these is known as one to one. In this relationship, one row in table A related to exactly one row in table B. There are no repeated values of a key in either table.
- Another common way data can be related is one to many. In this relationship, one of the tables has multiple rows that relate to a single row in the other table. Here one table has no repeated values of a key and the other table has repeated key values.

The Tidyverse provides a set of join functions that merge tables based on keys. The join functions are design along the lines of SQL joins.

To join tables, one starts by understanding what is in the tables and identifies the keys to be used.

- The key may be a single column of the tables or multiple columns. The key for a table may need to be constructed from other data in the tables.
- One tables may have multiple copies of the key. In this case, there may need to be more than one join.
- The keys are examined to determine if the join is one to one or one to many. If it is one to many, the many table is receiving table. That is the data from the table with no repeated keys is added to the table with the duplicated key values.

- An assessment of the other elements is also made to determine if the tables need to be manipulated before joining. There may be extra data in one or both of the tables. Multiple columns may need to be combined.

The following are a few of the Tidyverse join functions.

- left_join(A, B, by)

  The data in table B is joined to the data in table A. All rows in A are kept. Unmatched rows in B are discarded. The by parameter provides how the key in A is connected to the key in B. A may be piped in.

- right_join(A, B, by)

  Same as left_join() except all rows in B are kept and unmatched rows in B are discarded.

- full_join(A, B, by)

  Same as left_join() except all rows in both A and B are kept.

- inner_join(A, B, by)

  Same as left_join() except only rows that are in both A and B are kept.

It is good practice to check for missing keys in the tables being merged. This not needed for your current assignment and will be covered later.

Example: Create a baseball table that includes the information on batting with the data on salaries and the data on number of games played.

```
bb <-
  batting %>%
  left_join(salary,
            by = c("playerID" = "playerID",
                   "yearID" = "yearID",
                   "teamID" = "teamID"
            )
  )
```

```
## Warning: Column `teamID` joining factors with different levels, coercing to
## character vector
```

Note: This merge produced a warning. This is not an error. The merge completed successfully. Even though the merge worked, it is good practice to correct your code for warnings. In this case, the type of the variables can be corrected before the merge.

The following does the merge again without the warnings.

```
batting <- mutate(batting, teamID = as.character(teamID))
salary <- mutate(salary, teamID = as.character(teamID))
appear <- mutate(appear, teamID = as.character(teamID))
```

```
bb <-
  batting %>%
  left_join(salary,
            by = c("playerID" = "playerID",
                   "yearID" = "yearID",
                   "teamID" = "teamID"
            )
  )
```

The appear table has more columns that we need in merge. A new table is created with only the needed columns. the new table is then merged.

```
appear_stats <-
  appear %>%
  select(playerID, yearID, teamID, multi_position, total_games)

bb <-
  bb %>%
  left_join(appear_stats,
            by = c("playerID" = "playerID",
                   "yearID" = "yearID",
                   "teamID" = "teamID"
            )
  )
```

Could this have been done another way?

Yes. The right_join() function could have been used and this avoid the creation of the intermediate table for appear.

```
bb <-
  appear %>%
  select(playerID, yearID, teamID, multi_position, total_games) %>%
  right_join(bb,
             by = c("playerID" = "playerID",
                    "yearID" = "yearID",
                    "teamID" = "teamID"
             )
  )
```

Repeated operations

You already have seen functions that repeat operations across multiple elements of a structure, such as vectorized functions like sum() and group_by() and working on rows with pmap_dbl(). This section covers other methods for repeating operations.

The following are some additional approaches for repeating operations.

- map(), map_dbl()

  These are like pmap() except they take a single list. These are used in place of for loops.

- Defining new functions

  The syntax for defining a new function is listed below.

  ```
  function_name <- function(parameters) {

  }
  ```

  default parameter values are set using the equals sign in the parameter list.

  Example: Write a function to find the standard deviation of a vector of values.

  ```
  std_dev <- function(vec, std_err = TRUE) {
    if (!is.numeric(vec)) stop("vec must be numeric")
    if (length(vec) < 2) stop("vec must have at least two elements")
  ```

```
  if (std_err) {
    answ <- sum((vec - mean(vec))^2) / (length(vec) - 1)
  } else {
    answ <- sum((vec - mean(vec))^2) / length(vec)
  }
  sqrt(answ)
}

std_dev(1:5)
```

```
## [1] 1.581139
```

Does anyone see a parameter condition that was not checked for?

- for loop

  The syntax for defining a for loop is listed below.

```
for (*var_name* in *vector*) {

}
```

  Example: Write a loop to calculate the running sum of 1 through 5.

```
run_sum <- 0
for (i in 1:5) {
  run_sum <- run_sum + i
  print(paste0(i, " runing sum: ", run_sum))
}
```

```
## [1] "1 runing sum: 1"
## [1] "2 runing sum: 3"
## [1] "3 runing sum: 6"
## [1] "4 runing sum: 10"
## [1] "5 runing sum: 15"
```

OLS models

- Formulas

  Materials are here

- OLS Models
  - lm(formula, data, x)

    Constructs an OLS model. The data parameter is the data set that contains the variables for the regression.

    Set the x parameter to TRUE to get design matrix returned as part of the model object.

```
mod <- lm(salary ~ H + HR + RBI + multi_position + total_games, data = bb, x = TRUE)
```

  Do an str() on the mod object to see what lm() returned.

- summary()

```
summary(mod)
```

```
##
## Call:
```

```
## lm(formula = salary ~ H + HR + RBI + multi_position + total_games,
##     data = bb, x = TRUE)
##
## Residuals:
##       Min        1Q    Median        3Q       Max
## -10106270  -2212163  -1393414    937308  29951666
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)     2924354      74469  39.269  < 2e-16 ***
## H                 -3574       2012  -1.777 0.075641 .
## HR                80637      12476   6.463 1.06e-10 ***
## RBI               26793       6083   4.405 1.07e-05 ***
## multi_position  -481701      28224 -17.067  < 2e-16 ***
## total_games        4705       1231   3.821 0.000133 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 3991000 on 13346 degrees of freedom
##   (10214 observations deleted due to missingness)
## Multiple R-squared:  0.1172, Adjusted R-squared:  0.1169
## F-statistic: 354.4 on 5 and 13346 DF,  p-value: < 2.2e-16
```

The adj.r.squared value is one of the elements of the summary object. The summary object has a base type of list.

```
summary(mod)$adj.r.squared
```

```
## [1] 0.1168779
```

- predict(*mod*, *newdata*)

  Creates a vector of predicted values for the observations in *newdata*.

```
predict(mod, newdata = bb[1:5, ])
```

```
##       1       2       3       4       5
## 2362558 1681364 2600177 6639871 2734360
```

Some other useful model extractors functions

* coef()

```r
coef(mod)
```

```
##    (Intercept)              H             HR            RBI multi_position
##    2924353.823      -3574.317      80637.343      26793.139    -481701.189
##    total_games
##       4704.953
```

* BIC()

```r
BIC(mod)
```

```
## [1] 443842.8
```

* nobs()


```r
nobs(mod)
```

```
## [1] 13352
```

* Methods(class = \"*object*\")

Returns all the functions that accept an object of a certain class.


```r
methods(class = class(mod))
```

```
##  [1] add1           alias          anova          case.names
##  [5] coerce         confint        cooks.distance deviance
##  [9] dfbeta         dfbetas        drop1          dummy.coef
## [13] effects        extractAIC     family         formula
## [17] fortify        hatvalues      influence      initialize
## [21] kappa          labels         logLik         model.frame
## [25] model.matrix   nobs           plot           predict
## [29] print          proj           qqnorm         qr
## [33] residuals      rstandard      rstudent       show
## [37] simulate       slotsFromS3    summary        variable.names
## [41] vcov
## see '?methods' for accessing help and source code
```

Random sampling observations

There are many techniques to randomly sample from a data set. The sample() function is often used for doing this. With some calculations and sampling without replacement, random groups of fixed sizes can be produced. The method I use here results in groups sized to specified proportions of the data set. The group identifier is a factor variable.

- rnunif()

  A base R function that generated a random uniform distribution on $[0, 1]$.

- cut()

  A base R function that bins a vector based on cut points called breaks. It returns a factor variable.

- rank()

  A base R function that identifies the position a value would have in an ordered list.

Example: Divide the baseball data set into three sets. One for training, one for testing, and one for validation.

```
bb <-
  bb %>%
  mutate(rnum = runif(n()),
         rrank = rank(rnum),
         group = cut(rrank/n(), breaks = c(-1, .5, .75, 2), labels = c("train", "test", "valid"))
  ) %>%
  select(-rnum, -rrank)
```

glmnet package

- Lasso and Ridge models
  - glmnet(*x*, *y*, *family*, *alpha*, *lambda*)

    The glmnet functions do not use R's formula method of specifying models. The x parameter must be a matrix of type numeric.

    The family parameter's default value is "gaussian".

    The alpha parameter is used to specify lasso, ridge, or elasticnet. 1 is lasso, 0 is ridge, (0,1) is elasticnet.

    The lambda parameter can be defaulted to a set of lambda values picked by the function. It can also be a vector of lambda values to use.

    glmnet returns a set of coefficients for each lambda value used.

    The standardize parameter defaults to TRUE and this results in the columns of X being standardized prior to fitting the model. The coefficients are reported on the original scale.

  Example: Run a lasso regression on the baseball training data using the same regressor as with the OLS model above.

```
train <- filter(bb, group == "train")

mod_temp <- lm(salary ~ H + HR + RBI + multi_position + total_games, data = train, x = TRUE)
X <- mod_temp$x[,-1]
y <- mod_temp$model$salary

mod_lasso <- glmnet(X, y, alpha = 1)
```

- Cross validation of Ridge, Lasso, and Elasticnet
  - cv.glmnet(*x*, *y*, *family*, *alpha*, *lambda*, *nfold*)

    The parameters are the same as for glmnet() except for *nfold*. This parameter is used to specify the number of folds to use.

  Example: Find the coefficient values for the model that minimizes the mse.

```r
mod_cv_lasso <- cv.glmnet(X, y, alpha = 1, nfold = 10)

lambda_min <- mod_cv_lasso$lambda.min
coef(mod_cv_lasso, s = lambda_min)
```

```
## 6 x 1 sparse Matrix of class "dgCMatrix"
##                            1
## (Intercept)    2849042.688
## H                    .
## HR              100642.772
## RBI              13706.382
## multi_position -448542.596
## total_games       5920.723
```

The predict method is also supported for glmnet models. It will need the *s* parameter set with the value of lambda you want the predictions for.

Base R functions

These are a few other base R functions you may find useful.

- paste() and paste0()
- sum(), mean(), min(), max(), sqrt()
- ifelse()