Lecture 7

Eugen Buehler

October 30, 2016

Importing Data to R from a file

- CSV (comma seperated value)
- tab delimited files
- Excel formats (xls, xlsx)
- SPSS/SAS/Stata
- RStudio will tell you if you need to load packages

Problems to watch for after data import

- Data type (R may encode columns as strings that should be numbers)
- Missing values (how many are there and can we leave them be)

Fixing type problems after import

R will usually try to guess the type (numeric, charcacter, etc) that it should use to represent your data. But it makes mistakes! You can use the "as" functions to convert between many types. For example:

```r
myTable$column4 <- as.numeric(myTable$column4)
myTable$column3 <- as.character(myTable$column3)
myTable$column2 <- as.factor(myTable$column2)
```

Checking for missing values

Once importing a dataset, you can use "anyNA" to check if there are any missing values.

```r
titanic <- read.csv("test.csv")
anyNA(titanic)
```

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

Finding the distribution of missing values

Once you know there are missing values, you can start checking for where they occur. is.na will return TRUE if a value is NA and FALSE otherwise. Applied to a vector, it will return an equal sized vector of TRUE/FALSE. You can then "sum" that vector to see how many "NAs" you have.

```r
apply(titanic, 2, function(x) sum(is.na(x)))
```

```
## PassengerId      Pclass        Name         Sex         Age       SibSp 
##           0           0           0           0          86           0 
##       Parch      Ticket        Fare       Cabin    Embarked 
##           0           0           1           0           0
```

Excluding observations with missing values

If you have a lot of data, it might be OK to just exclude observations (rows) that having missing data.

```
fixedTitanic <- titanic[! apply(titanic,1, anyNA), ]
anyNA(fixedTitanic)
```

```
## [1] FALSE
```

Bear in mind that excluding rows with missing data could bias your results. What if ages were missing more for poor people than for rich people?

Imputing Missing Data

Imputing essentially means filling in missing values with educated guesses. The guesses could be based on models formed between the variables to try and predict the missing values, or could use a "k-nearest" neighbor approach.

Imputation is a big subject and there are numerous packages in R to do it. Just remember that if you choose to use imputed data, make sure that you can justify the method used and the underlying statistical assumptions, and that you never pass the imputed data off as "real" data.

Joining Tables

In the last lecture, you learned about the merge function for joining tables. If tables have the same format (same columns or the same numebr of rows), we can join them without having to merge.

```
newTable <- rbind(table1, table2)
newTable <- cbind(table1, table2)
```

Sometimes our two tables might not have exactly the same columns, in which case we could tell R to subset just on columns they both have.

```
commonColumns <- intersect(colnames(table1), colnames(table2))
rbind(table1[,commonColumns], table2[,commonColumns])
```

Aggregating Data

Frequently we want to summarize the data in a way that combines several observations into one. Examples would include take the avarge of all replicates, taking the average microarray readout for all probes against a given gene, etc. We can use the agregate function to do this.

```
meanWeightByFeed <- aggregate(weight ~ feed, data = chickwts, mean)
meanWeightByFeed <- aggregate(chickwts[,"weight"],
                              list(FeedType = chickwts$feed), mean)
```

Both versions of the above command will create a table of mean weights for each feed type. The first formula interface is easier to type when dealing with small numbers of things to aggregate. The second form can be convenient when we want to aggregate on a large set of columns. Not that here we specified the mean, but we could just as easily use any other function (median, sd, sum, etc)

Back to the reshape package

In the last lecture you learned about the "melt" function from the reshape2 package. There is a useful corresponding function in the package called "cast". Cast can be used to reshape the data that was melted, placing some information into columns and keeping others as rows.

```
ChickWeight[1:5,]
```

```
##   weight Time Chick Diet
## 1     42    0     1    1
## 2     51    2     1    1
## 3     59    4     1    1
## 4     64    6     1    1
## 5     76    8     1    1
```

Melted Chicks

First, let's melt the chick weight table

```
library(reshape2)
chick.melt <- melt(ChickWeight, id=2:4, na.rm=TRUE)
chick.melt[1:5,]
```

```
##   Time Chick Diet variable value
## 1    0     1    1   weight    42
## 2    2     1    1   weight    51
## 3    4     1    1   weight    59
## 4    6     1    1   weight    64
## 5    8     1    1   weight    76
```

Casting Melted Chicks

```r
chick.time <- dcast(chick.melt, Diet + Chick ~ Time)
chick.time[1:5,]
```

```
##   Diet Chick  0  2  4  6  8 10 12 14 16  18  20 21
## 1    1    18 39 35 NA NA NA NA NA NA NA  NA  NA NA
## 2    1    16 41 45 49 51 57 51 54 NA NA  NA  NA NA
## 3    1    15 41 49 56 64 68 68 67 68 NA  NA  NA NA
## 4    1    13 41 48 53 60 65 67 71 70 71  81  91 96
## 5    1     9 42 51 59 68 85 96 90 92 93 100 100 98
```

Notice the NAs! If each combination of Diet and Chick does not have a value for all possible times, it will be filled in with an NA!

Casting Melted Chicks 2

```
chick.diet <- dcast(chick.melt, Time + Chick ~ Diet)
chick.diet[1:5,]
```

```
##   Time Chick  1  2  3  4
## 1    0    18 39 NA NA NA
## 2    0    16 41 NA NA NA
## 3    0    15 41 NA NA NA
## 4    0    13 41 NA NA NA
## 5    0     9 42 NA NA NA
```

Casting Melted Chicks 3

```
chick.time <- dcast(chick.melt, Time ~ variable)
```

```
## Aggregation function missing: defaulting to length
```

```
chick.time[1:5,]
```

```
##   Time weight
## 1    0     50
## 2    2     50
## 3    4     49
## 4    6     49
## 5    8     49
```

Corrected cast with agregation method

```
chick.time <- dcast(chick.melt, Time ~ variable, fun.aggregate = mean)
chick.time[1:5,]
```

```
##   Time   weight
## 1    0 41.06000
## 2    2 49.22000
## 3    4 59.95918
## 4    6 74.30612
## 5    8 91.24490
```

Sample Workflow - Introduction

In the following slides, I'll demonstrate how these functions would be used together to merge, reshape, plot and analyze two replicate experiments with multiple data readouts. The ability to do this kind of analysis easily (once you know what you're doing) is what seperates R from Excel.

The data used in these examples is siRNA screening data. The data is "high content", meaning that computers generate multiple numeric features based on automated microscopy of each well of a 384 well plate.

Merging the data with rbind

Because our data sets have all of the same columns, we can use rbind to put them together. But before we do that, we create a new column in each called "replicate" so that we can keep track of where each data row came from.

```
kinome1 <- read.csv("kinome1.csv", stringsAsFactors=FALSE)
kinome2 <- read.csv("kinome2.csv", stringsAsFactors=FALSE)
kinome1$replicate <- "Rep1"
kinome2$replicate <- "Rep2"
kinome <- rbind(kinome1,kinome2)
```

Checking out the Data

Before melting the data, we need to make some intelligent decisions about what to keep. We can use View() or str() to take a look:

```
str(kinome[,1:10])
```

```
## 'data.frame':    4202 obs. of  10 variables:
##  $ X           : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ PLATE_ID    : chr  "R1001217" "R1001217" "R1001217" "R1001217" ...
##  $ VROW_IDX    : int  1 1 1 1 1 1 1 1 1 1 ...
##  $ VCOL_IDX    : int  1 10 11 12 13 14 15 16 17 18 ...
##  $ SEQ         : chr  "CCAUCUCCAGUUGUAUUUUtt" "CUCAAGAACUGUCAAGUAAtt" "GACCCA
##  $ SYMBOL      : chr  "CHKA" "GSK3B" "CKMT2" "GUK1" ...
##  $ LOCUSID     : int  1119 2932 1160 2987 1195 3055 1196 3098 1198 3099 ...
##  $ GENBANK     : chr  "NM_001277" "NM_002093" "NM_001825" "NM_000858" ...
##  $ NCGCIDR     : int  590581 590590 590591 590592 590593 590594 590595 59059
##  $ VPLATE_CATNUM: chr  "s223211" "s6239" "s3100" "s6357" ...
```

Checking out the Data 2

```
str(kinome[,11:25])
```

```
## 'data.frame':    4202 obs. of  15 variables:
##  $ PRODUCT_NAME                         : logi  NA NA NA NA NA NA ...
##  $ pass                                 : chr  "CCAUCUCCAGUUGUAUUUU" "CUCAAGAACU
##  $ guide                                : chr  "AAAATACAACTGGAGATGG" "TTACTTGACA
##  $ cells.raw                            : int  981 1020 1211 1098 373 530 989 94
##  $ cells.neg.norm                       : num  91.7 95.3 113.2 102.6 34.9 ...
##  $ nuclear.size.raw                     : num  519 533 524 530 529 ...
##  $ nuclear.size.neg.norm                : num  97.5 100.1 98.4 99.5 99.3 ...
##  $ nuclear.intensity.progerin.raw       : num  312 302 291 307 426 ...
##  $ nuclear.intensity.progerin.neg.norm: num  102.4 98.9 95.4 100.7 139.7 ...
##  $ nuclear.intensity.laminb1.raw        : num  211 206 184 179 239 ...
##  $ nuclear.intensity.laminb1.neg.norm : num  121 118 106 103 138 ...
##  $ integrated.intensity.gH2AX.raw       : num  8649 6969 7046 7278 7856 ...
##  $ integrated.intensity.gH2AX.neg.norm: num  105.9 85.3 86.3 89.1 96.2 ...
##  $ screen                               : chr  "rep1" "rep1" "rep1" "rep1" ...
##  $ replicate                            : chr  "Rep1" "Rep1" "Rep1" "Rep1" ...
```

Melting the Data

Let's keep the siRNA ID (so we know when we have different siRNAs against the same gene), the gene symbol, the replicate, and all of the measurements for each well.

```
kinome.melt <- melt(kinome[,c(10,6,25,14:23)])
```

```
## Using VPLATE_CATNUM, SYMBOL, replicate as id variables
```

```
kinome.melt[1:5,]
```

```
##   VPLATE_CATNUM SYMBOL replicate  variable value
## 1       s223211   CHKA      Rep1 cells.raw   981
## 2         s6239  GSK3B      Rep1 cells.raw  1020
## 3         s3100  CKMT2      Rep1 cells.raw  1211
## 4         s6357   GUK1      Rep1 cells.raw  1098
## 5         s3162   CLK1      Rep1 cells.raw   373
```

Data distribution for each readout

```
library(ggplot2)
ggplot(kinome.melt, aes(x=value)) + geom_density(alpha=0.5, color="blue") +
  facet_wrap(~ variable, scales = "free")
```

Data distribution per readout, color replicates

```
ggplot(kinome.melt, aes(x=value, color=replicate)) + geom_density(alpha=0.5) +
  facet_wrap(~ variable, scales = "free")
```

Casting the Kinome

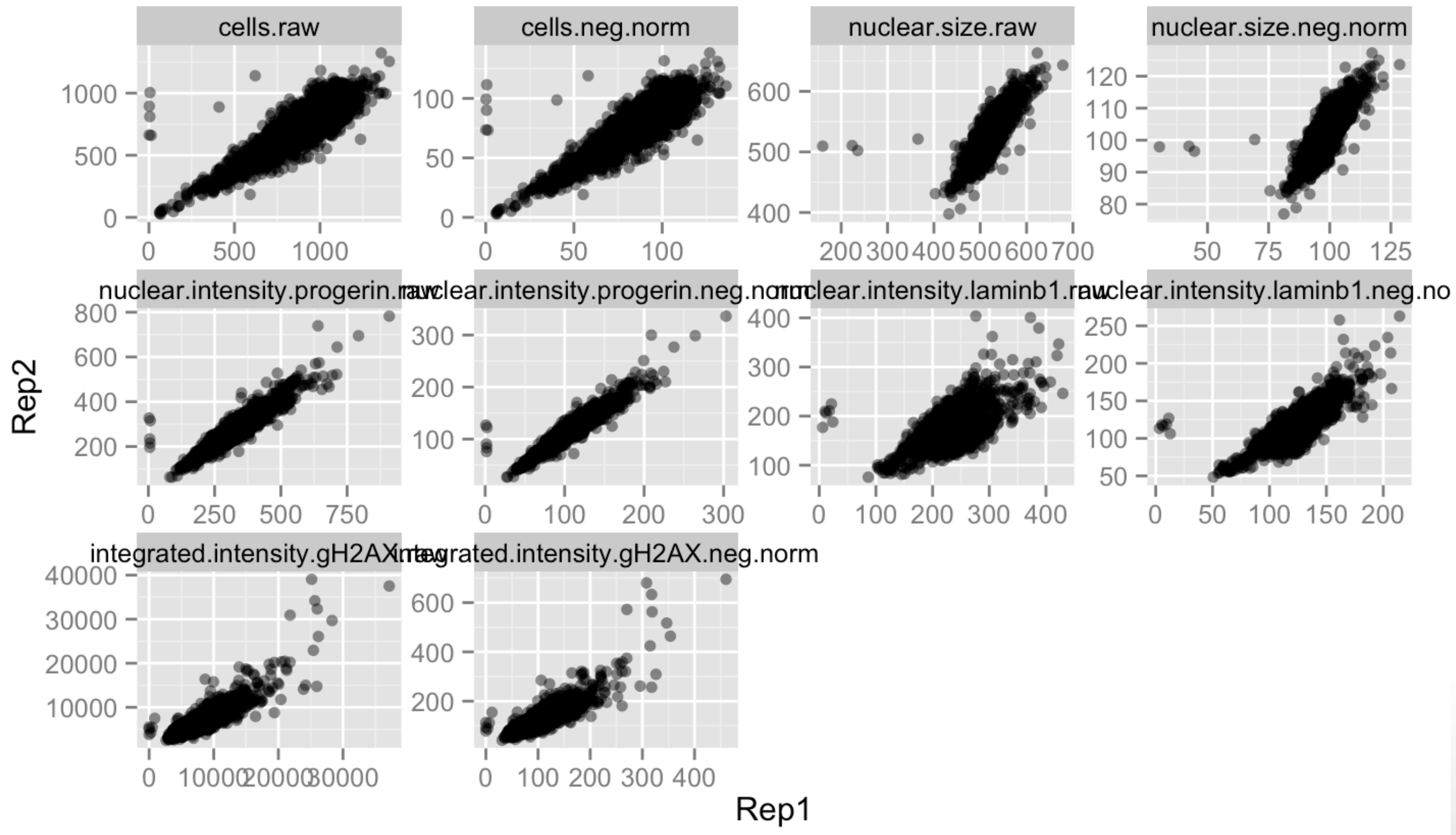
We would like to be able to plot the replicates opposite each other, but for that we need them in two seperate columns. Cast makes that easy.

```
kinome.reps <- dcast(kinome.melt, VPLATE_CATNUM + SYMBOL + variable ~ replicate)
kinome.reps[1:3,]
```

```
##   VPLATE_CATNUM SYMBOL         variable      Rep1     Rep2
## 1           s10 PDGFRL        cells.raw 829.00000 748.0000
## 2           s10 PDGFRL   cells.neg.norm  77.47664  78.1201
## 3           s10 PDGFRL nuclear.size.raw 563.04222 539.3930
```

Scatterplots of replicates

```
ggplot(kinome.reps, aes(x=Rep1, y=Rep2)) + geom_point(alpha=0.5) +
  facet_wrap(~ variable, scales="free")
```



Evaluating the Correlation of Replicates

```
by(kinome.reps, kinome.reps$variable,
   function(x) cor(x$Rep1, x$Rep2, method = "spearman"))
```

```
## kinome.reps$variable: cells.raw
## [1] 0.8632808
## --------------------------------------------------------
## kinome.reps$variable: cells.neg.norm
## [1] 0.853253
## --------------------------------------------------------
## kinome.reps$variable: nuclear.size.raw
## [1] 0.7990829
## --------------------------------------------------------
## kinome.reps$variable: nuclear.size.neg.norm
## [1] 0.7742904
## --------------------------------------------------------
## kinome.reps$variable: nuclear.intensity.progerin.raw
## [1] 0.9114601
## --------------------------------------------------------
```

Casting the Data for Collaborators (Part 1)

We can reform the data so that replicates of each readout are side by side in a table

```
kinome.output <- dcast(kinome.melt, VPLATE_CATNUM + SYMBOL ~ variable + replicate
kinome.output[1,]
```

```
##   VPLATE_CATNUM SYMBOL cells.raw_Rep1 cells.raw_Rep2 cells.neg.norm_Rep1
## 1           s10 PDGFRL            829            748            77.47664
##   cells.neg.norm_Rep2 nuclear.size.raw_Rep1 nuclear.size.raw_Rep2
## 1             78.1201              563.0422               539.393
##   nuclear.size.neg.norm_Rep1 nuclear.size.neg.norm_Rep2
## 1                   105.7754                   104.3789
##   nuclear.intensity.progerin.raw_Rep1 nuclear.intensity.progerin.raw_Rep2
## 1                            291.7604                            233.2458
##   nuclear.intensity.progerin.neg.norm_Rep1
## 1                                 95.63433
##   nuclear.intensity.progerin.neg.norm_Rep2
## 1                                 98.55344
```

Casting the Data for Collaborators (Part 2)

Alternatively, we can leave the replicate our of our melt formula and tell dcast what to do to merge the replicates (average them):

```
kinome.output2 <- dcast(kinome.melt, VPLATE_CATNUM + SYMBOL ~ variable, mean)
kinome.output2[1,]
```

```
##   VPLATE_CATNUM SYMBOL cells.raw cells.neg.norm nuclear.size.raw
## 1           s10 PDGFRL     788.5       77.79837         551.2176
##   nuclear.size.neg.norm nuclear.intensity.progerin.raw
## 1              105.0771                       262.5031
##   nuclear.intensity.progerin.neg.norm nuclear.intensity.laminb1.raw
## 1                            97.09388                      180.1997
##   nuclear.intensity.laminb1.neg.norm integrated.intensity.gH2AX.raw
## 1                           110.2928                       7356.978
##   integrated.intensity.gH2AX.neg.norm
## 1                            107.8141
```

Aggregating Further

If we want to take the average of the two replicates and then further aggregate to the gene level, we can:

```
kinome.output3 <- aggregate(kinome.output2[,3:12],
                            list(Symbol = kinome.output$SYMBOL), median)
kinome.output3[1,]
```

```
##   Symbol cells.raw cells.neg.norm nuclear.size.raw nuclear.size.neg.norm
## 1   AAK1     741.5       77.96853         532.8573                101.97
##   nuclear.intensity.progerin.raw nuclear.intensity.progerin.neg.norm
## 1                       246.9335                            92.48689
##   nuclear.intensity.laminb1.raw nuclear.intensity.laminb1.neg.norm
## 1                      210.0456                           119.6599
##   integrated.intensity.gH2AX.raw integrated.intensity.gH2AX.neg.norm
## 1                        8616.53                            119.1636
```

Exporting Data to other R users

If you want to share all of your work in R with someone, you can just give them the ".RData" file that will be automatically saved when you quit R. Or you can manually save to that file at any time with the "save.image" function.

```
save.image()
```

You can also save a subset of your work to an RData file to share with others, by listing the R objects you want to be stored in the file:

```
save(myTable, myList, myFunction, file="dataToShare.RData")
```

Your collaborator would then just use "load" to load the data file you provide:

```
load("dataToShare.RData")
```

Exporting Data for Excel (CSV)

Perhaps the most common platform for analyzing data is Excel. You could export data in CSV (comma seperated value format), which Excel can read.

```r
write.csv(myTable, file="myTable.csv", quote=FALSE, row.names=FALSE)
```

Note that the default values of TRUE for quote (whether you want quotes around all strings) and row.names (whether you want a column in the table with the row names) are probably not what you want.

Exporting Data for Excel (XLSX, XLS)

The problem with CSV is that is only allows you to export one table in a single file, where an Excel file can hold many tables (sheets), together with formatting information. There are several packages that allow you to read and write Excel files. I use the "xlsx" package:

```
write.xlsx2(geneLevelTable, file="myData.xlsx", sheetName="Gene Level")
write.xlsx2(siRNALevelTable, file="myData.xlsx",
            sheetName="siRNA Level", append = TRUE)
```

Note that this package relies on Java and can fail when presented with really large data due to limitations of Java's memory allocation. This package also has functions for formatting the data in the Excel file (changing the font style and background color of certain cells, for example).

Exporting Graphics

If you want to save a graph that you have generated in RStudio, there is an "Export" button that will allow you to save your results as a PDF. However, the pdf function gives us a little more power to do things like genearte multi-page graphs.

```
pdf(file = "histograms.pdf", width=10.5, height=8)
sapply(colnames(iris)[1:4], function (x) hist(iris[,x], main=x))
dev.off()
```

The pdf command creates a new graphical output which is a pdf file. The sapply command then generates four different histograms, one for each numeric column in the iris dataset. They will each be on a seperate page of the pdf. The "dev.off()" command closes the output and finishes creating the pdf. You won't be able to use the pdf until you run dev.off()!