Data Import and Export

Eugen Buehler

October 17, 2018

Importing Data to R from a file

- CSV (comma separated value)
- tab delimited files
- Excel formats (xls, xlsx)
- SPSS/SAS/Stata
- RStudio will tell you if you need to load packages
- This is not a complete list. If it is a popular file format, chances are there is a way for R to read it.

New and Improved with Tidyverse!

This lecture has been adapted from when we first gave it two years ago, to prefer using tidy syntax and import methods when possible. So first things first, we have to load the tidyverse packages!

```
library(tidyverse)
```

```
## ── Attaching packages ─────────────────────────────────────── tidyverse 1.2.1 ──
```

```
## ✔ ggplot2 2.2.1     ✔ purrr   0.2.5
## ✔ tibble  1.4.2     ✔ dplyr   0.7.5
## ✔ tidyr   0.8.1     ✔ stringr 1.3.1
## ✔ readr   1.1.1     ✔ forcats 0.3.0
```

```
## ── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
## ✖ dplyr::filter() masks stats::filter()
## ✖ dplyr::lag()    masks stats::lag()
```

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, character, 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:

```
myTable <- myTable %>% mutate(column4 = as.numeric(column4)) %>%
           mutate(column3 = as.character(column3)) %>%
           mutate(column2 = as.factor(column2))
```

Core R import functions vs. Tidy R

Tidy R import functions usually include an underscore, as opposed to a dot. The tidy R functions (from readr library and included in tidyverse) will produce a tibble, which will allow you to keep column names with their original characters and won't turn a column of characters into a factor by default.

```
myTitanicDataFrame <- read.csv("test.csv")
myTitanicTibble <- read_csv("test.csv")
```

```
## Parsed with column specification:
## cols(
##   PassengerId = col_integer(),
##   Pclass = col_integer(),
##   Name = col_character(),
##   Sex = col_character(),
##   Age = col_double(),
##   SibSp = col_integer(),
##   Parch = col_integer(),
##   Ticket = col_character(),
```

Checking for missing values

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

```
anyNA(myTitanicTibble)
```

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

Using Apply

The apply function allows you to apply a function 1 each row or column or a matrix or data frame. For data frames, it usually only makes sense to apply functions to each column (where everything has the same type)

```
apply(myTable, 2, sum)
apply(myMatrix, 1, mean)
```

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.

```
apply(myTitanicTibble, 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         327           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 <- myTitanicTibble[! apply(myTitanicTibble,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?

Excluding missing value rows (the tidy way)

```
fixedTitanic <- myTitanicTibble %>% drop_na()
anyNA(fixedTitanic)
```

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

If you only want to filter out NAs in certain columns, you can do that by supplying the column names to drop_na().

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

Recall that if tables have the same format (same columns or the same number of rows), we can join them easily.

```
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 (with TidyR)

```
meanWeightByFeed <- chickwts %>%
                    group_by(feed) %>%
                    summarize(meanWeight = mean(weight))
meanWeightByFeed
```

```
## # A tibble: 6 x 2
##   feed      meanWeight
##   <fct>          <dbl>
## 1 casein          324.
## 2 horsebean       160.
## 3 linseed         219.
## 4 meatmeal        277.
## 5 soybean         246.
## 6 sunflower       329.
```

Aggregating Data (with aggregate)

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

The reshape package

We will now use the "melt" and "cast" functions from the "reshape" package to manipulate the ChickWeight data frame.

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

```
## Grouped Data: weight ~ Time | Chick
##   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)
```

```
##
## Attaching package: 'reshape2'
```

```
## The following object is masked from 'package:tidyr':
##
##     smiths
```

```
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
```

Casting Melted Chicks

```
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
```

Reshape vs. TidyR's gather and spread

In Tidy R, gather does much the same thing as melt and spread does something very similar to cast. In general, you will find it much easier to use the Tidy R functions. Occasionally, reshape will allow you to have more power when you want to spread things out using multiple variables (via cast).

```
ChickWeight %>% as.tibble() %>%
    spread(Time, weight) %>%
    head()
```

```
## # A tibble: 6 x 14
##   Chick Diet    `0`   `2`   `4`   `6`   `8`  `10`  `12`  `14`  `16`  `18`
##   <ord> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
## 1 18    1        39    35    NA    NA    NA    NA    NA    NA    NA    NA
## 2 16    1        41    45    49    51    57    51    54    NA    NA    NA
## 3 15    1        41    49    56    64    68    68    67    68    NA    NA
## 4 13    1        41    48    53    60    65    67    71    70    71    81
## 5 9     1        42    51    59    68    85    96    90    92    93   100
```

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 separates 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")
```

```
## Warning: Missing column names filled in: 'X1' [1]
```

```
## Parsed with column specification:
## cols(
##   .default = col_character(),
##   X1 = col_integer(),
##   VROW_IDX = col_integer(),
##   VCOL_IDX = col_integer(),
##   LOCUSID = col_integer(),
##   NCGCIDR = col_integer(),
##   cells.raw = col_integer(),
##   cells.neg.norm = col_double(),
```

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])
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    4202 obs. of  10 variables:
##  $ X1           : 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" "GACCCACGC
##  $ 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 590596 59
##  $ VPLATE_CATNUM: chr  "s223211" "s6239" "s3100" "s6357" ...
```

Checking out the Data 2

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

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    4202 obs. of  15 variables:
##  $ PRODUCT_NAME                          : chr  NA NA NA NA ...
##  $ pass                                  : chr  "CCAUCUCCAGUUGUAUUUU" "CUCAAGAACUGUC
##  $ guide                                 : chr  "AAAATACAACTGGAGATGG" "TTACTTGACAGTT
##  $ cells.raw                             : int  981 1020 1211 1098 373 530 989 944 7
##  $ 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

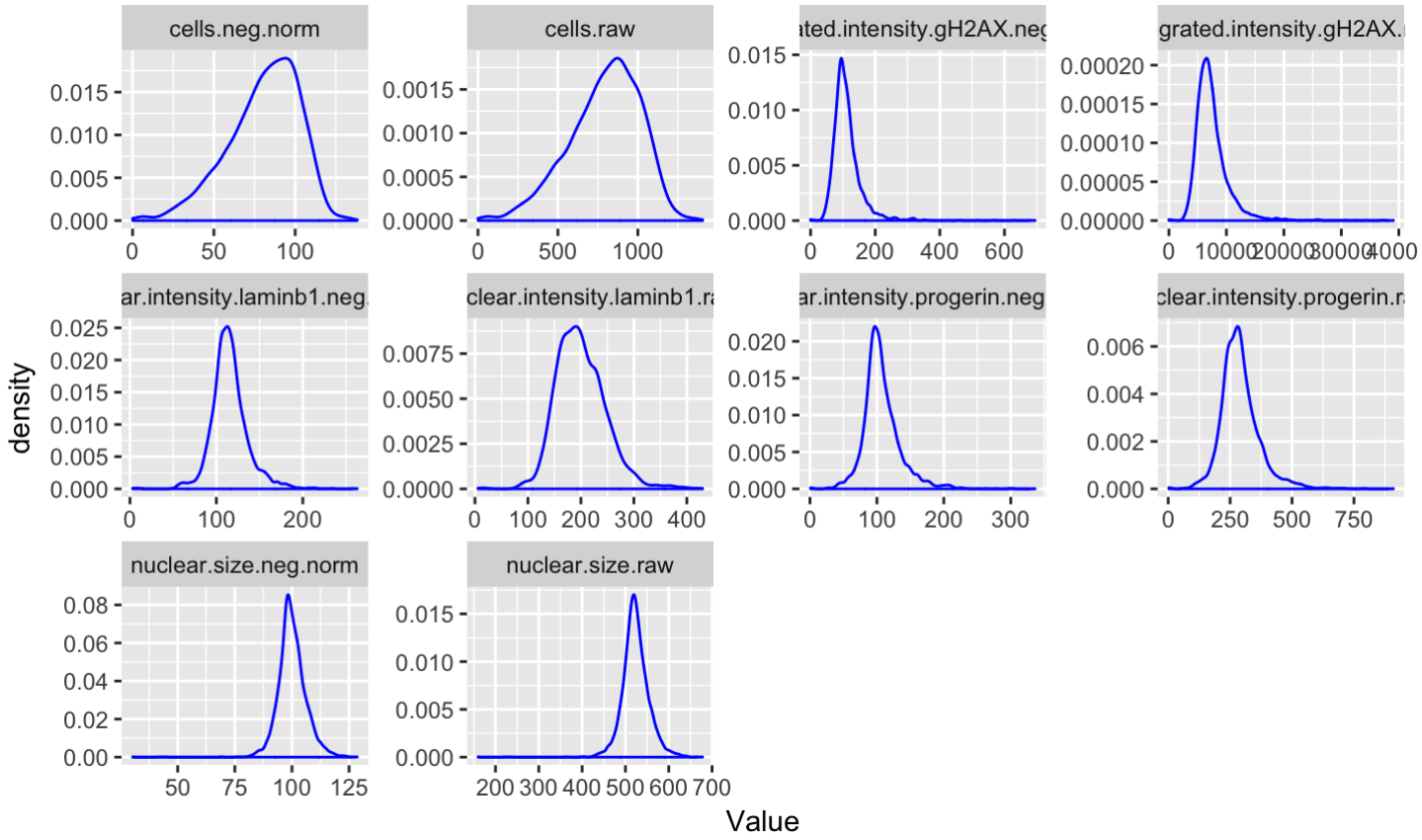
Let's keep the siRNA ID (so we know when we have different siRNAs against the same gene), the gene symbol, the replicate, and all of the measurements for each well.

```
kinome.tidy <- kinome %>%
  select(VPLATE_CATNUM,
         SYMBOL,replicate,
         cells.raw:integrated.intensity.gH2AX.neg.norm) %>%
  gather(Readout, Value, cells.raw:integrated.intensity.gH2AX.neg.norm)
kinome.tidy %>% head()
```

```
## # A tibble: 6 x 5
##   VPLATE_CATNUM SYMBOL replicate Readout   Value
##   <chr>         <chr>  <chr>     <chr>     <dbl>
## 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
## 6 s6479         HCK    Rep1      cells.raw   530
```

Data distribution for each readout

```
library(ggplot2)
ggplot(kinome.tidy, aes(x=Value)) + geom_density(alpha=0.5, color="blue") +
  facet_wrap(~ Readout, scales = "free")
```



Data distribution per readout, color replicates

```
ggplot(kinome.tidy, aes(x=Value, color=replicate)) + geom_density(alpha=0.5) +
  facet_wrap(~ Readout, scales = "free")
```

Spreading out the kinome

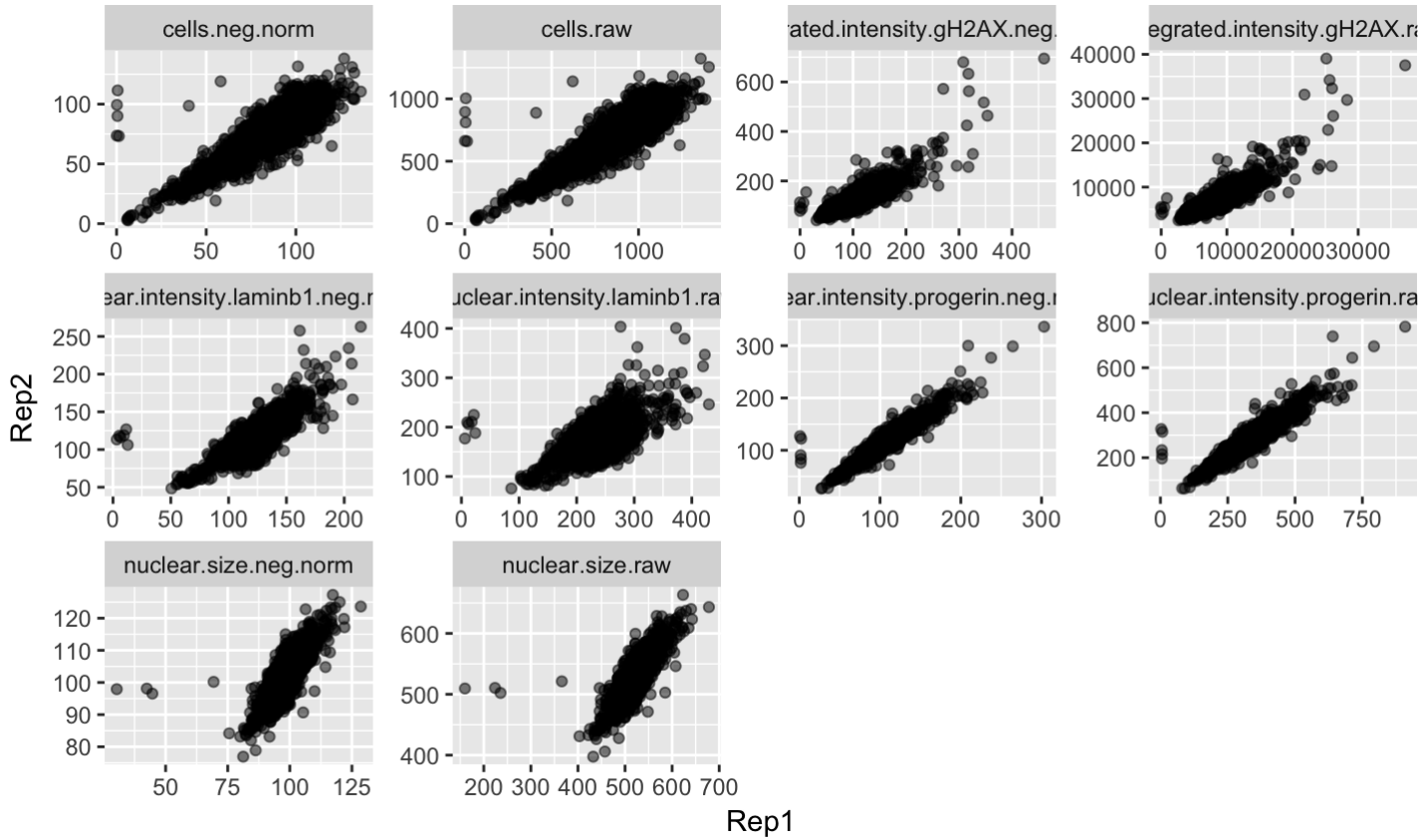
We would like to be able to plot the replicates opposite each other, but for that we need them in two separate columns.

```
kinome.reps <- kinome.tidy %>% spread(replicate, Value)
kinome.reps[1:3,]
```

```
## # A tibble: 3 x 5
##   VPLATE_CATNUM SYMBOL Readout                              Rep1  Rep2
##   <chr>         <chr>  <chr>                               <dbl> <dbl>
## 1 s10           PDGFRL cells.neg.norm                       77.5  78.1
## 2 s10           PDGFRL cells.raw                           829   748
## 3 s10           PDGFRL integrated.intensity.gH2AX.neg.norm  96.2 119.
```

Scatterplots of replicates

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



Evaluating the Correlation of Replicates

```
kinome.reps %>%
  group_by(Readout) %>%
  summarize(Correlation = cor(Rep1, Rep2, method = "spearman")) %>%
  head()
```

```
## # A tibble: 6 x 2
##   Readout                                Correlation
##   <chr>                                        <dbl>
## 1 cells.neg.norm                               0.853
## 2 cells.raw                                    0.863
## 3 integrated.intensity.gH2AX.neg.norm          0.835
## 4 integrated.intensity.gH2AX.raw               0.846
## 5 nuclear.intensity.laminb1.neg.norm           0.763
## 6 nuclear.intensity.laminb1.raw                0.523
```

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. This is an example of where "cast" will give you more flexibility than "spread".

```
kinome.output <- dcast(kinome.tidy, VPLATE_CATNUM + SYMBOL ~ Readout + replicate)
```

```
## Using Value as value column: use value.var to override.
```

```
kinome.output[1,]
```

```
##   VPLATE_CATNUM SYMBOL cells.neg.norm_Rep1 cells.neg.norm_Rep2
## 1           s10 PDGFRL             77.47664              78.1201
##   cells.raw_Rep1 cells.raw_Rep2 integrated.intensity.gH2AX.neg.norm_Rep1
## 1            829            748                                 96.17158
##   integrated.intensity.gH2AX.neg.norm_Rep2
## 1                                 119.4565
##   integrated.intensity.gH2AX.raw_Rep1 integrated.intensity.gH2AX.raw_Rep2
## 1                             7853.08                             6860.876
##   nuclear.intensity.laminb1.neg.norm_Rep1
```

Casting the Data for Collaborators (Part 2)

Alternatively, we can leave the replicate out by taking the mean of the Readouts before we spread.

```
kinome.output2 <- kinome.tidy %>%
                  group_by(VPLATE_CATNUM, SYMBOL, Readout) %>%
                  summarize(Value = mean(Value)) %>%
                  spread(Readout, Value)
kinome.output2[1,]
```

```
## # A tibble: 1 x 12
## # Groups:   VPLATE_CATNUM, SYMBOL [1]
##   VPLATE_CATNUM SYMBOL cells.neg.norm cells.raw integrated.intensity.gH2A…
##   <chr>         <chr>           <dbl>     <dbl>                      <dbl>
## 1 s10           PDGFRL           77.8      788.                       108.
## # ... with 7 more variables: integrated.intensity.gH2AX.raw <dbl>,
## #   nuclear.intensity.laminb1.neg.norm <dbl>,
## #   nuclear.intensity.laminb1.raw <dbl>,
## #   nuclear.intensity.progerin.neg.norm <dbl>,
## #   nuclear.intensity.progerin.raw <dbl>, nuclear.size.neg.norm <dbl>,
## #   nuclear.size.raw <dbl>
```

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 <- kinome.tidy %>%
                  group_by(VPLATE_CATNUM, SYMBOL, Readout) %>%
                  summarize(Value = mean(Value)) %>%
                  group_by(SYMBOL, Readout) %>%
                  summarize(Value = median(Value)) %>%
                  spread(Readout, Value)
kinome.output3[1,]
```

```
## # A tibble: 1 x 11
## # Groups:   SYMBOL [1]
##   SYMBOL cells.neg.norm cells.raw integrated.intensity… integrated.intens…
##   <chr>           <dbl>     <dbl>                 <dbl>              <dbl>
## 1 AAK1             78.0      742.                  119.               8617.
## # ... with 6 more variables: nuclear.intensity.laminb1.neg.norm <dbl>,
## #   nuclear.intensity.laminb1.raw <dbl>,
## #   nuclear.intensity.progerin.neg.norm <dbl>,
```

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 Individual R Objects

```
saveRDS(myTable, file="myTable.rds")
# some other time...
thatTable <- readRDS("myTable.rds")
```

Exporting Data for Excel (CSV)

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

```
myTable %>% write_csv(file="myTable.csv")
```

Note that write_csv is faster than write.csv and has better default behavior (like not writing row names).

Exporting Data Using Rio

Rio is a very handy package (short for R input/output). A very nice feature of rio is the export function, which will figure out what kind of file you want to export based on the file extension that you use.

```
library(rio)
myTable %>% export("myTable.xlsx") # writes an Excel file
myTable %>% export("myTable.csv") # writes a CSV file
myGeneInfo %>% export("myTable.xlsx", which="Genes")
mySirnaInfo %>% exort("myTable.xlsx", which="Sirnas", overwrite=FALSE)
```

Rio let's you forget largely about what package is doing the work. It can save things in Matlab format, SAS, JSON, XML, SPSS, and more. ## 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 generate multi-page graphs.