Data–Handling–lab

Bernd Klaus[1]

European Molecular Biology Laboratory (EMBL),
Heidelberg, Germany
[1]bernd.klaus@embl.de

March 19, 2016

Contents

1 Required packages and other preparations

```
library(TeachingDemos)
library(openxlsx)
library(multtest)
library(Biobase)
library(tidyr)
library(reshape2)
library(plyr)
library(dplyr)
library(ggplot2)
library(stringr)
library(magrittr)
library(purrr)
library(readr)
library("DESeq2")
library("pasilla")
library("Biobase")
data("pasillaGenes")

countData <- counts(pasillaGenes)
colData <- pData(pasillaGenes)[,c("condition","type")]

pasilla_se <- DESeqDataSetFromMatrix(countData = countData,
                                     colData = colData,
                                     design = ~ condition)

library("TxDb.Dmelanogaster.UCSC.dm3.ensGene")
```

2 Introduction

R offers a wide range of data manipulation tools that allow you to handle, compute on and reshape data very efficiently efficiently. We first review basic data handling techniques. Many of the techniques discussed in this lab are very well elaborated on in the upcoming book "R for data science" available on this website.

3 Review of data handling with *R*

In this section, we want to look at some basic data handling techniques, e.g. subsetting it or combining data from different sources.

3.1 Review of filtering and access techniques

This section reviews some basic data access techniques in *R*. Let's assume we have a very simple vector with named elements:

```
sampleVector <- c("Alice" = 5.4, "Bob" = 3.7, "Claire" = 8.8)
sampleVector
```

```
#>  Alice    Bob Claire
#>    5.4    3.7    8.8
```

3.1.1 Access by index

The simplest way to access the elements in a vector is via their indices. Specifically, you provide a vector of indices to say which elements from the vector you want to retrieve. A minus sign excludes the respective positions

```
sampleVector[1:2]
```

```
#> Alice   Bob
#>   5.4   3.7
```

```
sampleVector[-(1:2)]
```

```
#> Claire
#>    8.8
```

3.1.2 Access by boolean

If you generate a boolean vector the same size as your actual vector you can use the positions of the true values to pull out certain positions from the full set. You can also use smaller boolean vectors and they will be concatenated to match all of the positions in the vector, but this is less common.

```
sampleVector[c(TRUE, FALSE, TRUE)]
```

```
#>  Alice Claire
#>    5.4    8.8
```

```
# or
subset(sampleVector, c(TRUE, FALSE, TRUE))
```

```
#>  Alice Claire
#>    5.4    8.8
```

The `subset` functions is a more general function for subsetting, which also works on more complex objects.

This can also be used in conjunction with logical tests which generate a boolean result. Boolean vectors can be combined with logical operators (& and) to create more complex filters.

```
sampleVector[sampleVector < 6]
```

```
#> Alice   Bob
#>   5.4   3.7
```

```
# or
subset(sampleVector, sampleVector < 6 | names(sampleVector) == "Bob")
```

```
#> Alice   Bob
#>   5.4   3.7
```

3.1.3 Multidimensional data structures

Very often, the data structure you are looking at has a multidimensional structure, e.g. is data.frame or a list. Here, access works in exactly the same way but in two dimensions. We use a small patients data set as an example. We use the function `read_csv` from the *readr* package that provides consistent and fast data import functions that do not depend on your locale settings and option such as stringsAsFactors.

```
pat<-read_csv("http://www-huber.embl.de/users/klaus/BasicR/Patients.csv")
pat
```

```
#> Source: local data frame [3 x 4]
#>
#>   PatientId Height Weight Gender
#>       (chr)  (dbl)  (dbl)  (chr)
#> 1        P1   1.65     80      f
#> 2        P2   1.30     NA      m
#> 3        P3   1.20     50      f
```

```
pat[1,c(1:3)]
```

```
#> Source: local data frame [1 x 3]
#>
#>   PatientId Height Weight
#>       (chr)  (dbl)  (dbl)
#> 1        P1   1.65     80
```

```
pat["P1",]
```

```
#> Source: local data frame [1 x 4]
#>
#>   PatientId Height Weight Gender
#>       (chr)  (dbl)  (dbl)  (chr)
#> 1        NA     NA     NA     NA
```

There are additional access options for data frames and lists available (a data frame is always a list). You can use the dollar sign to access a column of a data frame and an element of a list or use the double bracket operator.

```
pat$"Weight"
```

```
#> [1] 80 NA 50
```

```
pat[["Weight"]][3]
```

```
#> [1] 50
```

```
# often acces via the dollar sign works without quotes (but for numbers!)
pat$Height
```

```
#> [1] 1.65 1.30 1.20
```

3.2 Subsetting and information extraction with a data table

We will illustrate subsetting and extraction techniques using a typical data table – a data set on variables influencing your body fat:

A variety of popular health books suggest that the readers assess their health, at least in part, by estimating their percentage of body fat. We will illustrate the techniques using the data set "bodyfat", which contains variables that could be used to build models predictive of body fat:

- Density determined from underwater weighing
- Percent body fat from Siri's (1956) equation
- Age (years)
- Weight (lbs)
- Height (inches)
- Neck circumference (cm)
- Chest circumference (cm)
- Abdomen 2 circumference (cm)
- Hip circumference (cm)
- Thigh circumference (cm)
- Knee circumference (cm)
- Ankle circumference (cm)
- Biceps (extended) circumference (cm)
- Forearm circumference (cm)
- Wrist circumference (cm)

First, we import the data set and inspect it a bit.

```
load(url("http://www-huber.embl.de/users/klaus/BasicR/bodyfat.rda"))
dim   (bodyfat)    # how many rows and columns in the dataset?
```

```
#> [1] 252  15
```

```
names (bodyfat)    # names of the columns
```

```
#>  [1] "density"        "percent.fat"    "age"            "weight"
#>  [5] "height"         "neck.circum"    "chest.circum"   "abdomen.circum"
#>  [9] "hip.circum"     "thigh.circum"   "knee.circum"    "ankle.circum"
#> [13] "bicep.circum"   "forearm.circum" "wrist.circum"
```

To get a first impression of the data, we can compute some summary statistics for e.g. age. We can get all these statistics for all the data at once by using an appropriate `apply` command.

```
## compute descriptive statistics for "age"
summary(bodyfat$age)
```

```
#>    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#>    22.0    35.8    43.0    44.9    54.0    81.0
```

```
sd(bodyfat$age)
```

```
#> [1] 12.6
```

```
mean(bodyfat$age)
```

```
#> [1] 44.9
```

```
IQR(bodyfat$age)/1.349
```

```
#> [1] 13.5
```

```
## mean value of every variable in the bodyfat data set
apply(bodyfat, MARGIN = 2, FUN = mean)
```

```
#>        density    percent.fat            age         weight         height
#>           1.06          19.15          44.88         178.92          70.15
#>    neck.circum   chest.circum abdomen.circum     hip.circum   thigh.circum
#>          37.99         100.82          92.56          99.90          59.41
#>    knee.circum   ankle.circum   bicep.circum forearm.circum   wrist.circum
#>          38.59          23.10          32.27          28.66          18.23
```

```
## alternative : sapply(bodyfat, FUN = mean)
```

Very often you want to access a certain subset, say all samples with age between 40 and 60 and height between 50 and 65 inches. This can be done by using logical operators in combination with the variables. This then evaluates to `TRUE` or `FALSE` for every sample. The corresponding indices that evaluate to `TRUE` can be obtained via the function `which`:

```
## all samples with age between 40 and 60 and height
##between 50 and 65
bodyfat[  bodyfat$age > 40 & bodyfat$age < 60 & bodyfat$height > 50 & bodyfat$height < 65, ]
```

```
#>     density percent.fat age weight height neck.circum chest.circum abdomen.circum
#> 74    1.068        13.5  55    125     64        33.2         87.7             76
#> 216   0.995        47.5  51    219     64        41.2        119.8            122
#>     hip.circum thigh.circum knee.circum ankle.circum bicep.circum forearm.circum
#> 74        88.6         50.9        35.4         19.1         29.3           25.7
#> 216      112.8         62.5        36.9         23.6         34.7           29.1
#>     wrist.circum
#> 74          16.9
#> 216         18.4
```

```
### get the corresponding indices
which( bodyfat$age > 40 & bodyfat$age < 60 & bodyfat$height > 50 & bodyfat$height < 65)
```

```
#> [1]  74 216
```

```
### and samples
bodyfat[  which( bodyfat$age > 40 & bodyfat$age < 60 & bodyfat$height > 50 & bodyfat$height < 65)
, ]
```

```
#>     density percent.fat age weight height neck.circum chest.circum abdomen.circum
#> 74    1.068        13.5  55    125     64        33.2         87.7             76
#> 216   0.995        47.5  51    219     64        41.2        119.8            122
#>     hip.circum thigh.circum knee.circum ankle.circum bicep.circum forearm.circum
#> 74        88.6         50.9        35.4         19.1         29.3           25.7
#> 216      112.8         62.5        36.9         23.6         34.7           29.1
#>     wrist.circum
#> 74          16.9
#> 216         18.4
```

However, there is a certain subtle side effect using `which`. It tends you have unintended consequences if all elements of the vector of booleans you're querying are `FALSE`. Then `which()` will return no indices. Therefore you should use which with care when accessing subsets of your data!

```
max(bodyfat$age)
```

```
#> [1] 81
```

```
# 81
head(bodyfat[  !(bodyfat$age  > 81), ])
```

```
#>   density percent.fat age weight height neck.circum chest.circum abdomen.circum
#> 1    1.07        12.3  23    154   67.8        36.2         93.1           85.2
#> 2    1.09         6.1  22    173   72.2        38.5         93.6           83.0
```

```
#> 3    1.04        25.3  22    154   66.2        34.0         95.8           87.9
#> 4    1.08        10.4  26    185   72.2        37.4        101.8           86.4
#> 5    1.03        28.7  24    184   71.2        34.4         97.3          100.0
#> 6    1.05        20.9  24    210   74.8        39.0        104.5           94.4
#>   hip.circum thigh.circum knee.circum ankle.circum bicep.circum forearm.circum
#> 1       94.5         59.0        37.3         21.9         32.0           27.4
#> 2       98.7         58.7        37.3         23.4         30.5           28.9
#> 3       99.2         59.6        38.9         24.0         28.8           25.2
#> 4      101.2         60.1        37.3         22.8         32.4           29.4
#> 5      101.9         63.2        42.2         24.0         32.2           27.7
#> 6      107.8         66.0        42.0         25.6         35.7           30.6
#>   wrist.circum
#> 1         17.1
#> 2         18.2
#> 3         16.6
#> 4         18.2
#> 5         17.7
#> 6         18.8
```

```
bodyfat[  !which(bodyfat$age  > 81), ]
```

```
#>  [1] density        percent.fat    age            weight         height
#>  [6] neck.circum    chest.circum   abdomen.circum hip.circum     thigh.circum
#> [11] knee.circum    ankle.circum   bicep.circum   forearm.circum wrist.circum
#> <0 rows> (or 0-length row.names)
```

```
## not equivalent!
```

An alternative to the use of `which()` for subsetting is to use the function `subset`. It can select rows and columns of a data frame.

```
## all samples with age between 40 and 60
## and height between 50 and 65
idx.age <- bodyfat$age > 40 & bodyfat$age < 60 & bodyfat$height > 50 & bodyfat$height < 65
subset(bodyfat, idx.age)
```

```
#>     density percent.fat age weight height neck.circum chest.circum abdomen.circum
#> 74    1.068        13.5  55    125     64        33.2         87.7             76
#> 216   0.995        47.5  51    219     64        41.2        119.8            122
#>     hip.circum thigh.circum knee.circum ankle.circum bicep.circum forearm.circum
#> 74        88.6         50.9        35.4         19.1         29.3           25.7
#> 216      112.8         62.5        36.9         23.6         34.7           29.1
#>     wrist.circum
#> 74          16.9
#> 216         18.4
```

```
## only their bodyfat
subset(bodyfat, idx.age, select ="percent.fat")
```

```
#>     percent.fat
#> 74         13.5
#> 216        47.5
```

4 Handling lists and the intelligent `apply` functions with the *purrr* package

Lists are the data structure R uses for hierarchical objects. Lists extend ordinary vectors to model objects that are like trees. You can create a hierarchical structure with a list because unlike vectors, a list can contain other lists.

There are three ways to subset a list:

- [extracts a sub-list. The result will always be a list.
- [[extracts a single component from a list. It removes a level of hierarchy from the list
- $ is a shorthand for extracting named elements of a list. It works similarly to [[except that you donâĂŹt need to use quotes.

which we will illustrate with the following example

```
a <- list(a = 1:3, b = "a string", c = pi, d = list(-1, -5))

a$a

#> [1] 1 2 3

a[["b"]]

#> [1] "a string"

a["b"]

#> $b
#> [1] "a string"
```

Looping over a list and doing something to each element is a very common operation in R. Unfortunately, the basic R functions like `apply, sapply, lapply` do neither offer a consistent interface nor a reliable return type.

The *purrr* package provides a family of `map_*` functions to to provide a consistent interface to list apply functions. They are called `map_*` since each steps consists of "mapping" a list value to a result. Detailed information on them can be found on this website.

Each function always returns the same type of output so there variations based on what sort of result you want. Each function takes a list as input, applies a function to each piece, and then returns a new vector thatâĂŹs the same length as the input. The type of the vector is determined by the specific map function. Usually you want to use the most specific available, using the default `map()` only as a fallback when there is no specialized equivalent available. Since a `data.frame` is just a special kind of list, We can use e.g. to compute the means of all the variables using `map_dbl` that is guaranteed to return a double value:

```
head(map_dbl(bodyfat, mean))

#>     density percent.fat         age      weight      height neck.circum
#>        1.06       19.15       44.88      178.92       70.15       37.99
```

The mapping functions have additional convenience feature like the automatic creation of simple functions as well as functions like `flatten` to remove a level of the hierarchy of the list (similar to `unlist`).

If you specify and index instead of a function, it will extract the respective element, in our example the fifth line of the data set.

```
head(map_dbl(bodyfat, 5))

#>     density percent.fat         age      weight      height neck.circum
#>        1.03       28.70       24.00      184.25       71.25       34.40

head(bodyfat[5, ])
```

```
#>   density percent.fat age weight height neck.circum chest.circum abdomen.circum
#> 5    1.03        28.7  24    184   71.2        34.4         97.3            100
#>   hip.circum thigh.circum knee.circum ankle.circum bicep.circum forearm.circum
#> 5        102         63.2        42.2           24         32.2           27.7
#>   wrist.circum
#> 5         17.7
```

This convenience function can of course become very handy with nested and named lists.

5 Handling complex objects in *R*: microrarray data*

So far, we have only been concerned with the handling of very simple objects in *R*. Now will look at to examples of more complex data set. The case is in point for such data sets are microrarray data, the handling of which we will explore in this section.

5.1 The Golub data

The gene expression data collected by Golub et al. (1999): Molecular classification of cancer: class discovery and class prediction by gene expression monitoring are among the classical data sets in bioinformatics. A pre–processed selection of the set is called golub and is contained in the *multtest* package, which is part of Bioconductor.

The data consist of gene expression values of 3051 genes (rows) from 38 leukemia patients (columns). Twenty seven patients are diagnosed as acute lymphoblastic leukemia (ALL) and eleven as acute myeloid leukemia (AML).

The tumor class is given by the numeric vector `golub.cl`, where ALL is indicated by 0 and AML by 1. The gene names are collected in the matrix `golub.gnames` of which the columns correspond to the gene index, ID, and Name, respectively.

We shall concentrate on expression values of a gene with probe set number "M92287_at", which is known as "CCND3 Cyclin D3". The expression values of this gene are collected in row 1042 of golub. To load the data and to obtain relevant information from row 1042 of `golub.gnames`, use the following code:

```
# load the golub data
data(golub, package = "multtest")
dim(golub)
```

```
#> [1] 3051   38
```

```
str(golub)
```

```
#>  num [1:3051, 1:38] -1.458 -0.752 0.457 3.135 2.766 ...
#>  - attr(*, "dimnames")=List of 2
#>   ..$ : NULL
#>   ..$ : NULL
```

```
golub.gnames[1042,]
```

```
#> [1] "2354"            "CCND3 Cyclin D3" "M92287_at"
```

```
### expression values of CCND 3
golub[1042,]
```

```
#>  [1]  2.109  1.524  1.964  2.336  1.851  1.994  2.066  1.816  2.176  1.809  2.446
#> [12]  1.905  2.766  1.326  2.594  1.928  1.105  1.276  1.831  1.784  0.458  2.181
#> [23]  2.314  1.999  1.368  2.374  1.835  0.889  1.450  0.429  0.827  0.636  1.022
```

```
#> [34]  0.128 -0.743  0.738  0.495  1.121

## define group factor
gol.fac <- factor(golub.cl, levels=0:1, labels = c("ALL","AML"))
```

So the matrix has 3051 rows and 38 columns, see also `dim(golub)`. Each data element has a row and a column index. Recall that the first index refers to rows and the second to columns. To view such large data sets, the function `head` is very useful.

The factor `gol.fac` and was constructed from the vector `golub.cl`, indicating the tumor class of the patients. This will turn out useful e.g. for separating the tumor groups in various visualization procedures.

Exercise: Handling the Golub data

(*a*) Print the gene expression values of Gene CCND3 for all AML patients using the factor `gol.fac`.
(*b*) For many types of computations it is very useful to combine a factor with the apply functionality: Use an apply function to compute the mean gene expression over the ALL and AML patients for each of the genes.
(*c*) Order the data matrix according to the mean expression values for ALL patients in decreasing order and give the names of the genes with largest mean expression value for ALL patients.

5.2 Bioconductor expression sets

In the last section we familiarized ourselves with the Golub microarray data which consists of three different objects: a matrix `golub` holding the gene expression measurements in a `data.frame`, `golub.gnames` holding the annotation of the genes and a `golub.cl` holding the sample groups.

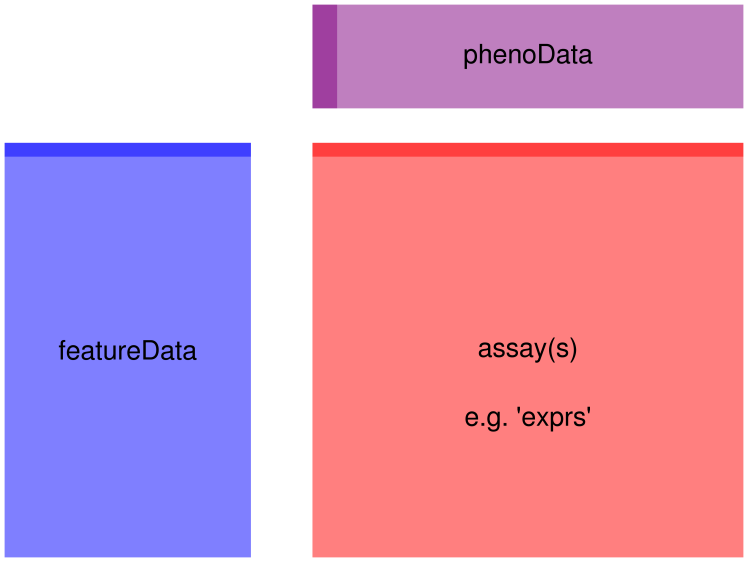
This illustrates that genomic data can be very complex, usually consisting of a number of different bits and pieces. In Bioconductor the approach is taken that these pieces should be stored in a single structure to easily manage the data.

The package *Biobase* contains standardized data structures to represent genomic data. The `ExpressionSet` class is designed to combine several different sources of information into a single convenient structure. An ExpressionSet can be manipulated (e.g., subsetted, copied), and is the input to or output of many Bioconductor functions.

The data in an ExpressionSet consist of

(*a*) **assayData**: Expression data from microarray experiments (assayData is used to hint at the methods used to access different data components).
(*b*) **metaData**: A description of the samples in the experiment (phenoData), metadata about the features on the chip or technology used for the experiment (featureData), and further annotations for the features, for example gene annotations from biomedical databases (annotation).
(*c*) **experimentData**: A flexible structure to describe the experiment.

The ExpressionSet class coordinates all of these data, so that you do not usually have to worry about the details. However, an ExpressionSet needs to be created in the first place, because it will be the starting point for many of the analyses using Bioconductor software.



In the following exercise, you learn how to handle `ExpressionSet` objects. Note that printing an Expression set object will return an informative summary of the object: it often gives you hints on how to extract data from the object. For example, you can get the expression data using the function `exprs`.

Exercise: Handling Bioconductor expression sets

(*a*) obtain sample expression set object from the *Biobase* Bioconductor package using `data(sample.ExpressionSet)` and extract the contained gene expression data. Use the function `slotNames()` to obtain an overview of the elements or "slots" of the object.
(*b*) Extract a description of the experiment from the object. Which variables have been measured on which samples? Is there any metadata on the variables?
(*c*) Which microarray was used in the experiment? Which "features" (probes) are measured?
(*d*) How many control probes are contained in the data set? HINT: Their names starts with "AFFX", use the function `grep` to obtain them. They are usually filtered out prior to further analysis.
(*e*) Find out how to obtain a `phenoData` table, i.e. a table containing the sample annotation.

5.3 Bioconductor GRanges and summarized experiments

The Bioconductor calls *SummarizedExperiment* can be see as an ExpressionSet variant for sequencing experiments. The feature data (now called rowRanges) contains *GenomicRanges* that represent genomic information on features assayed. In the simplest case, these are the genomic coordinates of the features. The *GenomicRanges* offers a very flexible and nice framework for working with genomic data. A summary of its capabilities is given in the article on it, as well as in this presentation.

Here we look at simple use case: We start from a drosophila RNA–Seq data set, *pasilla*, which contains only the Flybase gene identifiers and then add the annotation of the corresponding transcripts.

We obtain this annotation from the *TxDb.Dmelanogaster.UCSC.dm3.ensGene*. Since the creation of the pasilla data set dates back to 2012 or so, some gene ids are no longer present in the current annotation and we have to tackle them so that the matching of the two id sets works.

```
pasilla_se
```

```
#> class: DESeqDataSet
#> dim: 14470 7
#> metadata(0):
#> assays(1): counts
#> rownames(14470): FBgn0000003 FBgn0000008 ... FBgn0261574 FBgn0261575
#> rowRanges metadata column names(0):
#> colnames(7): treated1fb treated2fb ... untreated3fb untreated4fb
#> colData names(2): condition type
```

```
# phenoData
colData(pasilla_se)
```

```
#> DataFrame with 7 rows and 2 columns
#>              condition        type
#>               <factor>    <factor>
#> treated1fb     treated single-read
#> treated2fb     treated  paired-end
#> treated3fb     treated  paired-end
#> untreated1fb untreated single-read
#> untreated2fb untreated single-read
#> untreated3fb untreated  paired-end
#> untreated4fb untreated  paired-end
```

```
# featureData
rowRanges(pasilla_se)
```

```
#> GRangesList object of length 14470:
#> $FBgn0000003
#> GRanges object with 0 ranges and 0 metadata columns:
#>    seqnames    ranges strand
#>       <Rle> <IRanges>  <Rle>
#>
#> $FBgn0000008
#> GRanges object with 0 ranges and 0 metadata columns:
#>      seqnames ranges strand
#>
#> $FBgn0000014
#> GRanges object with 0 ranges and 0 metadata columns:
#>      seqnames ranges strand
#>
#> ...
#> <14467 more elements>
#> -------
#> seqinfo: no sequences
```

```
ids_pasilla <- names(rowRanges(pasilla_se))

# get the transcripts by genes
dm_genes <- transcriptsBy(TxDb.Dmelanogaster.UCSC.dm3.ensGene, by = "gene")

dm_genes
```

```
#> GRangesList object of length 15682:
#> $FBgn0000003
#> GRanges object with 1 range and 2 metadata columns:
```

```
#>       seqnames              ranges strand |     tx_id     tx_name
#>          <Rle>           <IRanges>  <Rle> | <integer> <character>
#>   [1]    chr3R [2648220, 2648518]      + |     17345 FBtr0081624
#>
#> $FBgn0000008
#> GRanges object with 3 ranges and 2 metadata columns:
#>       seqnames              ranges strand | tx_id     tx_name
#>   [1]    chr2R [18024494, 18060339]      + |  7681 FBtr0100521
#>   [2]    chr2R [18024496, 18060346]      + |  7682 FBtr0071763
#>   [3]    chr2R [18024938, 18060346]      + |  7683 FBtr0071764
#>
#> $FBgn0000014
#> GRanges object with 4 ranges and 2 metadata columns:
#>       seqnames              ranges strand | tx_id     tx_name
#>   [1]    chr3R [12632936, 12655767]      - | 21863 FBtr0306337
#>   [2]    chr3R [12633349, 12653845]      - | 21864 FBtr0083388
#>   [3]    chr3R [12633349, 12655300]      - | 21865 FBtr0083387
#>   [4]    chr3R [12633349, 12655474]      - | 21866 FBtr0300485
#>
#> ...
#> <15679 more elements>
#> -------
#> seqinfo: 15 sequences (1 circular) from dm3 genome
```

```
# add missing gene IDs to dm_genes
missing_ids <- setdiff(ids_pasilla, names(dm_genes))
dm_genes <- c(dm_genes, rowRanges(pasilla_se)[missing_ids])

# match the two sets
idx_se_in_dm_genes <- match(ids_pasilla, names(dm_genes))

# sanity check whether there is a missing match
any(is.na(idx_se_in_dm_genes))
```

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

```
# sanity check whether gene id_s match
all.equal(ids_pasilla, names(dm_genes[idx_se_in_dm_genes]))
```

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

```
# finally annotate the ranges
rowRanges(pasilla_se) <- dm_genes[idx_se_in_dm_genes]

# sample ranges to preview the data
rowRanges(pasilla_se)[sample(length(ids_pasilla), 5)]
```

```
#> GRangesList object of length 5:
#> $FBgn0024245
#> GRanges object with 1 range and 2 metadata columns:
#>       seqnames              ranges strand |     tx_id     tx_name
#>          <Rle>           <IRanges>  <Rle> | <integer> <character>
#>   [1]    chr2L [19338676, 19363224]      - |      5008 FBtr0081224
#>
#> $FBgn0035238
#> GRanges object with 2 ranges and 2 metadata columns:
```

```
#>       seqnames             ranges strand | tx_id     tx_name
#>   [1]    chr3L [1600726, 1602674]      - | 14313 FBtr0333360
#>   [2]    chr3L [1600942, 1602674]      - | 14314 FBtr0072825
#>
#> $FBgn0083988
#> GRanges object with 1 range and 2 metadata columns:
#>       seqnames             ranges strand | tx_id     tx_name
#>   [1]    chr2R [8087167, 8087257]      - |  9204 FBtr0111041
#>
#> ...
#> <2 more elements>
#> -------
#> seqinfo: 15 sequences (1 circular) from dm3 genome
```

```
# another sanity check
all.equal(ids_pasilla, names(rowRanges(pasilla_se)))
```

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

6 Advanced data handling with *dplyr* verbs

The handling techniques available in base *R* can be greatly enhanced by using the data manipulation "verbs" that are available in *dplyr*. They allow easy and efficient handling and manipulation of data frames.

6.1 Subsetting and viewing functions in *dplyr*

The package *dplyr* provides a "grammar" of data manipulation. We will also use it later in the context of the "split–apply–combine" strategy that we will discuss below.

Since the first thing you do in a data manipulation task is to subset/transform your data, it includes "verbs" that provide basic functionality. We will introduce these in the following. The command structure for all *dplyr* verbs is :

- first argument is a data frame
- return value is a data frame
- nothing is modified in place

Note that *dplyr* generally does not preserve row names. A further introductory document including a youtube video by Kevin Markham can be found here.

6.2 Selecting rows with `filter()`

The function `filter()` allows you to select a subset of the rows of a data frame. The first argument is the name of the data frame, and the second and subsequent are filtering expressions evaluated in the context of that data frame. This makes the selection commands above less verbose and easier to grasp.

```
## all samples with age between 40 and 60

head(filter(bodyfat, age > 40, age < 60 ))
```

```
#>   density percent.fat age weight height neck.circum chest.circum abdomen.circum
#> 1    1.05        21.3  41    218   71.0        39.8          112          100.5
#> 2    1.03        32.3  41    247   73.5        42.1          117          115.6
#> 3    1.01        40.1  49    192   65.0        38.4          118          113.1
```

```
#> 4    1.03        28.4  50    197   68.2        42.1          106           98.8
#> 5    1.02        35.2  46    363   72.2        51.2          136          148.1
#> 6    1.03        32.6  50    203   67.0        40.2          115          108.1
#>   hip.circum thigh.circum knee.circum ankle.circum bicep.circum forearm.circum
#> 1        108         67.1        44.2         25.2         37.5           31.5
#> 2        116         71.2        43.3         26.3         37.3           31.7
#> 3        114         61.9        38.3         21.9         32.0           29.8
#> 4        105         66.0        41.5         24.7         33.2           30.5
#> 5        148         87.3        49.1         29.6         45.0           29.0
#> 6        102         61.3        41.1         24.7         34.1           31.0
#>   wrist.circum
#> 1         18.7
#> 2         19.7
#> 3         17.0
#> 4         19.4
#> 5         21.4
#> 6         18.3
```

```
tail(filter(bodyfat, age > 40, age < 60 ))
```

```
#>     density percent.fat age weight height neck.circum chest.circum abdomen.circum
#> 119    1.06        14.9  56    174   69.5        38.1        104.0           89.4
#> 120    1.06        17.0  56    168   68.5        37.4         98.6           93.0
#> 121    1.07        10.6  57    148   65.8        35.2         99.6           86.4
#> 122    1.06        16.1  57    182   71.8        39.4        103.4           96.7
#> 123    1.06        15.4  58    176   71.5        38.0        100.2           88.1
#> 124    1.04        26.7  58    162   67.2        35.1         94.9           94.9
#>     hip.circum thigh.circum knee.circum ankle.circum bicep.circum forearm.circum
#> 119       98.4         58.4        37.4         22.5         34.6           30.1
#> 120       97.0         55.4        38.8         23.2         32.4           29.7
#> 121       90.1         53.0        35.0         21.3         31.7           27.3
#> 122      100.7         59.3        38.6         22.8         31.8           29.1
#> 123       97.8         57.1        38.9         23.6         30.9           29.6
#> 124      100.2         56.8        35.9         21.0         27.8           26.1
#>     wrist.circum
#> 119         18.8
#> 120         19.0
#> 121         16.9
#> 122         19.0
#> 123         18.0
#> 124         17.6
```

`filter()` works similarly to `subset()` except that you can give it any number of filtering conditions which are joined together with & (not && which is easy to do accidentally otherwise). You can use other boolean operators explicitly as in :

```
## all samples with age of 40 or 60

head(filter(bodyfat, age == 40 | age == 60 ), 3)
```

```
#>   density percent.fat age weight height neck.circum chest.circum abdomen.circum
#> 1    1.04        24.2  40    202   70.0        38.5        106.5          100.9
#> 2    1.07        10.8  40    134   67.5        33.6         88.2           73.7
#> 3    1.08         6.6  40    139   69.0        34.3         89.2           77.9
#>   hip.circum thigh.circum knee.circum ankle.circum bicep.circum forearm.circum
#> 1      106.2         63.5        39.9         22.6         35.1           30.6
```

```
#> 2       88.5         53.3        34.5         22.5         27.9          26.2
#> 3       91.0         51.4        34.9         21.0         26.7          26.1
#>   wrist.circum
#> 1         19.0
#> 2         17.3
#> 3         17.2
```

```
tail(filter(bodyfat, age == 40 | age == 60 ), 3)
```

```
#>    density percent.fat age weight height neck.circum chest.circum abdomen.circum
#> 16    1.06        17.5  40    170   74.2        37.7         98.9           90.4
#> 17    1.08         8.6  40    168   71.5        39.4         89.5           83.7
#> 18    1.04        25.8  60    158   67.5        40.4         97.2           93.3
#>    hip.circum thigh.circum knee.circum ankle.circum bicep.circum forearm.circum
#> 16       95.5         55.4        38.9         22.4         30.5          28.9
#> 17       98.1         57.3        39.7         22.6         32.9          29.3
#> 18       94.0         54.3        35.7         21.0         31.3          28.7
#>    wrist.circum
#> 16         17.7
#> 17         18.2
#> 18         18.3
```

`head()` and `tail()` return the first and the last entries of a data frame respectively.

6.3 Arranging rows with `arrange()`

`arrange()` works similarly to `filter()` except that instead of filtering or selecting rows, it reorders them. It takes a data frame, and a set of column names (or more complicated expressions) to order by. If you provide more than one column name, each additional column will be used to break ties in the values of preceding columns:

```
## arrange by age and bodyfat
head(arrange(bodyfat, age, percent.fat),3 )
```

```
#>   density percent.fat age weight height neck.circum chest.circum abdomen.circum
#> 1    1.09         6.1  22    173   72.2        38.5         93.6           83.0
#> 2    1.04        25.3  22    154   66.2        34.0         95.8           87.9
#> 3    1.08         9.4  23    160   72.2        35.5         92.1           77.1
#>   hip.circum thigh.circum knee.circum ankle.circum bicep.circum forearm.circum
#> 1       98.7         58.7        37.3         23.4         30.5          28.9
#> 2       99.2         59.6        38.9         24.0         28.8          25.2
#> 3       93.9         56.1        36.1         22.7         30.5          27.2
#>   wrist.circum
#> 1         18.2
#> 2         16.6
#> 3         18.2
```

Use `desc()` to order a column in descending order:

```
## descending
head(arrange(bodyfat, desc(age), percent.fat),3 )
```

```
#>   density percent.fat age weight height neck.circum chest.circum abdomen.circum
#> 1    1.05        21.5  81    161   70.2        37.8         96.4           95.4
#> 2    1.03        31.9  74    208   70.0        40.8        112.4          108.5
#> 3    1.06        14.9  72    158   67.2        37.7         97.5           88.1
#>   hip.circum thigh.circum knee.circum ankle.circum bicep.circum forearm.circum
#> 1       99.3         53.5        37.5         21.5         31.4          26.8
```

```
#> 2      107.1         59.3        42.2         24.6         33.7          30.0
#> 3       96.9         57.2        37.7         21.8         32.6          28.0
#>   wrist.circum
#> 1         18.3
#> 2         20.9
#> 3         18.8
```

6.4 Select columns with `select()`

Often you work with large data sets with many columns where only a few are actually of interest to you. `select()` allows you to rapidly zoom in on a useful subset using operations that usually only work on numeric variable positions:

```
## select fact age and heigth only
head(select(bodyfat, age, height, percent.fat ))
```

```
#>   age height percent.fat
#> 1  23   67.8        12.3
#> 2  22   72.2         6.1
#> 3  22   66.2        25.3
#> 4  26   72.2        10.4
#> 5  24   71.2        28.7
#> 6  24   74.8        20.9
```

```
## select all body measures
head(select(bodyfat, weight:wrist.circum))
```

```
#>   weight height neck.circum chest.circum abdomen.circum hip.circum thigh.circum
#> 1    154   67.8        36.2         93.1           85.2       94.5         59.0
#> 2    173   72.2        38.5         93.6           83.0       98.7         58.7
#> 3    154   66.2        34.0         95.8           87.9       99.2         59.6
#> 4    185   72.2        37.4        101.8           86.4      101.2         60.1
#> 5    184   71.2        34.4         97.3          100.0      101.9         63.2
#> 6    210   74.8        39.0        104.5           94.4      107.8         66.0
#>   knee.circum ankle.circum bicep.circum forearm.circum wrist.circum
#> 1        37.3         21.9         32.0           27.4         17.1
#> 2        37.3         23.4         30.5           28.9         18.2
#> 3        38.9         24.0         28.8           25.2         16.6
#> 4        37.3         22.8         32.4           29.4         18.2
#> 5        42.2         24.0         32.2           27.7         17.7
#> 6        42.0         25.6         35.7           30.6         18.8
```

```
## exclude all body measures
head(select(bodyfat, -(weight:wrist.circum)))
```

```
#>   density percent.fat age
#> 1    1.07        12.3  23
#> 2    1.09         6.1  22
#> 3    1.04        25.3  22
#> 4    1.08        10.4  26
#> 5    1.03        28.7  24
#> 6    1.05        20.9  24
```

Note that the base function `subset()` includes an option that works similar to the `filter()` function.

6.5 Add columns with `mutate()`

Additionally, `dplyr::mutate()`, similarly to `base::transform()`, allows to add columns. The key difference between `mutate()` and `transform()` is that mutate allows you to refer to columns that you just created. (The double colon allows you to access functions from a specific package.)

For example, going to the patients data, we can easily calculate a BMI column using `mutate`:

```
pat <- read_csv("http://www-huber.embl.de/users/klaus/BasicR/Patients.csv")
pat$Weight[2] <- mean(pat$Weight, na.rm=TRUE)
mutate(pat, BMI <- Weight / Height^2)
```

```
#> Source: local data frame [3 x 5]
#>
#>   PatientId Height Weight Gender BMI <- Weight/Height^2
#>       (chr)  (dbl)  (dbl)  (chr)                  (dbl)
#> 1        P1   1.65     80      f                   29.4
#> 2        P2   1.30     65      m                   38.5
#> 3        P3   1.20     50      f                   34.7
```

6.6 Summaries with `summarize()`

The function `summarize()`, which creates a new data frame from a calculation on the current one collapses `bodyfat` to a single row. This not very useful yet but becomes very handy on grouped/splitted data.

```
summarize(bodyfat, mean.age = mean(age, na.rm = TRUE),
              mean.BMI = mean( (weight*0.454) / (height*.0254)^2 ) )
```

```
#>   mean.age mean.BMI
#> 1     44.9       26
```

Again, it will become useful later, when we apply functions on splitted data frames.

Commonalities

You may have noticed that all these functions are very similar:

- The first argument is a data frame.
- The subsequent arguments describe what to do with it, and you can refer to columns in the data frame directly without using $
- The result is a new data frame

Together these properties make it easy to chain together multiple simple steps to achieve a complex result.

These five functions provide the basis of a language of data manipulation. At the most basic level, you can only alter a tidy data frame in five useful ways: you can reorder the rows (`arrange()`), pick observations and variables of interest (`filter()` and `select()`), add new variables that are functions of existing variables (`mutate()`) or collapse many values to a summary (`summarize()`).

The remainder of the language comes from applying the five functions to different types of data, like to grouped data, as described in the next section.

Exercise: Bodyfat data

(*a*) Calculate mean and sd for all the variables in the data set. HINT: Use an appropriate `apply` function.
(*b*) Find the indexes of all men in the data set that are relatively small, i.e, who are less than `mean(height) - sd(height)` tall.

(*c*) Compute a similar index for weight, i.e. find the light people.
(*d*) Find the small and light people, i.e. find the intersection of the two index-sets. Use the logical operator `&` — and.

7 Computing with large data sets: the split–apply–combine strategy

Very often, data analysis happens in a "split–apply–combine" fashion. You break up a bigger problem into manageable pieces, operate on each piece independently and then put all the pieces back together.

We will illustrate this using a subset of plates from a high throughput screen RNAi screen (Simpson et. al. 2012, Nature Cell Biology). In a nutshell, plates with 384 spots containing HeLa cells were transfected with specific siRNAs targeting certain genes.

The general aim of the screen was then to identify genes important in the early secretory pathway. This was studied by imaging the transport of a model protein called VSVG from the ER to the golgi and then to the plasma membrane.

Here we look at 3 plates and all their replicates from the screen, the data contains annotation information, (e.g. plate number, well number, well row and column etc.) and the columns `TransR.n` (for normalized transport ratio) which is a measure indicating the transport success as well as `CellNumber.n` containing the number of cells per spot. The transport ratio is the ratio of intracellular VSVG protein over VSVG protein exposed on the cell surface.

Note that it is common to represent a spot on a plate by combination of its row– and column number, whereby row numbers are given by upper–case letters. We first load the data.

```
load(url("http://www-huber.embl.de/users/klaus/BasicR/HTSdata.RData"))
head(HTSdata)
```

```
#>       plate replicate well WellNumber TransR CellNumber
#> 16513     9         1  A02         13 -0.609         28
#> 16514     9         1  A03         25    Inf         19
#> 16515     9         1  A04         37 -0.373         25
#> 16516     9         1  A05         49 -0.371         29
#> 16517     9         1  A06         61 -0.139         26
#> 16518     9         1  A07         73 -0.206         53
#>                                 rawAnno WellColumn WellRow Unknown siRNAID  GeneID
#> 16513 013--02--01--(2,9)--112361--GALNTL5          2       A   (2,9)  112361 GALNTL5
#> 16514    025--03--01--(3,17)--110623--OAT          3       A  (3,17)  110623     OAT
#> 16515   037--04--01--(5,1)--111945--NAA20          4       A   (5,1)  111945   NAA20
#> 16516   049--05--01--(1,2)--28431--INCENP          5       A   (1,2)   28431  INCENP
#> 16517   061--06--01--(2,10)--121391--PARN          6       A  (2,10)  121391    PARN
#> 16518  073--07--01--(3,18)--104391--CLCN4          7       A  (3,18)  104391   CLCN4
#>       content TransR.n CellNumber.n
#> 16513   empty   -1.475       -0.578
#> 16514  sample    6.000       -1.445
#> 16515  sample   -0.323       -0.867
#> 16516  sample   -0.311       -0.482
#> 16517  sample    0.819       -0.771
#> 16518  sample    0.492        1.831
```

7.1 Grouping and summarizing

The *dplyr* now provides the framework to use a split–apply-combine strategy. These selection verbs introduced above are useful, but they become really powerful when you combine them with the idea of "group by" or **splitting** operator, repeating the operation individually on groups of observations within the dataset.

In *dplyr*, you use the `group_by()` function to describe how to break a dataset down into groups of rows. You can then use the resulting object in the exactly the same verb–functions as above; they'll automatically work "by group" when the input is a grouped.

Of the five verbs, `select()` is unaffected by grouping, and grouped `arrange()` orders first by grouping variables. Group-wise `mutate()` and `filter()` are most useful in conjunction with window functions, and are described in detail in a corresponding vignette and will not be discussed here.

As an example, in order to split our data according to the plate numbers, we can use the following command:

```
split_HTS <- group_by(HTSdata, plate)
split_HTS
```

```
#> Source: local data frame [5,760 x 15]
#> Groups: plate [3]
#>
#>    plate replicate  well WellNumber  TransR CellNumber
#>    (int)     (int) (chr)      (dbl)   (dbl)      (int)
#> 1      9         1   A02         13 -0.6092         28
#> 2      9         1   A03         25     Inf         19
#> 3      9         1   A04         37 -0.3731         25
#> 4      9         1   A05         49 -0.3706         29
#> 5      9         1   A06         61 -0.1392         26
#> 6      9         1   A07         73 -0.2060         53
#> 7      9         1   A08         85 -0.3319         33
#> 8      9         1   A09         97 -0.0863         43
#> 9      9         1   A10        109 -0.3231         31
#> 10     9         1   A11        121 -0.1807         30
#> ..   ...       ...   ...        ...     ...        ...
#> Variables not shown: rawAnno (chr), WellColumn (int), WellRow (chr), Unknown (chr),
#>   siRNAID (chr), GeneID (chr), content (chr), TransR.n (dbl), CellNumber.n (dbl)
```

We get back a `grouped_df`, which is a special class in *dplyr* that is also a *data.frame*. We can also split by single plates

```
split_HTS_rep  <-  group_by(HTSdata, plate, replicate)
split_HTS_rep
```

```
#> Source: local data frame [5,760 x 15]
#> Groups: plate, replicate [15]
#>
#>    plate replicate  well WellNumber  TransR CellNumber
#>    (int)     (int) (chr)      (dbl)   (dbl)      (int)
#> 1      9         1   A02         13 -0.6092         28
#> 2      9         1   A03         25     Inf         19
#> 3      9         1   A04         37 -0.3731         25
#> 4      9         1   A05         49 -0.3706         29
#> 5      9         1   A06         61 -0.1392         26
#> 6      9         1   A07         73 -0.2060         53
#> 7      9         1   A08         85 -0.3319         33
#> 8      9         1   A09         97 -0.0863         43
#> 9      9         1   A10        109 -0.3231         31
#> 10     9         1   A11        121 -0.1807         30
#> ..   ...       ...   ...        ...     ...        ...
#> Variables not shown: rawAnno (chr), WellColumn (int), WellRow (chr), Unknown (chr),
#>   siRNAID (chr), GeneID (chr), content (chr), TransR.n (dbl), CellNumber.n (dbl)
```

We can now make full use of the `summarize()` function, which constructs a new data frame using the columns of the

current one. Fore example, we can easily use `summarize` to compute average cell numbers per single plate using the grouped data frame just obtained.

```
HTS_cellNumbers  <- summarize(split_HTS_rep, mean_CN = mean(CellNumber, na.rm = T))
HTS_cellNumbers
```

```
#> Source: local data frame [15 x 3]
#> Groups: plate [?]
#>
#>    plate replicate mean_CN
#>    (int)     (int)   (dbl)
#> 1      9         1    33.6
#> 2      9         2    41.0
#> 3      9         3    22.4
#> 4      9         4    25.6
#> 5      9         5    41.3
#> 6     49         1    35.9
#> 7     49         2    39.8
#> 8     49         3    43.8
#> 9     49         4    44.5
#> 10    49         5    43.1
#> 11   152         1    24.7
#> 12   152         2    16.1
#> 13   152         3    20.1
#> 14   152         4    25.2
#> 15   152         5    25.7
```

We see that plate 152 has a lower cell number than the other two plates on average. It is also handy to use custom functions, for example computing the ratio of mean and median cell number for every plate:

```
HTS_skew <- summarize(split_HTS_rep,
HTS_CN_skew = mean(CellNumber, na.rm = T) / median(CellNumber, na.rm = T))

HTS_skew
```

```
#> Source: local data frame [15 x 3]
#> Groups: plate [?]
#>
#>    plate replicate HTS_CN_skew
#>    (int)     (int)       (dbl)
#> 1      9         1       0.989
#> 2      9         2       1.000
#> 3      9         3       1.068
#> 4      9         4       1.065
#> 5      9         5       1.008
#> 6     49         1       0.997
#> 7     49         2       0.972
#> 8     49         3       0.973
#> 9     49         4       0.978
#> 10    49         5       0.980
#> 11   152         1       1.029
#> 12   152         2       1.006
#> 13   152         3       1.003
#> 14   152         4       1.006
#> 15   152         5       0.988
```

We see that the cell numbers are quite symmetrically distributed. Note that summarizing peels of one level of grouping,

thus we can now easily compute a mean skew per plate.

```
plate_HTS_skew <- summarize(HTS_skew, mean_CN_skew = mean(HTS_CN_skew))

plate_HTS_skew
```

```
#> Source: local data frame [3 x 2]
#>
#>   plate mean_CN_skew
#>   (int)        (dbl)
#> 1     9         1.03
#> 2    49         0.98
#> 3   152         1.01
```

7.2 Other useful functions

dplyr provides a handful of others useful helper functions:

- `tbl_df()` : creates a "local data frame". It simply a wrapper for a data frame that prints nicely, similar to *DataFrame* from *IRanges*.
- `glimpse()` : nice alternative to `str` that will print columns down the page and data rows run across.
- `sample_n()` : sample n random rows from a *tbl_df*. There also exists sample_frac()
- `n()` : number of observations in the current group
- `tally()` : will call `n()` or `sum(n)`, depending on whether you call it you're tallying for the first time, or re–tallying.
- `n_distinct(x)` : count the number of unique values in `x`.
- `first(x)`, `last(x)` and `nth(x, n)` — these work similarly to `x[1]`, `x[length(x)]`, and `x[n]` but give you more control of the result if the value isn't present.
  Some examples of their usage:

```
## nice plotting
HTSdata <- tbl_df(HTSdata )

sample_n(HTSdata, 3)
```

```
#> Source: local data frame [3 x 15]
#>
#>   plate replicate  well WellNumber TransR CellNumber
#>   (int)     (int) (chr)      (dbl)  (dbl)      (int)
#> 1     9         3   E04         41 -0.102         15
#> 2    49         3   C31        363  0.916         56
#> 3   152         4   J05         58  0.148         29
#> Variables not shown: rawAnno (chr), WellColumn (int), WellRow (chr), Unknown (chr),
#>   siRNAID (chr), GeneID (chr), content (chr), TransR.n (dbl), CellNumber.n (dbl)
```

```
## overview of variable
glimpse(HTSdata)
```

```
#> Observations: 5,760
#> Variables: 15
#> $ plate        (int) 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9, 9,...
#> $ replicate    (int) 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,...
#> $ well         (chr) "A02", "A03", "A04", "A05", "A06", "A07", "A08", "A09", "A1...
#> $ WellNumber   (dbl) 13, 25, 37, 49, 61, 73, 85, 97, 109, 121, 133, 145, 157, 16...
#> $ TransR       (dbl) -0.6092, Inf, -0.3731, -0.3706, -0.1392, -0.2060, -0.3319, ...
#> $ CellNumber   (int) 28, 19, 25, 29, 26, 53, 33, 43, 31, 30, 31, 29, 32, 37, 41,...
#> $ rawAnno      (chr) "013--02--01--(2,9)--112361--GALNTL5", "025--03--01--(3,17)...
#> $ WellColumn   (int) 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,...
#> $ WellRow      (chr) "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A", "A",...
```

```
#> $ Unknown      (chr) "(2,9)", "(3,17)", "(5,1)", "(1,2)", "(2,10)", "(3,18)", "(...
#> $ siRNAID      (chr) "112361", "110623", "111945", "28431", "121391", "104391", ...
#> $ GeneID       (chr) "GALNTL5", "OAT", "NAA20", "INCENP", "PARN", "CLCN4", "CLCA...
#> $ content      (chr) "empty", "sample", "sample", "sample", "sample", "sample", ...
#> $ TransR.n     (dbl) -1.4751, 6.0000, -0.3230, -0.3108, 0.8186, 0.4924, -0.1216,...
#> $ CellNumber.n (dbl) -0.5781, -1.4453, -0.8672, -0.4818, -0.7708, 1.8308, -0.096...
## number of different plate designs
summarize(HTSdata, n_distinct(plate) )
#> Source: local data frame [1 x 1]
#>
#>   n_distinct(plate)
#>               (int)
#> 1                 3
## number of replicates per plate for  plates
## with number greater than 15
filter(
  summarize(
      group_by(HTSdata, plate),
      rep_per_plate = n_distinct(replicate)
      )
  , plate > 15)
#> Source: local data frame [2 x 2]
#>
#>   plate rep_per_plate
#>   (int)         (int)
#> 1    49             5
#> 2   152             5
```

8 chaining with *magrittr*

The *dplyr* interface is functional in the sense that function calls don't have side–effects so you must always save their results. This doesn't lead to particularly elegant code if you want to do many operations at once. You either have to do it step–by–step or wrap the function calls inside each other as we did above.

This is difficult to read because the order of the operations is from inside to out, and the arguments are a long way away from the function. To get around this problem, *dplyr* imports the % >% operator (read "in") from the package *magrittr*.

8.1 The chaining operators and its usage

x % >% f(y) turns into f(x, y) so you can use it to rewrite multiple operations so you can read from left–to–right, top–to–bottom:

```
# create two vectors and calculate Euclidian distance between them
x1 <- 1:5; x2 <- 2:6

# usual way
sqrt(sum((x1-x2)^2))

#> [1] 2.24

# chaining method
(x1-x2)^2 %>%
sum() %>%
```

```
sqrt()
```

```
#> [1] 2.24
```

which is much easier to grasp. We can also apply this to our HTS:

```
## number of plates
summarize(HTSdata, n_distinct(plate) )
```

```
#> Source: local data frame [1 x 1]
#>
#>   n_distinct(plate)
#>               (int)
#> 1                 3
```

```
## number of replicates per plate for  plates
## plate number greater than 15

HTSdata %>%
group_by(plate)  %>%
summarize(rep_per_plate = n_distinct(replicate)) %>%
filter(plate > 15)
```

```
#> Source: local data frame [2 x 2]
#>
#>   plate rep_per_plate
#>   (int)         (int)
#> 1    49             5
#> 2   152             5
```

It makes clear that you group first, then you summarize and then you filter.

8.2 The `do` function for general operations on grouped data*

The do allows to apply functions on (grouped) data frames that return complex return values, e.g. lists. A case in point are regression fits. Nonetheless, its handling is quite complex but described here for completeness.

The function do always returns a data frame. The first columns in the data frame will be the group labels, the others will be computed from arguments. Named arguments become list–columns in the result data frame, with one element for each group; unnamed elements **must be data frames** and labels will be duplicated accordingly.

For example, we can print the first two sample points per single plate using the dot operator for accessing the input to do in an unnamed argument. Note the duplicated label columns!

```
HTSdata %>%
group_by(plate, replicate)  %>%
do( head(.,2))
```

```
#> Source: local data frame [30 x 15]
#> Groups: plate, replicate [15]
#>
#>    plate replicate  well WellNumber  TransR CellNumber
#>    (int)     (int) (chr)      (dbl)   (dbl)      (int)
#> 1      9         1   A02         13 -0.6092         28
#> 2      9         1   A03         25     Inf         19
#> 3      9         2   A02         13 -0.7227         30
#> 4      9         2   A03         25 -0.3434         56
#> 5      9         3   A03         25  0.2437         43
```

```
#> 6      9         3   A04         37  0.0974         37
#> 7      9         4   A02         13  0.7239         31
#> 8      9         4   A03         25  0.7594         55
#> 9      9         5   A02         13 -0.2531         57
#> 10     9         5   A03         25 -0.2062         53
#> ..   ...       ...   ...        ...     ...        ...
#> Variables not shown: rawAnno (chr), WellColumn (int), WellRow (chr), Unknown (chr),
#>   siRNAID (chr), GeneID (chr), content (chr), TransR.n (dbl), CellNumber.n (dbl)
```

Naming the argument returns a list in the non–group columns. Note that due to the usage of lists, there are now no duplicated label columns.

```
Preview <- HTSdata %>%
group_by(plate, replicate)  %>%
do(plate_preview = head(.,2))

Preview$plate_preview[[1]]
```

```
#> Source: local data frame [2 x 15]
#>
#>   plate replicate  well WellNumber TransR CellNumber
#>   (int)     (int) (chr)      (dbl)  (dbl)      (int)
#> 1     9         1   A02         13 -0.609         28
#> 2     9         1   A03         25    Inf         19
#> Variables not shown: rawAnno (chr), WellColumn (int), WellRow (chr), Unknown (chr),
#>   siRNAID (chr), GeneID (chr), content (chr), TransR.n (dbl), CellNumber.n (dbl)
```

Exercise: HTS data handling

(*a*) load the HTS data from

http://www-huber.embl.de/users/klaus/BasicR/HTSdata.RData.

(*b*) Compute the mean (function `mean`) and standard deviation (function `sd`) of the cell number for every single plate.
(*c*) Using the function `identity` in connection with the dot operator and the `do`, create a list of vectors containing only the non–infinite transport ratios for every single plate, replacing the non-finite ratios by zero.
HINT: Use the function `is.finite` as well as an `ifelse` command.

9 Tidy data and easy reshaping of data frames

9.1 The concept of tidy data

A lot of analysis time is spent on the process of cleaning and preparing the data. Data preparation is not just a first step, but must be repeated many over the course of analysis as new problems come to light or new data is collected. An often neglected, but important aspect of data cleaning is data tidying: structuring datasets to facilitate analysis.

This "data tidying" includes the ability to move data between different different shapes.

In a nutshell, a dataset is a collection of values, usually either numbers (if quantitative) or strings (if qualitative). Values are organized in two ways. Every value belongs to a variable and an observation. A variable contains all values that measure the same underlying attribute (like height, temperature, duration) across units.

An observation contains all values measured on the same unit (like a person, or a day, or a race) across attributes.

A tidy data frame now organizes the data in such a way that each observation corresponds to an single line in the data set. This is in general the most appropriate format for downstream analysis, although it might not be the most appropriate form for viewing the data.

For a thorough discussion of this topic see the paper by Hadley Wickham - tidy data.

9.2 Reshaping data: *tidyr* and *reshape2*

To illustrate the concepts, we're looking at some experimental data: different doses of HFG (a cytokine) were applied to cells and the downstream effect to the phosphorylation of target proteins were assessed recording a time course–signal in different conditions. This data and the exercise ideas were provided by Lars Velten (Steinmetz lab). We first load the dataset.

```
proteins <- read_csv("http://www-huber.embl.de/users/klaus/BasicR/proteins.csv")
sample_n(proteins, 4)
```

```
#> Source: local data frame [4 x 5]
#>
#>            Condition   min Target    Signal    Sigma
#>                (chr) (int)  (chr)     (dbl)    (dbl)
#> 1 40ng/mL HGF + AKTi    10   pMEK  1.65e+08 39256897
#> 2 40ng/mL HGF + AKTi     5   pAKT  1.62e+08 98974062
#> 3        10ng/mL HGF     0  pERK2 -1.91e+08 29899056
#> 4        10ng/mL HGF    60  pERK2  2.55e+08 29718346
```

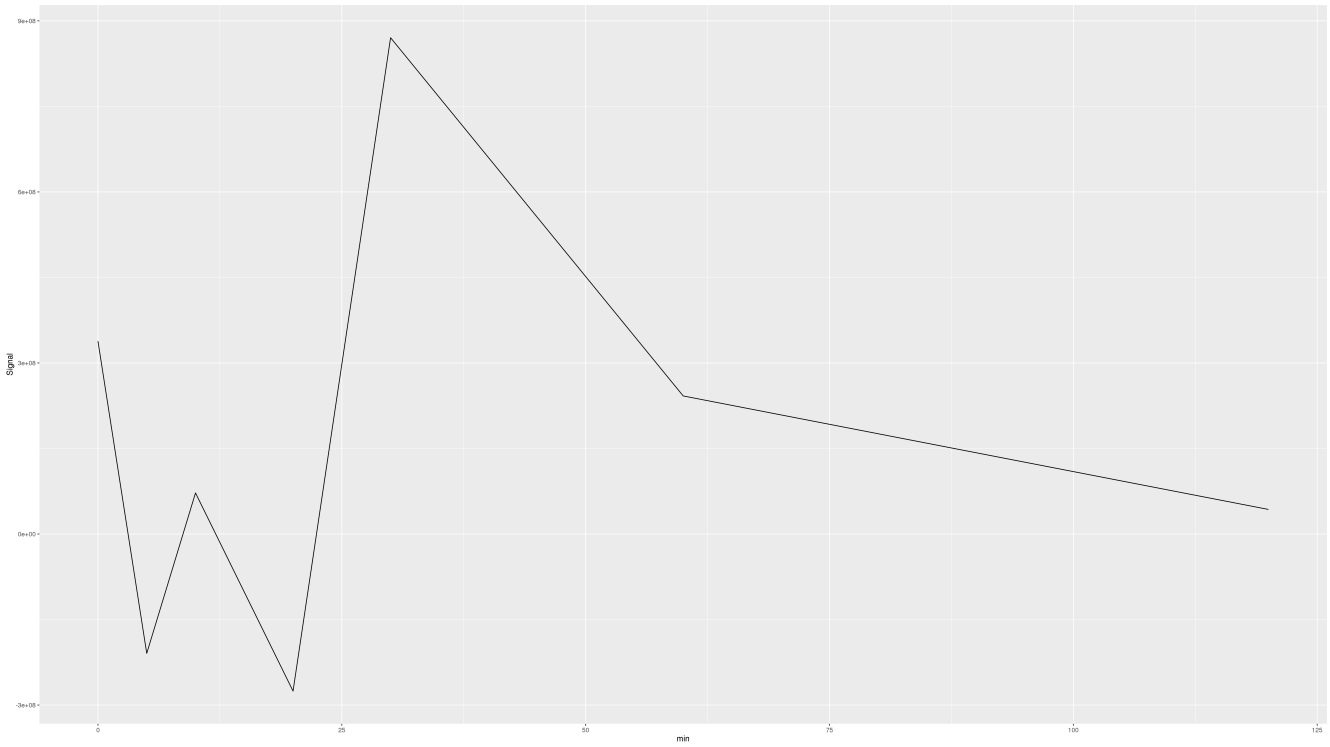
```
proteins_pMek <- subset(proteins, Target == "pMEK")
proteins_pMek_sub <- subset(proteins_pMek, Condition == "10ng/mL HGF")
```

We can start simple by only looking at the first condition of the "pMEK" protein target for now. We produce a line plot of the signal across time. In this plot we use the function `qplot` from the *ggplot2*, which is very similar to the standard *R* function `plot`.

```
proteins_pMek_sub
```

```
#>       Condition min Target    Signal    Sigma
#> 103 10ng/mL HGF   0   pMEK  3.38e+08 31005696
#> 104 10ng/mL HGF   5   pMEK -2.09e+08 31400418
#> 105 10ng/mL HGF  10   pMEK  7.20e+07 31199120
#> 106 10ng/mL HGF  20   pMEK -2.76e+08 31015194
#> 107 10ng/mL HGF  30   pMEK  8.70e+08 31140886
#> 108 10ng/mL HGF  60   pMEK  2.42e+08 31040783
#> 109 10ng/mL HGF 120   pMEK  4.31e+07 31072343
```

```
qplot(min, Signal, data = proteins_pMek_sub, geom = "line")
```



9.3 Gathering and spreading of data frames

The data table we loaded was already suitable for the plot we wanted to produce since every line represented exactly one observation. However, this is not necessarily the case and we might want to represent the different time points by different columns, not just a single one.

In time series analysis parlance our current data would be in "long" format, but we might want to transform it into a "wide" format, with a separate column for every time point. For example, the package *tidyr* allows you to do this. For example, *ggplot2* usually requires "long" formats, which can be obtained by using the function `gather`. Thus a "gathered" data frame corresponds to a "long" format. "Wide" formats can be computed using the function `spread`.

As an example we will now represent every time point of our data frame as a single column.

Note that the wide format is only compatible with a single numerical target variable, so we only include signal as a variable here.

In summary, *tidyr* has two main functions:

- `gather()` takes multiple columns, and gathers them into key–value pairs: it makes "wide" data longer.
- `spread()` takes two columns (key & value) and spreads into multiple columns, it makes "long" data wider.

Since it works with key–value pairs, *tidyr* also provides the functions `separate()` and `extract()` which makes it easier to pull apart a column that represent multiple variables. The complement to `separate()` is `unite()`.

We first remove the `Signal` column and the "spread" the minute column.

```
proteins_spread <- proteins %>%
  select(-Sigma) %>%
  spread(key = min, value = Signal)

 sample_n(proteins_spread, 4)
```

```
#>            Condition Target         0         5        10        20        30
```

```
#> 18        80ng/mL HGF  pERK1  6.22e+08 -6.58e+07 5.43e+08 -2.90e+08 -1.69e+08
#> 10 40ng/mL HGF + AKTi  pERK1 -1.04e+09 -7.92e+08 4.77e+08        NA  1.44e+07
#> 12 40ng/mL HGF + AKTi   pMEK -1.30e+08  1.83e+08 1.65e+08        NA  6.47e+08
#> 1         10ng/mL HGF   pAKT -6.71e+08  5.68e+08 1.05e+09 -3.23e+08 -9.00e+08
#>           45       60       120
#> 18 7.42e+08 1.31e+08 -9.58e+08
#> 10       NA 6.69e+08 -6.29e+08
#> 12       NA 5.57e+07  8.62e+08
#> 1        NA 3.17e+08 -4.69e+08
```

We can now gather the data frame again. For gathering, we specify that we want to "gather" time columns again by excluding the `Target` and `Condition` columns.

A factor column is then added to indicate to which former column a measurement belongs to. In our case this is the time point. Another useful function is `arrange` which allows you to reorder the data frame according to certain columns.

```
proteins_gathered <- proteins_spread %>%
  gather(key = min, value = Signal, -Target, -Condition)

sample_n(proteins_gathered, 4)
```

```
#>              Condition Target min    Signal
#> 55  40ng/mL HGF + MEKi  pERK2  10 -7.78e+08
#> 34  40ng/mL HGF + MEKi  pERK1   5 -3.60e+07
#> 131 40ng/mL HGF + AKTi  pERK2  60 -1.94e+08
#> 136 40ng/mL HGF + MEKi   pMEK  60  1.61e+09
```

9.4 Another example: TSS plots*

In this example, we have a table summarizing the coverage around 150 transcription start sites of a ChiPSeq experiment with 3 conditions: Put simply, it has been counted how often a ChIP–Seq fragment overlapped the genomic positions around the TSSs.

A certain radiation dose has been applied to mouse oocytes for one / four hours respectively and there is one "mock IP" control sample (WT). The condition is saved in a column `time`, the ENSEMBL ID in `geneID` and the other columns give the position relative to the TSS. It is hypothesized that increasing radiation will increase the binding of the transcription factor. Thus, as time progresses a distinct peak should become visible upstream of the TSS indicating the binding of the transcription factor to the promoters of the genes.

The data has been provided by Elisabeth Zielonka from the Hentze lab.

```
covs <- read_csv(url("http://www-huber.embl.de/users/klaus/BasicR/DataTSS/covs.csv"),
                 progress = FALSE)
names(covs) <- c("geneID", setdiff(seq(-3000, 3000), 0), "time")
sample_n(covs,10)[,1:5]
```

```
#> Source: local data frame [10 x 5]
#>
#>                geneID -3000 -2999 -2998 -2997
#>                 (chr) (int) (int) (int) (int)
#> 1  ENSMUSG00000003873     2     2     2     2
#> 2  ENSMUSG00000028893     0     0     0     0
#> 3  ENSMUSG00000054072     0     0     0     0
#> 4  ENSMUSG00000031480     1     1     1     0
#> 5  ENSMUSG00000029561     0     0     0     0
#> 6  ENSMUSG00000027663     0     0     0     0
```

```
#> 7  ENSMUSG00000027381     3     3     3     3
#> 8  ENSMUSG00000027356     4     4     4     4
#> 9  ENSMUSG00000036278     3     3     3     3
#> 10 ENSMUSG00000020299     3     3     3     3
```

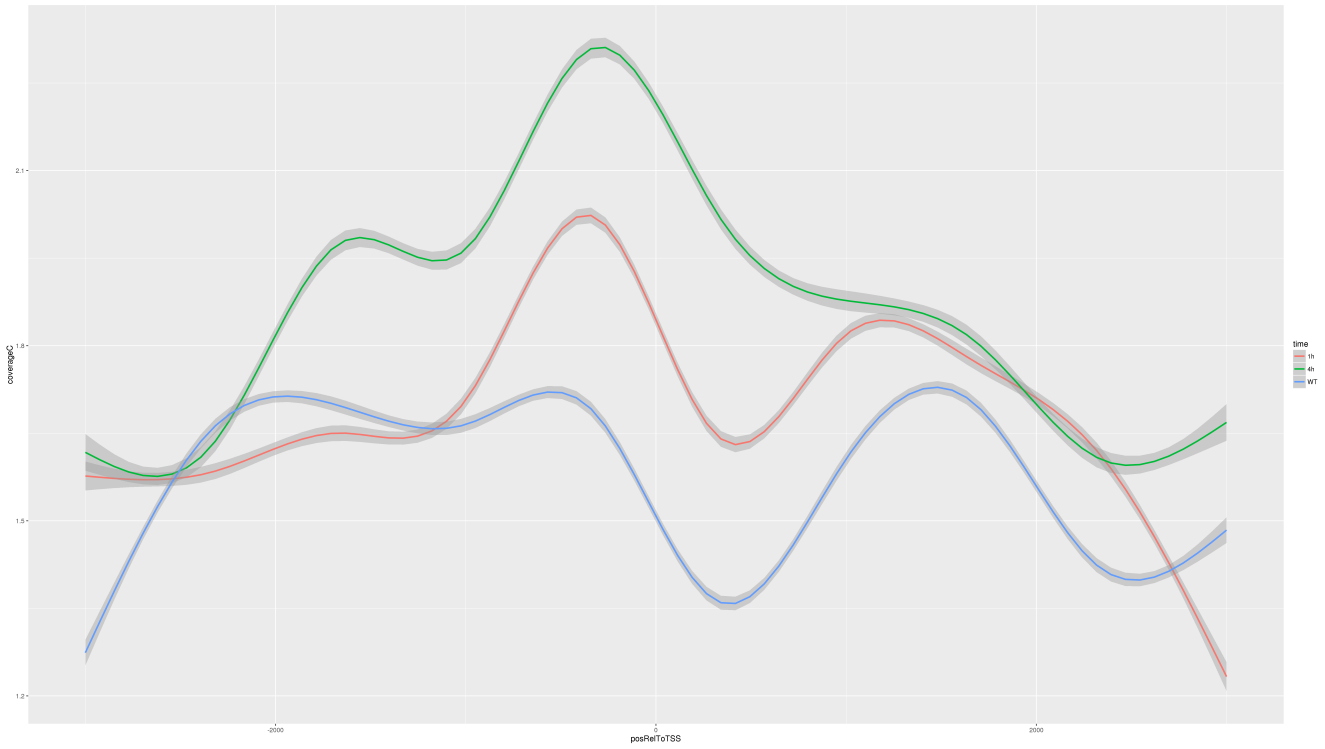
We clearly see that the data is in long format, since each position has its own column. In order to turn it into a wide format, we need to gather the position columns. This can be easily achieved by a call to gather, excluding

We then apply the mean per position and plot a smoothed version of the curves.

```
data_gathered <- covs %>%
    gather(key = "pos_rel_to_tss", value = "coverage", -geneID, -time)

data_gathered$pos_rel_to_tss <- as.integer(data_gathered$pos_rel_to_tss)

sample_n(data_gathered, 10)
```

```
#> Source: local data frame [10 x 4]
#>
#>                geneID  time pos_rel_to_tss coverage
#>                 (chr) (chr)          (int)    (int)
#> 1  ENSMUSG00000069874    1h           2721        0
#> 2  ENSMUSG00000064247    4h           2231        0
#> 3  ENSMUSG00000021798    4h            258        1
#> 4  ENSMUSG00000026395    4h            343        0
#> 5  ENSMUSG00000031480    1h           2542        1
#> 6  ENSMUSG00000024769    4h           2390        0
#> 7  ENSMUSG00000047810    4h           -301        2
#> 8  ENSMUSG00000042179    4h           -287        1
#> 9  ENSMUSG00000027663    1h           2302        1
#> 10 ENSMUSG00000021451    1h          -1741        2
```

```
covs_collapsed <- data_gathered %>%
        group_by(time, pos_rel_to_tss) %>%
                  summarize(coverageC = mean(coverage))

qplot(pos_rel_to_tss, coverageC,  color = time,
      data=covs_collapsed, geom="smooth")
```



We can see that the binding of the transcription factor to the promoter region increases over time.

10 String handling and manipulations

R has some functions which can be useful for text manipulation. In data analysis you will often need to create compound strings out of existing ones or extract information from a string, e.g. a file name.

Specifically, we will look at some of the capabilities of the package *stringr*. Check out the documentation of this package for more string manipulation capabilities.

10.1 Extracting information from a string

Lets assume we have the following image filename, which contains various information: the experimental series, the green color of a protein used and the glucose was used in the cell culture medium:

```
fName <- "tau138MGFP_Glu.lif - Series004 green_measure.tif"
```

In order to extract the information from the string, we need to split into several parts. This is done by defining a splitting pattern. Ideally, the parts of the string are separated by the same character. However, in our example this is not the case, we spaces, hyphens, underscores and dots.

Thus we have to use a so–called regular expression to split up the string. A thorough introduction to them is beyond the scope of this lab. A nice introduction to them can be found here. They allow for a very flexible description of text patterns. For example, the square brackets in the pattern "[- _.]" mean to select any of the three characters in within the brackets. This result of the splitting operation is a list that contains a vector for each string we splitted.

```
f_name_split <- str_split(string=fName, pattern = "[ - _ .]")
f_name_split
```

```
#> [[1]]
#> [1] "tau138MGFP" "Glu"        "lif"        "-"          "Series004"  "green"
#> [7] "measure"    "tif"
```

10.2 Creating compound strings from input strings

Now that we splitted the filename, we might wan to extract the information and create a new string. This is done with the functions `paste` and `paste0` which paste strings together with or without spaces. For example, we can create a new string containing the series, the color and medium condition, separated by double–hyphens. In order to do this, we supply the vector with the corresponding entries and set a character string to separate them with the `collapse` option.

```
f_name_split[[1]][c(5,2,6)]
#> [1] "Series004" "Glu"       "green"
paste0(f_name_split[[1]][c(5,2,6)], collapse = "--")
#> [1] "Series004--Glu--green"
```

11 Case studies in data handling*

11.1 Extracting information from file names

The first data example was provided by Michele Christovoa (Müller lab) and contains .csv files with image feature readouts from Microscopy. The image file names given in the first column of the table contain information about the medium (glucose or galactose) and about the color of the protein (green or red).

The goal is now to extract this information from the table in an automated fashion.

```
cell_imaging <- read_csv(url("http://www-huber.embl.de/users/klaus/BasicR/DataCellImaging/cellImaging.csv"))

head(cell_imaging)
#> Source: local data frame [6 x 7]
#>
#>                                              Label  Area  Mean StdDev   Min   Max
#>                                              (chr) (dbl) (dbl)  (dbl) (int) (int)
#> 1 tau138MGFP_Gal.lif - Series002 green_measure.tif  1381  1138   1186     0  7284
#> 2   tau138MGFP_Gal.lif - Series002 red_measure.tif  1381   832    915     0  6851
#> 3 tau138MGFP_Gal.lif - Series004 green_measure.tif  2146  1080   1111     0  6628
#> 4   tau138MGFP_Gal.lif - Series004 red_measure.tif  2146   807    878     0  6975
#> 5 tau138MGFP_Gal.lif - Series006 green_measure.tif  2063  1083   1106     0  6026
#> 6   tau138MGFP_Gal.lif - Series006 red_measure.tif  2063   902    991     0  9650
#> Variables not shown: Median (int)
```

Exercise: Handling cell imaging data

(*a*) Use the file names in the column `Label` to extract the color (green or red) and the medium (Glu or Gal). HINT: Use an appropriate split– command and then use `map` with a custom function to extract the info!

(*b*) Add columns `gal_glu` and `green_red` with the function `mutate` to the data frame that code for membership of the cell in each of the four groups
HINT: Use the function `str_match` on the file names and an `ifelse` statement. Be careful: `str_match` returns a matrix so subset the result accordingly.

(*c*) Group the data by the columns `gal_glu` and `green_red` and then compute the mean `Mean` per group using `summarize`.

11.2 Organizing data that comes in several files

The next example data set was provided by Iana Kalinina from the Nedelec/Merten labs. It consists of several proteomics experiments where always three samples have been analyzed in one experiment. The samples correspond either to an extract from a mixture of eggs or single eggs. You can download the data here.

We extract it to a folder "DataProteomics" in the current working directory. In order to get an overview of the data, we list all the files

```
data_dir <- file.path("DataProteomics")
list.files(data_dir)
```

```
#>  [1] "OO1O2_ratio.csv"  "PPP_ratio.csv"    "ProteinList.csv"  "QQQ_ratio.csv"
#>  [5] "RR5R6_ratio.csv"  "SampleLegend.csv" "SeSeSe_ratio.csv" "TT6T7_ratio.csv"
#>  [9] "TTT_ratio.csv"    "UUU_ratio.csv"    "UXZ_ratio.csv"    "VV8V5_ratio.csv"
#> [13] "VVV_ratio.csv"    "ZZ6Z8_ratio.csv"
```

The proteomics data for the individual experiments is stored in separate .csv files. The file ProteinList.csv contains the protein annotation and the file SampleLegend.csv a description of the samples of the experiments. All of the files that contain the actual data have the suffix "ratios". Thus we extract them first. Then, we extract the names of the experiments from them with the function `str_sub`.

```
# get the proteomics data files
prot_data <- file.path(data_dir, list.files(data_dir, pattern="ratio"))
prot_data
```

```
#>  [1] "DataProteomics/OO1O2_ratio.csv"  "DataProteomics/PPP_ratio.csv"
#>  [3] "DataProteomics/QQQ_ratio.csv"    "DataProteomics/RR5R6_ratio.csv"
#>  [5] "DataProteomics/SeSeSe_ratio.csv" "DataProteomics/TT6T7_ratio.csv"
#>  [7] "DataProteomics/TTT_ratio.csv"    "DataProteomics/UUU_ratio.csv"
#>  [9] "DataProteomics/UXZ_ratio.csv"    "DataProteomics/VV8V5_ratio.csv"
#> [11] "DataProteomics/VVV_ratio.csv"    "DataProteomics/ZZ6Z8_ratio.csv"
```

```
namesExp <- str_sub(sapply(str_split(prot_data, "_"), "[", 1), 16)
namesExp
```

```
#>  [1] "OO1O2"  "PPP"    "QQQ"    "RR5R6"  "SeSeSe" "TT6T7"  "TTT"    "UUU"    "UXZ"
#> [10] "VV8V5"  "VVV"    "ZZ6Z8"
```

We now import the data using `lapply` on the file names and name the columns in the imported table with the appropriate sample number.

```
prot_data  <- map_df(prot_data, read_csv, .id = "experiment"
, col_names = c("S1", "S2", "S3"))

prot_data <- mutate(prot_data, experiment = mapvalues(experiment,
                   from = unique(experiment), to = namesExp))
```

In order to have the annotation of the samples and the proteins, we also import this data now.

```
sample_metadata <- read_csv(file.path(data_dir, "SampleLegend.csv"),
                            skip=1)

names(sample_metadata)[c(1,3:5)] <- c("sample_name", "S1", "S2", "S3")
sample_n(sample_metadata, 10)
```

```
#> Source: local data frame [10 x 5]
#>
#>    sample_name        condition      S1      S2      S3
#>          (chr)            (chr)   (chr)   (chr)   (chr)
#> 1          UXZ positive control extract extract extract
#> 2       SeSeSe negative control     egg     egg     egg
#> 3          QQQ negative control extract extract extract
#> 4        ZZ6Z8       experiment extract     egg     egg
#> 5          UUU negative control extract extract extract
#> 6          VVV negative control extract extract extract
#> 7        OO1O2       experiment extract     egg     egg
#> 8          TTT negative control extract extract extract
#> 9        VV8V5       experiment extract     egg     egg
#> 10         PPP negative control extract extract extract
```

```
# load protein identifiers
protein_list <-  read_csv(file.path(data_dir, "ProteinList.csv"),
                          skip=0)
sample_n(protein_list, 10)
```

```
#> Source: local data frame [10 x 2]
#>
#>    Accession
#>        (chr)
#> 1     Q5EAZ3
#> 2     Q52KZ0
#> 3     Q6PB03
#> 4     Q7ZY46
#> 5     Q3B8J8
#> 6     Q3B8C2
#> 7     Q6NTJ3
#> 8     B1WBD3
#> 9     Q66J28
#> 10    Q6DJI0
#> Variables not shown: Description (chr)
```

We now add the protein accessions as row names to the data and tidy the sample table in such a way that we have one line per unique sample.

```
## add protein names as rownames
prot_data$Accession <- rep(protein_list$Accession, length(namesExp))

# modify sample metadata to have one line per sample
sample_metadata <- gather(sample_metadata, key = "sample", value= "type", S1:S3)
sample_metadata <-  mutate(sample_metadata,
                           sample_id = paste0(sample_name, "_", sample))
sample_metadata <- arrange(sample_metadata, sample_id)



sample_metadata
```

```
#> Source: local data frame [36 x 5]
#>
#>    sample_name        condition sample    type sample_id
#>          (chr)            (chr)  (chr)   (chr)     (chr)
```

```
#> 1        OO1O2       experiment     S1 extract  OO1O2_S1
#> 2        OO1O2       experiment     S2     egg  OO1O2_S2
#> 3        OO1O2       experiment     S3     egg  OO1O2_S3
#> 4          PPP negative control     S1 extract    PPP_S1
#> 5          PPP negative control     S2 extract    PPP_S2
#> 6          PPP negative control     S3 extract    PPP_S3
#> 7          QQQ negative control     S1 extract    QQQ_S1
#> 8          QQQ negative control     S2 extract    QQQ_S2
#> 9          QQQ negative control     S3 extract    QQQ_S3
#> 10       RR5R6       experiment     S1 extract  RR5R6_S1
#> ..         ...              ...    ...     ...       ...
```

We now have created a tidy data table from the input data that we can use for downstream analysis.

```
sample_n(prot_data, 10)
```

```
#> Source: local data frame [10 x 5]
#>
#>    experiment    S1    S2    S3 Accession
#>         (chr) (dbl) (dbl) (dbl)     (chr)
#> 1       ZZ6Z8   NaN   NaN   NaN    Q6DKB4
#> 2       ZZ6Z8   NaN   NaN   NaN    Q91695
#> 3       OO1O2 4.608  1.64 9.835    Q6DCS9
#> 4         UUU 0.373  1.43 0.563    Q7ZY16
#> 5       RR5R6   NaN   NaN   NaN    Q6DCW9
#> 6       ZZ6Z8 0.407  1.37 0.498    Q6DE60
#> 7         PPP   NaN   NaN   NaN    B4F6R2
#> 8         UUU   NaN   NaN   NaN    A1E8I5
#> 9       ZZ6Z8   NaN   NaN   NaN    Q66IV5
#> 10        PPP 5.531   NaN   NaN    Q4QR58
```

12 Answers to exercises

Exercise: Bodyfat Data

(*a*) Calculate mean and sd for all the variables in the data set. HINT: Use an appropriate `apply` function.
(*b*) Find the indexes of all men in the data set that are relatively small, i.e, who are less than `mean(height) - sd(height)` tall.
(*c*) Compute a similar index for weight, i.e. find the light people.
(*d*) Find the small and light people, i.e. find the intersection of the two index-sets. Use the logical operator `&`. and .

Solution: Bodyfat Data

```
#a
##############################################################################
sapply(bodyfat, mean)
sapply(bodyfat, sd)

# or

map_dbl(bodyfat, mean)
map_dbl(bodyfat, sd)
```

```
#b
#########################################################################
small.idx <-  bodyfat$height < mean(bodyfat$height) - sd(bodyfat$height)
subset(bodyfat, small.idx )
## or
filter(bodyfat, small.idx)

#c
#########################################################################
light.idx <-  bodyfat$weight < mean(bodyfat$weight) - sd(bodyfat$weight)
subset(bodyfat, light.idx)
## or
filter(bodyfat, light.idx)

#d small and light people
#########################################################################
small.and.light.idx  <- small.idx & light.idx
subset(bodyfat, small.and.light.idx)
# or
filter(bodyfat, small.idx, light.idx)
```

Exercise: HTS handling

(*a*) load the HTS data from

http://www-huber.embl.de/users/klaus/BasicR/HTSdata.RData.

(*b*) Compute the mean (function `mean`) and standard deviation (function `sd`) of the cell number for every single plate.
(*c*) Using the function `identity` in connection with the dot operator and the `do`, create a list of vectors containing only the non–infinite transport ratios for every single plate, replacing the non-finite ratios by zero.
HINT: Use the function `is.finite` as well as an `ifelse` command.

Solution: HTS handling

```
#a
#########################################################################
load(url("http://www-huber.embl.de/users/klaus/BasicR/HTSdata.RData"))

#b
#########################################################################
HTS_n_cs <-  HTSdata %>%
group_by(plate, replicate) %>%
summarize(mean_CN = mean(CellNumber, na.rm = T),
          sd_CN = sd(CellNumber, na.rm = T))
HTS_n_cs

#c
#########################################################################
TransR <- HTSdata

TransR$TransR <- ifelse(is.finite(TransR$TransR), TransR$TransR ,0)
```

```
TransRlist <-  TransR %>%
group_by(plate, replicate)  %>%
dplyr::select(TransR)  %>%
do(TRlist = identity(.))

TransRlist$TRlist[[1]]
```

Exercise: Handling cell imaging data

(*a*) Use the file names in the column `Label` to extract the color (green or red) and the medium (Glu or Gal). HINT: Use an appropriate split– command and then use `sapply` with a custom function to extract the info!

(*b*) Add columns `gal_glu` and `green_red` with the function `mutate` to the data frame that code for membership of the cell in each of the four groups
HINT: Use the function `str_match` on the file names and an `ifelse` statement. Be careful: `str_match` returns a matrix so subset the result accordingly.

(*c*) Group the data by the columns `gal_glu` and `green_red` and then compute the mean `Mean` per group using `summarize`.

Solution: Handling cell imaging data

```
#a

label_Info <- str_split(string=cell_imaging$Label, pattern = "[ - _ .]")
extractedInfo <- map_df(label_Info,
                        function(x){data.frame(series = x[5],
                                               medium = x[2],
                                               label = x[6])})

head(extractedInfo)

#b

## add columns to the data thaht give the categories

cell_imaging <- mutate(cell_imaging, green_red=ifelse(is.na(str_match(Label, "green")[,1]),
                                                      "Red", "Green"),
               gal_glu=ifelse(is.na(str_match(Label, "Gal")[,1]), "Glu", "Gal"))
## check variable types
glimpse(cell_imaging)

#c

## group by category and get the mean
cell_imaging %>%
  group_by(green_red, gal_glu) %>%
  summarize(mean(Mean))
```

Exercise: Handling the Golub data

(*a*) Print the gene expression values of Gene CCND3 for all AML patients using the factor `gol.fac`.

(*b*) For many types of computations it is very useful to combine a factor with the apply functionality: Use an apply function to compute the mean gene expression over the ALL and AML patients for each of the genes.
(*c*) Order the data matrix according to the mean expression values for ALL patients in decreasing order and give the names of the genes with largest mean expression value for ALL patients.

Solution: Handling the Golub data

```
#a
#########################################################################
golub[1042,gol.fac=="AML"]
#b
#########################################################################
meanALL <- apply(golub[,gol.fac=="ALL"], 1, mean)
meanAML <- apply(golub[,gol.fac=="AML"], 1, mean)

#c
#########################################################################
head(meanALL[order(meanALL, decreasing = TRUE)], 100)
head(golub[order(meanALL, decreasing = TRUE),  ], 20)


(golub.gnames[order(meanALL, decreasing = TRUE),2])[1:3]
```

Exercise: Handling Bioconductor expression sets

(*a*) obtain sample expression set object from the *Biobase* Bioconductor package using `data(sample.ExpressionSet)` and extract the contained gene expression data. Checkout the "vignette" on expression sets of the package *Biobase* via `browseVignettes("Biobase")` to learn more about expression sets in *R*.
Use the function `slotNames()` to obtain an overview of the elements or "slots" of the object.
(*b*) Extract a description of the experiment from the object. Which variables have been measured on which samples? Is there any metadata on the variables?
(*c*) Which microarray was used in the experiment? Which "features" (probes) are measured?
(*d*) How many control probes are contained in the data set? HINT: Their names starts with "AFFX", use the function `grep` to obtain them. They are usually filtered out prior to further analysis.
(*e*) Find out how to obtain a `phenoData` table, i.e. a table containing the sample annotation.

Solution: Handling Bioconductor expression sets

```
#a
#########################################################################
library(Biobase)
data(sample.ExpressionSet)
sample.ExpressionSet
exprs(sample.ExpressionSet)
slotNames(sample.ExpressionSet)

#b
#########################################################################
varLabels(sample.ExpressionSet)
sampleNames(sample.ExpressionSet)
varMetadata(sample.ExpressionSet)
```

```
#c
##########################################################################
annotation(sample.ExpressionSet)
featureNames(sample.ExpressionSet)

#d
##########################################################################
controls <- grep("AFFX", featureNames(sample.ExpressionSet), value = TRUE)
str(controls)

#e
##########################################################################
phenoData(sample.ExpressionSet)
head(pData(sample.ExpressionSet))
```

Session Info

```
toLatex(sessionInfo())
```

- R version 3.2.2 (2015-08-14), x86_64-pc-linux-gnu
- Locale: LC_CTYPE=en_US.UTF-8, LC_NUMERIC=C, LC_TIME=en_US.UTF-8, LC_COLLATE=en_US.UTF-8, LC_MONETARY=en_US.UTF-8, LC_MESSAGES=en_US.UTF-8, LC_PAPER=en_US.UTF-8, LC_NAME=C, LC_ADDRESS=C, LC_TELEPHONE=C, LC_MEASUREMENT=en_US.UTF-8, LC_IDENTIFICATION=C
- Base packages: base, datasets, graphics, grDevices, methods, parallel, stats, stats4, utils
- Other packages: AnnotationDbi 1.32.3, Biobase 2.30.0, BiocGenerics 0.16.1, BiocInstaller 1.20.1, DESeq2 1.10.1, dplyr 0.4.3, GenomeInfoDb 1.6.3, GenomicFeatures 1.22.13, GenomicRanges 1.22.4, ggplot2 2.1.0, IRanges 2.4.8, knitr 1.12.3, magrittr 1.5, multtest 2.26.0, openxlsx 3.0.0, pasilla 0.10.0, plyr 1.8.3, purrr 0.2.1, Rcpp 0.12.3, RcppArmadillo 0.6.600.4.0, readr 0.2.2, reshape2 1.4.1, S4Vectors 0.8.11, stringr 1.0.0, SummarizedExperiment 1.0.2, TeachingDemos 2.10, tidyr 0.4.1, TxDb.Dmelanogaster.UCSC.dm3.ensGene 3.2.2
- Loaded via a namespace (and not attached): acepack 1.3-3.3, annotate 1.48.0, assertthat 0.1, BiocParallel 1.4.3, BiocStyle 1.8.0, biomaRt 2.26.1, Biostrings 2.38.4, bitops 1.0-6, cluster 2.0.3, codetools 0.2-14, colorspace 1.2-6, curl 0.9.6, DBI 0.3.1, DESeq 1.22.1, digest 0.6.9, evaluate 0.8.3, foreign 0.8-66, formatR 1.3, Formula 1.2-1, futile.logger 1.4.1, futile.options 1.0.0, genefilter 1.52.1, geneplotter 1.48.0, GenomicAlignments 1.6.3, grid 3.2.2, gridExtra 2.2.1, gtable 0.2.0, highr 0.5.1, Hmisc 3.17-2, labeling 0.3, lambda.r 1.1.7, lattice 0.20-33, latticeExtra 0.6-28, lazyeval 0.1.10, locfit 1.5-9.1, MASS 7.3-45, Matrix 1.2-4, mgcv 1.8-12, munsell 0.4.3, nlme 3.1-126, nnet 7.3-12, R6 2.1.2, RColorBrewer 1.1-2, RCurl 1.95-4.8, rpart 4.1-10, Rsamtools 1.22.0, RSQLite 1.0.0, rtracklayer 1.30.3, scales 0.4.0, splines 3.2.2, stringi 1.0-1, survival 2.38-3, tools 3.2.2, XML 3.98-1.4, xtable 1.8-2, XVector 0.10.0, zlibbioc 1.16.0