Proteomics Data Analysis Workshop

Paul Stewart

2025-07-17

Introduction

Set up RStudio

This assumes you have followed the instructions here https://posit.co/download/rstudio-desktop/ and downloaded and installed both R and RStudio.

Before we start writing or editing code, there are a few settings in RStudio that can make your experience much more comfortable. One helpful option is to turn on Soft Wrap, so that long lines of code or text automatically wrap within the editor window instead of scrolling off to the right. You can enable this by going to Tools -> Global Options -> Code -> Editing and checking Soft-wrap source files.

Another useful feature is Rainbow Parentheses, which color-matches pairs of parentheses and brackets to help you keep track of nesting. This is especially helpful when working with complex expressions. You can turn this on under Tools -> Global Options -> Code -> Display by checking Rainbow parentheses.

Finally, you may want to change the background color of the RStudio editor to something easier on the eyes. To do this, go to Tools -> Global Options -> Appearance and choose a different editor theme, such as "Tomorrow Night" for a dark background or "Textmate" for a clean, light look. This is a personal preference, but it can make coding for long periods much easier on the eyes.

R Markdown

This is an R Markdown file (or the PDF report generated from it, depending on what you are reading). R Markdown lets us combine regular text and executable code in a single document. You can run code chunks interactively or use the Knit button to create a formatted report (such as a PDF, HTML, or Word document). We will focus first on running code interactively and touch on reports towards the end. For the most part, code run through an R Markdown file are equivalent to them being run in the console.

In an R Markdown file, we use text characters to format other text that appear once the document is rendered into the formatted report.

- Lines that begin with # are section headers.
- # creates a main section
- ## creates a subsection
- *'s around some text will *italicize* it
- A single - followed by a space will make a bulleted list

Code is written in code chunks, which are formatted like this:

```
summary(cars)  # cars is a built in dataset
```

```
##      speed           dist
##  Min.   : 4.0   Min.   :  2.00
##  1st Qu.:12.0   1st Qu.: 26.00
##  Median :15.0   Median : 36.00
```

```
##  Mean   :15.4   Mean   : 42.98
##  3rd Qu.:19.0   3rd Qu.: 56.00
##  Max.   :25.0   Max.   :120.00
```

You can run chunks by pressing the green arrow on the right hand side of the chunk, by pressing Cmd + Enter on a Mac or Ctrl + Enter on Windows. When running a chunk, the output appears underneath the chunk.

Read in and view data

Set the working directory

It is important that R knows where to look for your files. This is controlled by the working directory. If R is giving you an error about a file not existing but you are sure it does, then the most common problem is that the working directory was not set correctly (second most common is that there is a typo in your file location).

If we are using an R Project (which we are), the working directory is automatically set to the folder that contains the .Rproj file. However, since we are working in an R Markdown document, it may default the working directory to the location of the .Rmd file instead. This can be confusing, especially when code behaves differently depending on whether it is run in the console or inside chunks.

To fix this, we will explicitly set the working directory using the here package. This ensures that all file paths are interpreted relative to the folder where the R Project is located, rather than where the .Rmd file happens to live.

To install the here package, run this command in the console. The chunk below has the option eval = FALSE, which means it will not be evaluated when we knit the document or run all chunks at once. However, we can still run it manually using the green arrow or by placing your cursor in the chunk and pressing Ctrl + Enter (or Cmd + Enter on Mac).

```
install.packages("here")
```

By default, this installs the package from CRAN, the Comprehensive R Archive Network (https://cran.r-project.org), which is the main repository for R packages (in addition to hosting the R package and source code).

Once it is installed, load the here package using

```
library(here)
```

```
## here() starts at /Users/paulstewart/Library/CloudStorage/Box-Box/projects/upag_2025
```

We will now add a setup chunk that sets the root directory for the entire document. It is important that this chunk option has “setup” included for this to work:

```
knitr::opts_knit$set(root.dir = here::here())
```

Experimental data

Now let us load the protein abundance data (which we may also refer to as protein expression). We give it a short, descriptive variable name like prot so we know what the contents are without having to look at the contents. Header = TRUE means that the first row will be used for the column names, and row.names = 1 indicates that the first row will be used for the row names:

```
prot = read.delim("data/dia_lfq_protein_updated_headers_2025-07-15.txt",
    header = TRUE, row.names = 1)
```

Let us also load the meta data (subsequently referred to as “meta” for shorthand):

```
meta = read.delim("data/sample_meta_2025-07-15.txt", header = TRUE)
```

We can look at the first few rows with the head() function:

```
head(prot)
```

```
##             KO_TP_1  KO_TP_2  KO_TP_3  KO_TP_4  KO_TP_5 KO_baseline_1
## A0A087WNL7 11474.10 11282.30 11643.90 11380.40 10295.00       9160.92
## A0A087WPH7 30820.80 26099.50 26100.00 21899.40 12647.90      13000.30
## A0A087WQ16  3940.87  3468.04  4029.22  4074.38  3945.08       5428.93
## A0A087WQ32  3105.63  2958.71  3181.00  3436.47  2813.20       2402.61
## A0A087WQH8 24761.30 22779.10 19991.10 23988.70 21811.60      20409.80
## A0A087WQN7 15597.90 14915.60 16650.50 15111.10 15260.20      18220.00
##            KO_baseline_2 KO_baseline_3 KO_baseline_4 KO_baseline_5  WT_TP_1
## A0A087WNL7       9502.60      10262.80       8830.69       8811.47 17884.50
## A0A087WPH7      13811.50      17601.00      21117.60      16041.20 45882.20
## A0A087WQ16       5963.76       5768.15       5495.85       5258.89  5755.49
## A0A087WQ32       2231.49       2643.82       2792.94       2108.33  2445.45
## A0A087WQH8      17450.80      16773.30      19130.80      12470.50 15126.80
## A0A087WQN7      19931.70      18656.40      18633.60      18601.80 22733.40
##             WT_TP_2  WT_TP_3  WT_TP_4  WT_TP_5 WT_baseline_1 WT_baseline_2
## A0A087WNL7 18419.20 17171.40 20689.40 18759.20      12195.00      12179.40
## A0A087WPH7 26880.20 29827.60 31319.00 28735.40      21048.60      21009.90
## A0A087WQ16  6140.00  5701.61  6565.67  5523.40       6456.51       6252.31
## A0A087WQ32  2515.12  2573.07  2992.97  3270.87       1870.27       2234.11
## A0A087WQH8 26396.10 26077.90 23304.40 20414.10      13645.80      15457.50
## A0A087WQN7 22180.00 20287.60 21790.70 20775.90      20459.60      19026.50
##            WT_baseline_3 WT_baseline_4 WT_baseline_5
## A0A087WNL7      12448.30      12905.10      13872.00
## A0A087WPH7      27824.90      23312.30      28987.10
## A0A087WQ16       7033.00       6757.87       6734.30
## A0A087WQ32       3151.96       2610.88       2551.85
## A0A087WQH8      15301.20      14389.40      15566.50
## A0A087WQN7      20645.10      19840.50      19611.90
```

Given we have 20 samples this is a little crowded, and the rows wrap around so it is not very easy to interpret. Let us just look at the first five rows and first five columns using square brackets and the ":" sequence operator:

```
prot[1:5, 1:5]
```

```
##             KO_TP_1  KO_TP_2  KO_TP_3  KO_TP_4  KO_TP_5
## A0A087WNL7 11474.10 11282.30 11643.90 11380.40 10295.00
## A0A087WPH7 30820.80 26099.50 26100.00 21899.40 12647.90
## A0A087WQ16  3940.87  3468.04  4029.22  4074.38  3945.08
## A0A087WQ32  3105.63  2958.71  3181.00  3436.47  2813.20
## A0A087WQH8 24761.30 22779.10 19991.10 23988.70 21811.60
```

We can look at the column names of prot

```
head(names(prot))
```

```
## [1] "KO_TP_1"       "KO_TP_2"       "KO_TP_3"       "KO_TP_4"
## [5] "KO_TP_5"       "KO_baseline_1"
```

as well as the row names:

```
head(row.names(prot))
```

```
## [1] "A0A087WNL7" "A0A087WPH7" "A0A087WQ16" "A0A087WQ32" "A0A087WQH8"
## [6] "A0A087WQN7"
```

The meta data is narrow enough that it will not be too crowded if we show the whole thing. We can just evaluate the data frame with no other arguments to display it:

```
meta
```

```
##    Sample.number expression_column       group genotype condition replicate
## 1              1           KO_TP_1       KO_TP       KO        TP         1
## 2              2           KO_TP_2       KO_TP       KO        TP         2
## 3              3           KO_TP_3       KO_TP       KO        TP         3
## 4              4           KO_TP_4       KO_TP       KO        TP         4
## 5              5           KO_TP_5       KO_TP       KO        TP         5
## 6              6     KO_baseline_1 KO_baseline       KO  baseline         1
## 7              7     KO_baseline_2 KO_baseline       KO  baseline         2
## 8              8     KO_baseline_3 KO_baseline       KO  baseline         3
## 9              9     KO_baseline_4 KO_baseline       KO  baseline         4
## 10            10     KO_baseline_5 KO_baseline       KO  baseline         5
## 11            11           WT_TP_1       WT_TP       WT        TP         1
## 12            12           WT_TP_2       WT_TP       WT        TP         2
## 13            13           WT_TP_3       WT_TP       WT        TP         3
## 14            14           WT_TP_4       WT_TP       WT        TP         4
## 15            15           WT_TP_5       WT_TP       WT        TP         5
## 16            16     WT_baseline_1 WT_baseline       WT  baseline         1
## 17            17     WT_baseline_2 WT_baseline       WT  baseline         2
## 18            18     WT_baseline_3 WT_baseline       WT  baseline         3
## 19            19     WT_baseline_4 WT_baseline       WT  baseline         4
## 20            20     WT_baseline_5 WT_baseline       WT  baseline         5
##        color
## 1    skyblue
## 2    skyblue
## 3    skyblue
## 4    skyblue
## 5    skyblue
## 6     salmon
## 7     salmon
## 8     salmon
## 9     salmon
## 10    salmon
## 11 palegreen
## 12 palegreen
## 13 palegreen
## 14 palegreen
## 15 palegreen
## 16      plum
## 17      plum
## 18      plum
## 19      plum
## 20      plum
```

Let us use the subset() function on meta, which takes a data frame as an argument and returns a filtered subset of the original data based on some criteria:

```
meta_wt = subset(meta, genotype == "WT")
```

We can now use a very useful trick that forms the basis for most of our upcoming analyses. Say we only wanted the columns from the protein abundance for WT samples. We simply use meta_wt to get the columns we want from the protein data frame:

```
head(prot[, meta_wt$expression_column])
```

```
##             WT_TP_1  WT_TP_2  WT_TP_3  WT_TP_4  WT_TP_5 WT_baseline_1
## A0A087WNL7 17884.50 18419.20 17171.40 20689.40 18759.20      12195.00
## A0A087WPH7 45882.20 26880.20 29827.60 31319.00 28735.40      21048.60
## A0A087WQ16  5755.49  6140.00  5701.61  6565.67  5523.40       6456.51
## A0A087WQ32  2445.45  2515.12  2573.07  2992.97  3270.87       1870.27
## A0A087WQH8 15126.80 26396.10 26077.90 23304.40 20414.10      13645.80
## A0A087WQN7 22733.40 22180.00 20287.60 21790.70 20775.90      20459.60
##            WT_baseline_2 WT_baseline_3 WT_baseline_4 WT_baseline_5
## A0A087WNL7      12179.40      12448.30      12905.10      13872.00
## A0A087WPH7      21009.90      27824.90      23312.30      28987.10
## A0A087WQ16       6252.31       7033.00       6757.87       6734.30
## A0A087WQ32       2234.11       3151.96       2610.88       2551.85
## A0A087WQH8      15457.50      15301.20      14389.40      15566.50
## A0A087WQN7      19026.50      20645.10      19840.50      19611.90
```

This code produces the same result:

```
head(prot[, meta_wt[, "expression_column"]])
```

```
##             WT_TP_1  WT_TP_2  WT_TP_3  WT_TP_4  WT_TP_5 WT_baseline_1
## A0A087WNL7 17884.50 18419.20 17171.40 20689.40 18759.20      12195.00
## A0A087WPH7 45882.20 26880.20 29827.60 31319.00 28735.40      21048.60
## A0A087WQ16  5755.49  6140.00  5701.61  6565.67  5523.40       6456.51
## A0A087WQ32  2445.45  2515.12  2573.07  2992.97  3270.87       1870.27
## A0A087WQH8 15126.80 26396.10 26077.90 23304.40 20414.10      13645.80
## A0A087WQN7 22733.40 22180.00 20287.60 21790.70 20775.90      20459.60
##            WT_baseline_2 WT_baseline_3 WT_baseline_4 WT_baseline_5
## A0A087WNL7      12179.40      12448.30      12905.10      13872.00
## A0A087WPH7      21009.90      27824.90      23312.30      28987.10
## A0A087WQ16       6252.31       7033.00       6757.87       6734.30
## A0A087WQ32       2234.11       3151.96       2610.88       2551.85
## A0A087WQH8      15457.50      15301.20      14389.40      15566.50
## A0A087WQN7      19026.50      20645.10      19840.50      19611.90
```

This approach avoids hardcoding the wild type samples where we specify the column numbers:

```
head(prot[, 11:20])
```

```
##             WT_TP_1  WT_TP_2  WT_TP_3  WT_TP_4  WT_TP_5 WT_baseline_1
## A0A087WNL7 17884.50 18419.20 17171.40 20689.40 18759.20      12195.00
## A0A087WPH7 45882.20 26880.20 29827.60 31319.00 28735.40      21048.60
## A0A087WQ16  5755.49  6140.00  5701.61  6565.67  5523.40       6456.51
## A0A087WQ32  2445.45  2515.12  2573.07  2992.97  3270.87       1870.27
## A0A087WQH8 15126.80 26396.10 26077.90 23304.40 20414.10      13645.80
## A0A087WQN7 22733.40 22180.00 20287.60 21790.70 20775.90      20459.60
##            WT_baseline_2 WT_baseline_3 WT_baseline_4 WT_baseline_5
## A0A087WNL7      12179.40      12448.30      12905.10      13872.00
## A0A087WPH7      21009.90      27824.90      23312.30      28987.10
## A0A087WQ16       6252.31       7033.00       6757.87       6734.30
```

```
## A0A087WQ32       2234.11       3151.96       2610.88       2551.85
## A0A087WQH8      15457.50      15301.20      14389.40      15566.50
## A0A087WQN7      19026.50      20645.10      19840.50      19611.90
```

What if we specify columns 11-20 are originally WT, but then we change the data by adding or removing samples? Then columns 11-20 may no longer correspond to WT, and our subsequent analyses will be wrong. By using the expression_column from the meta, we will always be sure to get the wild type samples, regardless of their order. Importantly, this only works if header = TRUE was used when we read in the protein data such that meta_wt$expression_column corresponds to the header/column names of prot. If we need to delete a specific sample, then we should refer to it by its specific name (e.g. WT_TP_4) instead of the specific column (e.g. column 14).

Questions

For this and future questions, create your own chunk in this markdown file after the questions. Bonus points if you can find the keyboard shortcut for making a new chunk (hint: it is in one of the drop-down menus).

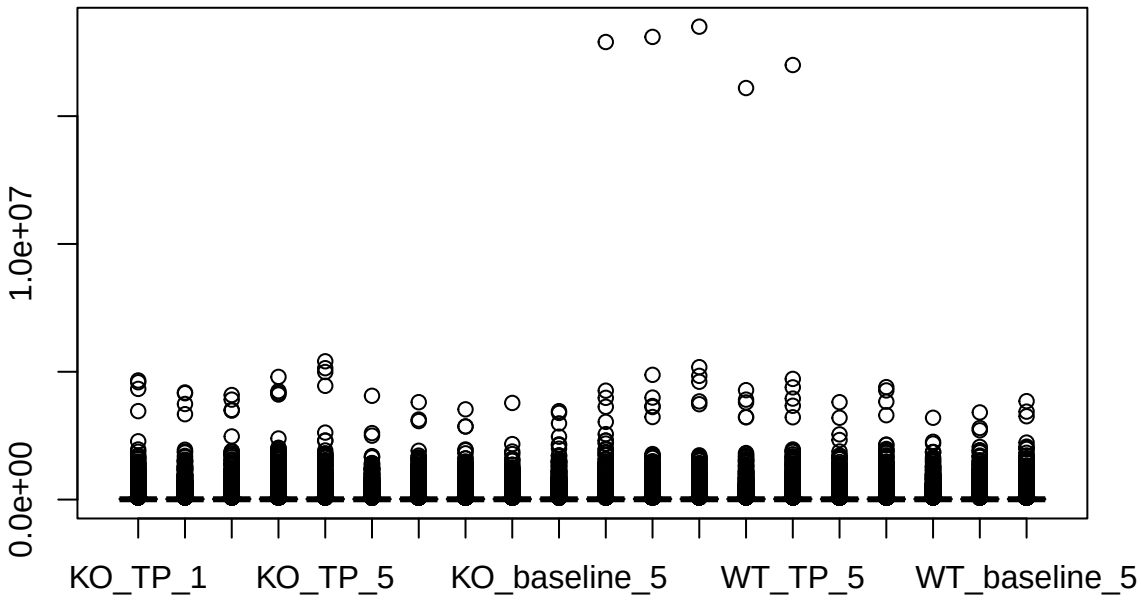
1. Data frames are examples of an object. Different objects have different properties. Prot and meta are both data frames. What happens if you put nrow(prot) in your chunk and run it? ncol(prot)? dim(prot)? summary(prot)?
2. Make a variable called fav_num, and assign your favorite number to it. What happens when you try to get the number of rows from it? What about summary()?
3. How would you display the head of only the TP samples?
4. Try running mean(prot) or sd(prot). What happens, and why? How could you make it work for a single column?
5. What does class(meta) return? What about typeof(meta)?
6. Try printing the first few row names using head(row.names(meta)). Do any of them match the column names of prot?
7. Create your own subset of meta for only the TP timepoint. Can you use it to select the corresponding columns from prot?
8. What does head(meta, 10) do? head(meta, 15)? head(meta, 21)?

Visualize data

Boxplot

R has a built-in boxplot() function that can take a data frame as input:

```
boxplot(prot[, meta$expression_column])
```



The result is not very informative. Proteomics data, like most mass spectrometry data, tends to follow an exponential distribution. This means many values cluster at the lower end, while a few extreme values stretch the plot upward, and plotting the values spans many orders of magnitude. As a result, it can be difficult to see detail in the lower part of the distribution. Additionally, the sample labels are not displaying clearly, making the plot harder to interpret.

To improve this, we will start by applying a log2 transformation to the intensity values. This transformation makes the distribution more symmetric and closer to log-normal, which helps with visualization and also allows us to use parametric statistical tests like the t-test to compare group means.

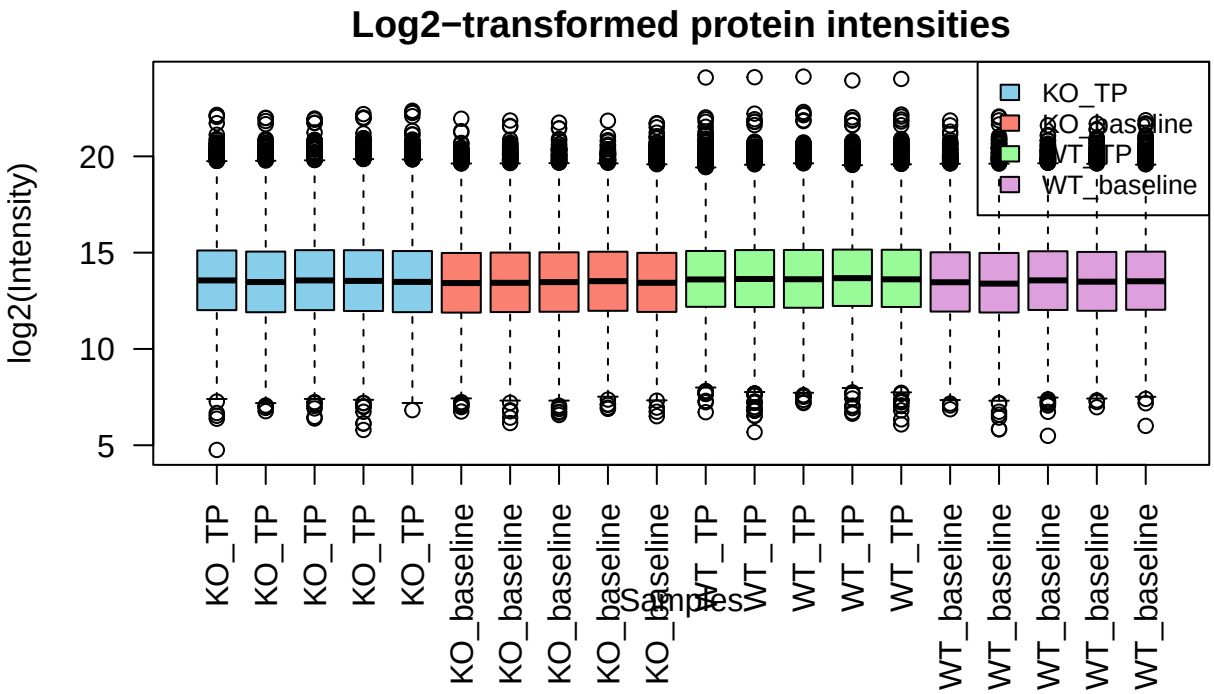
```
prot[, meta$expression_column] = log2(prot[, meta$expression_column])
prot[1:5, 1:5]
```

```
##             KO_TP_1  KO_TP_2  KO_TP_3  KO_TP_4  KO_TP_5
## A0A087WNL7 13.48609 13.46177 13.50729 13.47426 13.32966
## A0A087WPH7 14.91162 14.67173 14.67176 14.41860 13.62661
## A0A087WQ16 11.94430 11.75990 11.97628 11.99236 11.94584
## A0A087WQ32 11.60067 11.53075 11.63526 11.74671 11.45800
## A0A087WQH8 14.59580 14.47542 14.28707 14.55007 14.41281
```

Now that the data has been log2-transformed, let us replot it. Note that due to size constraints, there may be some overlap in the plot elements when this is knit into a PDF.

```
# Increase bottom margin to fit longer x-axis labels
par(mar = c(10, 4, 2, 1))  # bottom, left, top, right

# Draw boxplot
boxplot(
  prot[, meta$expression_column],
  col = meta$color,           # use color directly from metadata
  names = meta$group,         # x-axis labels = group names
  las = 2,
  xlab = "Samples",
  ylab = "log2(Intensity)",
  main = "Log2-transformed protein intensities"
)
```

```r
# Add legend (one entry per unique group-color combo)
legend_groups = unique(meta[, c("group", "color")])

legend(
  "topright",
  legend = legend_groups$group,
  fill = legend_groups$color,
  cex = 0.8 # scaling of the legend
)
```



It is now much easier to see what is happening in the data. After applying the log2 transformation, the intensities no longer span several orders of magnitude. Instead, the values are compressed into a more manageable range, roughly between 5 and 25 on the log2 scale. Notice how the medians nearly form a straight line across the boxes. This is a sign that the data has been normalized or otherwise processed, in this case using LFQ.

We should always watch for samples that behave abnormally in plots like these. For example, a sample with a much lower median than the others, even after normalization, might indicate a technical issue or sample failure. Color is also a helpful tool in data visualization. Here, the use of distinct colors makes it easy to separate the groups visually, even without referring to the legend.

Boxplot with ggplot2

While we will emphasize a lot of plotting using base R in this workshop, we will also take a look at the ggplot2 package. ggplot2 is part of the tidyverse suite of R packages, and it's a powerful system for creating more polished and customizable plots. However, it usually requires more code and additional data preparation, especially for grouped plots like boxplots across multiple samples. When possible, we will stick to simpler base R plots to keep things clear, and introduce ggplot2 when we want more flexibility or aesthetics, but for the sake of demonstration let us make a box plot with ggplot2.

First this requires some additional packages:

```
install.packages(c("tidyr", "dplyr", "ggplot2")).
```

```
# Load required libraries
library(tidyr)
library(dplyr)
```

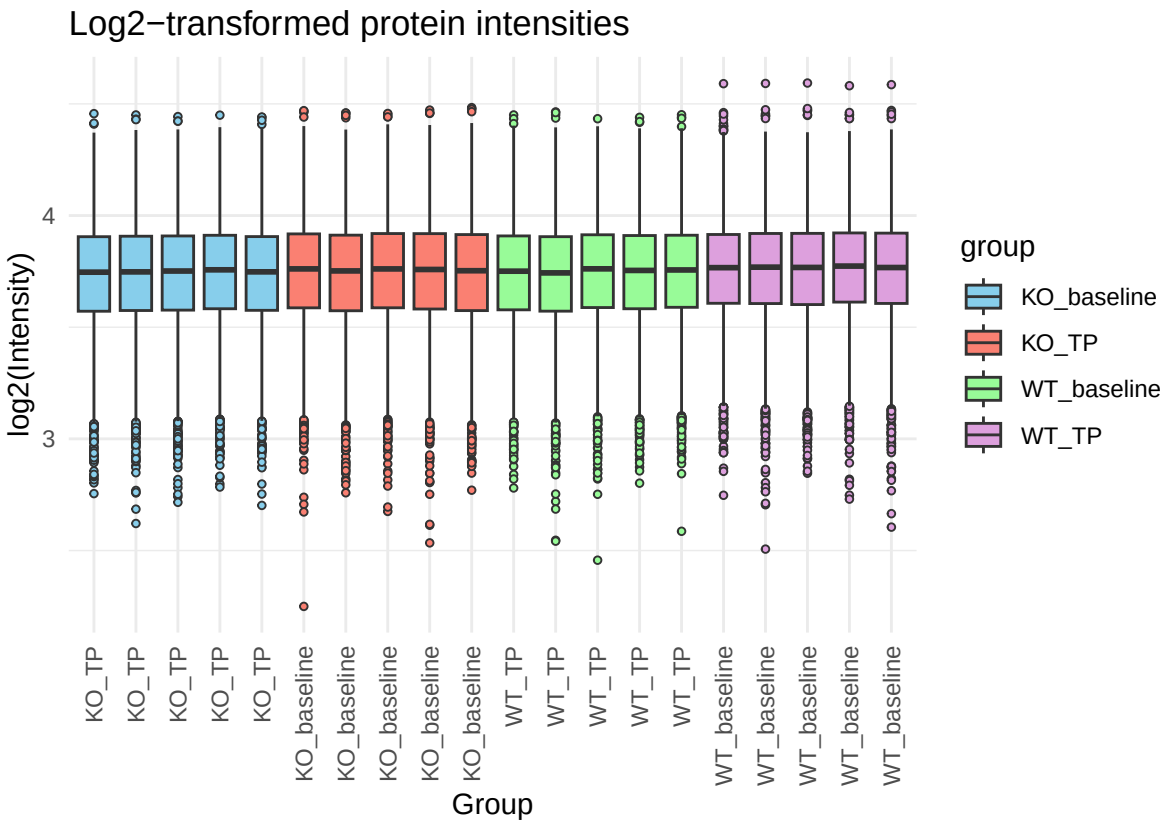
```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(ggplot2)

# Create a new column to use for x-axis labels: same number of boxes (1 per sample) but labeled by grou
meta$sample_label = meta$group

# Subset expression data
expression_data = prot[, meta$expression_column]
expression_data$protein = rownames(expression_data)

# Reshape to long format
library(tidyr)
long_df = pivot_longer(
  expression_data,
  cols = -protein,
  names_to = "expression_column",
  values_to = "intensity"
)

# Join sample_label and group info
library(dplyr)
long_df = left_join(long_df, meta[, c("expression_column", "group", "sample_label")], by = "expression_

# Remove 0 or missing intensity values before log2
long_df = long_df %>%
  filter(!is.na(intensity) & intensity > 0)

# Plot: one box per sample, labeled by group
library(ggplot2)
ggplot(long_df, aes(x = expression_column, y = log2(intensity), fill = group)) +
  geom_boxplot(outlier.shape = 21, outlier.size = 1) +
  scale_x_discrete(labels = meta$group) +  # override axis labels
  theme_minimal() +
  theme(
    axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1),
    plot.margin = margin(10, 10, 10, 20)
  ) +
  labs(
    x = "Group",
```

```
    y = "log2(Intensity)",
    title = "Log2-transformed protein intensities"
  ) +
  scale_fill_manual(values = c("skyblue", "salmon", "palegreen", "plum", "orange", "lightyellow")[1:leng
```



Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a very important dimension reduction technique. PCA works by identifying new axes, called principal components (PCs), that capture the most variability in the data. Instead of plotting hundreds or thousands of features, PCA reduces this to just a few new dimensions that explain the overall structure. We usually focus on the first two principal components, since they explain the largest amount of variation, but additional components can also be explored.

To run PCA in R, we need to address a common issue in mass spectrometry data: missing values. PCA cannot handle missing values, so we need to fill them in, or "impute" them. A simple approach is to replace missing values with a small number, such as the minimum value observed in the dataset. We can use is.na() to find missing values and replace them using brackets for assignment.

```
# Find the minimum non-missing intensity value
min_value = min(prot, na.rm = TRUE)

# Replace missing values with the minimum
prot_imputed = prot  # make a copy so we do not overwrite the original
prot_imputed[is.na(prot_imputed)] = min_value
```

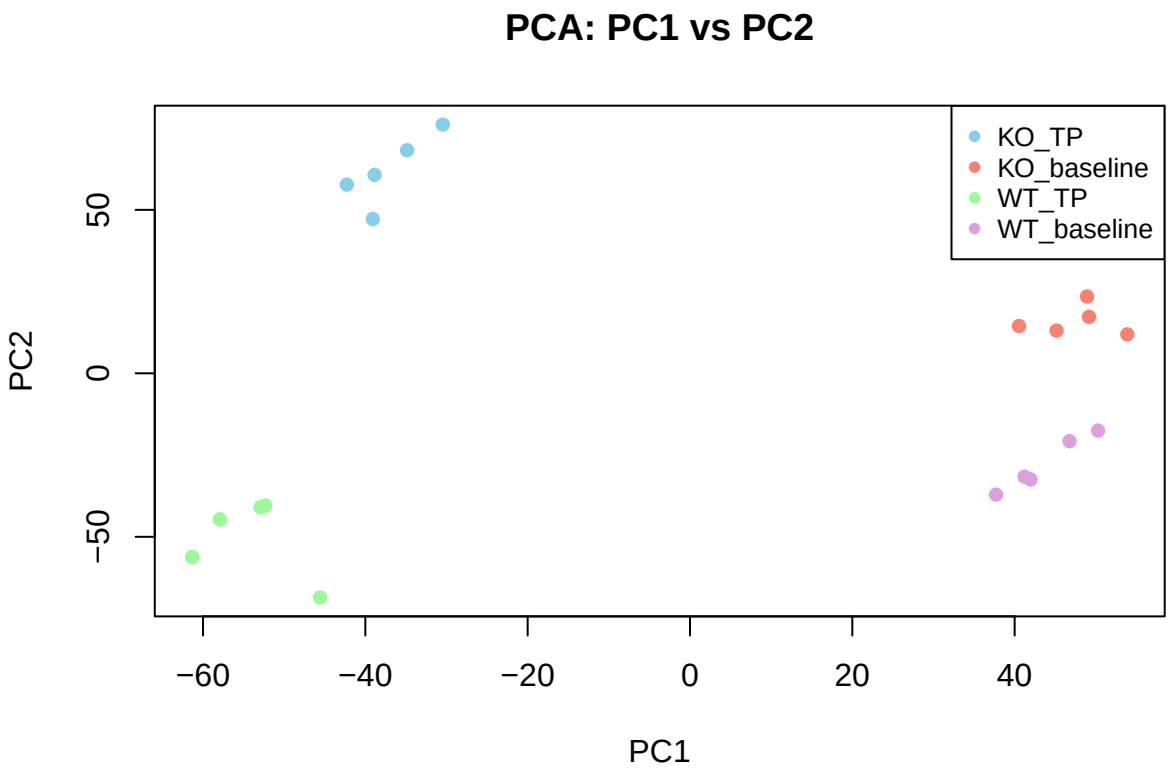
Once we have a complete matrix with no missing values, we can run PCA using the prcomp() function. This function expects samples to be rows and variables to be columns, so we need to transpose the data first. We also set center = TRUE and scale = TRUE to standardize the data. Scaling and centering (also known as

z-scoring) is important because it puts all features on the same scale. Without it, features with larger values or more variability, like a particularly abundant or noisy protein, can dominate the analysis. This would make the PCA reflect the behavior of just a few features rather than the overall data structure. Standardizing ensures that each feature contributes equally to the principal components.

```
# Transpose the data so that rows are samples and columns
# are features
prot_for_pca = t(prot_imputed)

# Run PCA
pca_result = prcomp(prot_for_pca, center = TRUE, scale = TRUE)
```

Once PCA is complete, we will visualize the first two principal components in a scatter plot, colored by group using the values from meta$color.

```
# Extract PCA scores
pca_scores = pca_result$x

# Plot the first two principal components using base R
plot(pca_scores[, 1], pca_scores[, 2], col = meta$color, pch = 16,
    xlab = "PC1", ylab = "PC2", main = "PCA: PC1 vs PC2")

# Add a legend to explain the colors
legend("topright", legend = unique(meta$group), col = unique(meta$color),
    pch = 16, cex = 0.8)
```



What a nice dataset! We can see distinct clusters spread across PC1 and PC2, which suggests clear biological differences between the groups. This kind of separation is a good sign because it means the variation captured by the top principal components is likely driven by meaningful biological signals rather than noise. When we perform differential expression analysis, we are likely to detect many significant differences between the groups.

PCA is also a helpful tool for spotting technical artifacts. If we saw unexpected clustering by something unrelated to biology, such as sample order, batch, or run date, that could indicate the presence of batch effects or technical bias. Likewise, if a sample falls far from all the others, it might be a true outlier due to poor quality or a possible sample swap. In this case, we do not see any signs of those issues. The data looks clean, and the grouping is consistent with what we expect based on the experimental design.

Heatmap

Heatmaps are another useful way to visualize our data while performing hierarchical clustering. A heatmap shows the intensity of many features, such as proteins, across all samples, using color to represent relative abundance. Clustering is applied to both rows (features) and columns (samples) to group together those with similar expression patterns.

However, clustering can break or behave unpredictably if too many values are missing. Distance calculations between samples or features can fail when the data contains large amounts of NA, especially if entire rows or columns are mostly missing. To avoid this problem, we first remove proteins with too much missing data. In this example, we filter out proteins that are missing in more than 50% of samples.

```
row_na_threshold = 0.5
keep_rows = rowMeans(is.na(prot)) < row_na_threshold
prot = prot[keep_rows, ]
```

We typically do not include all proteins in the heatmap. In any omics experiment, many features will not vary much across samples. Including these adds noise and makes patterns harder to see. Instead, we focus on a subset of the most variable features. One common approach is to use standard deviation to identify the most variable proteins. Here, we will select the top 1000 proteins based on their standard deviation.

To do this, we use the apply() function, which applies a function to either the rows or columns of a data frame. By setting MARGIN = 1, we tell R to apply the sd() (standard deviation) function to each row (i.e., each protein). We also use na.rm = TRUE so that missing values (NAs) are ignored. Without it, even a single missing value would cause the result to be NA.

```
# Select top 1000 proteins by standard deviation
protein_sds = apply(prot, MARGIN = 1, sd, na.rm = TRUE)
top_proteins = names(sort(protein_sds, decreasing = TRUE))[1:1000]
top_prot = prot[top_proteins, ]
```

Before plotting, we need to center and scale the data by performing a z-score transformation on each row. This step is important. Just like in PCA, large differences in raw intensity values can overwhelm the clustering and lead to misleading patterns. Without standardization, proteins with higher average abundance will cluster together not because they behave similarly across samples, but simply because their overall values are larger.

```
# Z-score normalization (row-wise)
z_top_prot = t(scale(t(top_prot)))
```

After scaling, the heatmap will better reflect biological patterns of change rather than differences in measurement scale. Clusters of samples and features may emerge that reflect group structure or co-regulated proteins. The heatmap can also help detect outliers or technical artifacts.

To create our heatmap, we will use the ComplexHeatmap package. This package is part of Bioconductor (https://bioconductor.org), which is a specialized repository for bioinformatics tools in R. Unlike CRAN, which hosts general-purpose R packages, Bioconductor focuses on packages for analyzing genomic and other biological data. You install Bioconductor packages using the BiocManager package.

You can do so with the following code. This block checks whether BiocManager is installed, installs it if needed, and then uses it to install ComplexHeatmap (https://www.bioconductor.org/packages/release/bioc/html/ComplexHeatmap.html). You can copy and paste this into your R console:

```r
if (!require("BiocManager", quietly = TRUE)) install.packages("BiocManager")
BiocManager::install("ComplexHeatmap")
```

Once the package is installed, we can prepare our data. We will select the top 1,000 most variable proteins based on standard deviation, apply z-score scaling to center and standardize each row (protein), and color the columns (samples) by group/genotype/condition from meta.

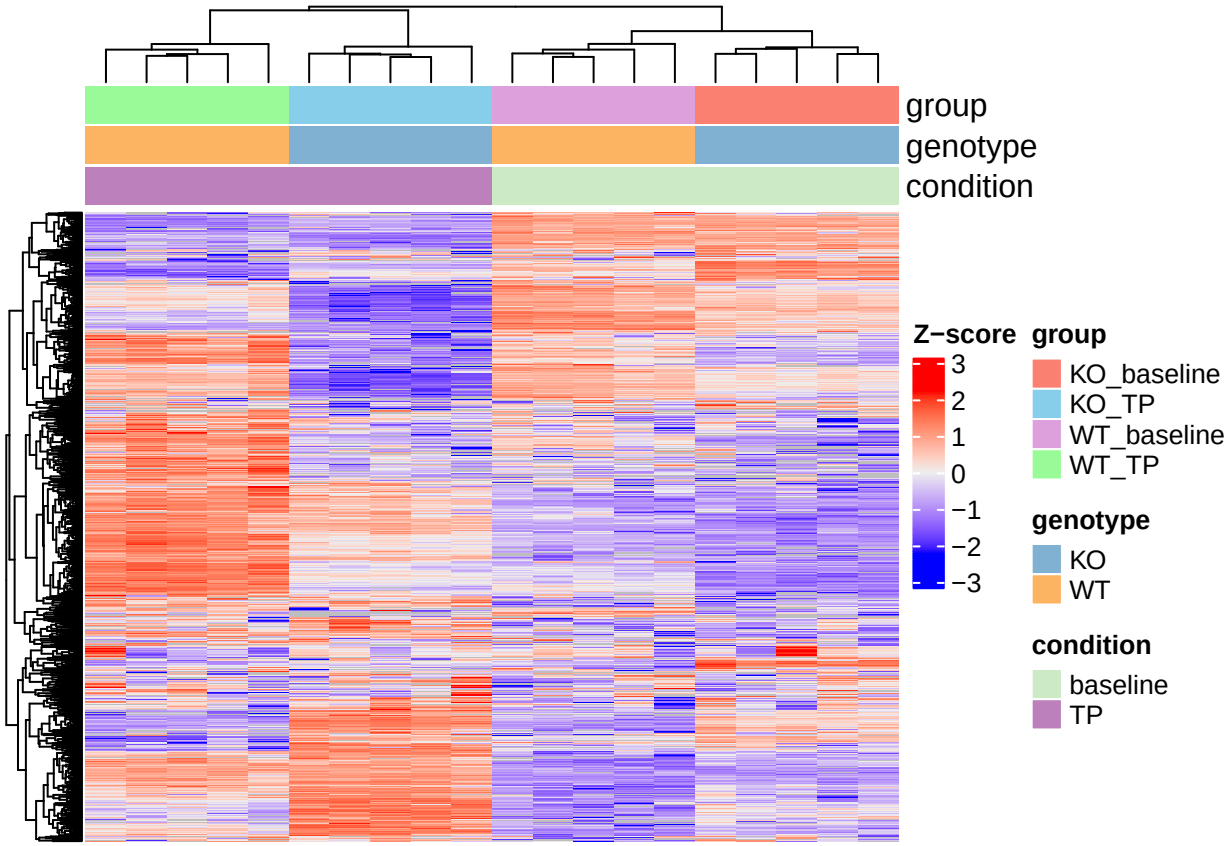
```r
# Load ComplexHeatmap
library(ComplexHeatmap)
```

```
## Loading required package: grid
```

```
## ========================================
## ComplexHeatmap version 2.24.0
## Bioconductor page: http://bioconductor.org/packages/ComplexHeatmap/
## Github page: https://github.com/jokergoo/ComplexHeatmap
## Documentation: http://jokergoo.github.io/ComplexHeatmap-reference
##
## If you use it in published research, please cite either one:
## - Gu, Z. Complex Heatmap Visualization. iMeta 2022.
## - Gu, Z. Complex heatmaps reveal patterns and correlations in multidimensional
##     genomic data. Bioinformatics 2016.
##
##
## The new InteractiveComplexHeatmap package can directly export static
## complex heatmaps into an interactive Shiny app with zero effort. Have a try!
##
## This message can be suppressed by:
##   suppressPackageStartupMessages(library(ComplexHeatmap))
## ========================================
```

```r
# Group colors (already defined in meta)
group_colors_named = meta$color
names(group_colors_named) = meta$group

# Define pastel but distinct hex colors
geno_colors = c(
  "WT" = "#FDB462",   # soft orange
  "KO" = "#80B1D3"    # pastel blue (not skyblue)
)

cond_colors = c(
  "baseline" = "#CCEBC5",  # light mint green
  "TP"       = "#BC80BD"   # muted violet
)

# Create column annotations
column_annot = HeatmapAnnotation(
  group = meta$group,
  genotype = meta$genotype,
  condition = meta$condition,
  col = list(
    group = group_colors_named,
    genotype = geno_colors,
    condition = cond_colors
```

```
  )
)

# Draw heatmap
Heatmap(
  z_top_prot,
  name = "Z-score",
  top_annotation = column_annot,
  show_column_names = FALSE,
  show_row_names = FALSE,
  cluster_rows = TRUE,
  cluster_columns = TRUE
)
```



Questions

1. Create a new data frame with just the top 25 most variable proteins (by standard deviation). Try drawing a heatmap using just this smaller set.
2. Try setting cluster_rows = FALSE or cluster_columns = FALSE in your heatmap to see what changes.

Annotate data

In proteomics datasets, protein identifiers are often provided as Uniprot accession numbers (e.g., P12345). However, for downstream analysis and easier interpretation, we often want to map these to more familiar gene symbols (e.g., TP53). To do this, we can make use if Uniprot's batch ID mapping tool https:

//www.uniprot.org/id-mapping . Other options can be fully done in R but are a bit more complicated and are beyond the scope of the current workshop.

Load in the ID mapping data and take a look:

```
id_map = read.delim("data/idmapping_2025_07_17.tsv", header = TRUE)
head(id_map)
```

```
##         From      Entry   Reviewed       Entry.Name
## 1 A0A087WNL7 A0A087WNL7 unreviewed A0A087WNL7_MOUSE
## 2 A0A087WPH7 A0A087WPH7 unreviewed A0A087WPH7_MOUSE
## 3 A0A087WQ16 A0A087WQ16 unreviewed A0A087WQ16_MOUSE
## 4 A0A087WQ32 A0A087WQ32 unreviewed A0A087WQ32_MOUSE
## 5 A0A087WQH8 A0A087WQH8            A0A087WQH8_MOUSE
## 6 A0A087WQN7 A0A087WQN7 unreviewed A0A087WQN7_MOUSE
##                                          Protein.names Gene.Names
## 1                Snf2-related CREBBP activator protein      Srcap
## 2 Proteasome (prosome, macropain) 26S subunit, ATPase 3      Psmc3
## 3                           DNA mismatch repair protein       Msh3
## 4 GPI ethanolamine phosphate transferase 1 (EC 2.-.-.-)       Pign
## 5                                              deleted
## 6                5'-3' exoribonuclease 1 (EC 3.1.13.-)       Xrn1
##               Organism Length
## 1 Mus musculus (Mouse)   3210
## 2 Mus musculus (Mouse)    305
## 3 Mus musculus (Mouse)   1095
## 4 Mus musculus (Mouse)    826
## 5                           NA
## 6 Mus musculus (Mouse)   1719
```

We need to match the gene names to the UniProt identifiers using the match() function. To understand how it works, let's look at a simple example.

```
letters = c("a", "b", "c", "d")
lookup = c("b", "d", "a", "c")
match(letters, lookup)
```

```
## [1] 3 1 4 2
```

This returns: 3 1 4 2 . It tells us that "a" is in position 3 of the lookup vector, "b" is in position 1, "c" is in position 4, and "d" is in position 2.

Now let's apply this idea to our protein data. The id_map data frame should have a column called Entry (UniProt ID) and another called Gene.Names. In practice, some proteins map to multiple gene names or vice versa. This can be a major challenge when working with proteomics or other omics datasets. To simplify things for now, we will remove duplicated gene names using the duplicated() function so that each gene name appears only once. This avoids ambiguity when we map IDs to gene names.

```
# Which ones are NOT duplicated?
id_map_nodup = id_map[!duplicated(id_map$Gene.Names), ]

# Find the positions in id_map_nodup$Entry that match the
# row names of prot
match_positions = match(row.names(prot), id_map_nodup$Entry)

# Use those positions to get the gene names
matched_genes = id_map_nodup$Gene.Names[match_positions]
```

```
# Identify which gene names are missing (either NA or blank
# '')
missing = is.na(matched_genes) | matched_genes == ""

# Now replace those missing gene names with the original
# UniProt IDs
matched_genes[missing] = row.names(prot)[missing]

# Now we update the row names of the prot data frame with
# the gene names make.names() ensures all row names are
# valid R names and unique
row.names(prot) = make.names(matched_genes, unique = TRUE)

head(row.names(prot))
```

```
## [1] "Srcap"      "Psmc3"      "Msh3"       "Pign"       "A0A087WQH8"
## [6] "Xrn1"
```

Differential expression

Hypothesis testing

Now that we are happy with how the data looks and our proteins are labeled with more interpretable gene symbols, we can ask which proteins are differentially expressed between experimental conditions. A common way to test for differences in expression is to use a two-sample t-test. This test compares the means of two groups and asks whether the observed difference could have occurred by chance.

The t-test returns a p-value, which tells us the probability of observing a difference as large (or larger) than what we saw if there were actually no real difference. A small p-value (typically less than 0.05) suggests that the observed difference is unlikely to be due to random chance.

However, statistical significance does not always mean biological significance. For example, a diabetes drug might reduce blood sugar by only 1 percent, say from 200 to 198. Such a change might be statistically significant in a large study but would not be meaningful for the patient. That is why we often pair p-values with effect size. In this case, we use log2 fold-change. A larger absolute fold-change (for example, greater than 2-fold or log2 fold-change greater than 1) is more likely to be biologically important.

We will compare the WT_baseline and WT_TP groups and look for proteins that are both statistically significant and show a meaningful change in expression. There are several steps, so first we will again use our trick with the meta data to identify the sample columns for each group. It is a good idea to add the results to additional columns at the end of the dataset because it is easy in R and then we have both the abundance and the analysis results in the same place.

```
WT_baseline_samples = meta$expression_column[meta$group == "WT_baseline"]
WT_TP_samples = meta$expression_column[meta$group == "WT_TP"]
```

We will again use apply, this time to calculate means instead of standard deviation:

```
prot$WT_baseline_mean = apply(prot[, WT_baseline_samples], 1,
    function(x) mean(x, na.rm = TRUE))
prot$WT_TP_mean = apply(prot[, WT_TP_samples], 1, function(x) mean(x,
    na.rm = TRUE))
```

We will next calculate the log2 fold change (or the "log2 ratios") between samples, but first an important reminder from college algebra: log2(a/b) = log2(a) - log2(b). Since our data is already log2 transformed, we can simply subtract the group means to compute the log2 fold-change.

```
prot$WT_TP_vs_WT_baseline_log2fc = prot$WT_TP_mean - prot$WT_baseline_mean
```

Next, we perform t-tests for every protein to compare expression between the WT_baseline and WT_TP groups.

However, some proteins may have missing values (NA) in one or both groups. If we do not handle this properly, the t.test() function will fail when it encounters missing data and will stop the code with an error.

To avoid this, we wrap the t.test() call in a tryCatch() block. This tells R to try running the code, but if it fails (for example, due to too many NAs), it should return NA instead of crashing. This allows the loop to continue for the rest of the proteins.

```
# Perform t-tests row-wise
prot$pval = apply(prot, 1, function(x) {
    tryCatch({
        t.test(as.numeric(x[WT_baseline_samples]), as.numeric(x[WT_TP_samples]))$p.value
    }, error = function(e) NA)
})
```

After calculating a p-value for each protein, we need to account for the fact that we are performing thousands of statistical tests at once. Without correction, we would expect many false positives just by chance. To address this, we adjust the p-values using the Benjamini-Hochberg ("BH") method to control the false discovery rate (FDR).

```
# Adjust p-values using the Benjamini-Hochberg (FDR) method
prot$padj = p.adjust(prot$pval, method = "BH")
```

Now let us flag what entries pass our criteria for differential expression.

```
prot$differentially_expressed = abs(prot$WT_TP_vs_WT_baseline_log2fc) >=
    1 & prot$padj <= 0.05
```

Volcano plot

A volcano plot is a useful visualization tool for differential expression analysis. It helps us quickly identify proteins (or genes) that are both statistically significant and strongly changed in abundance. The x-axis of the volcano plot shows the log2 fold change, which represents how much a protein's expression increases or decreases. The y-axis shows –log10 of the adjusted p-value (padj), so that more statistically significant proteins appear higher up on the plot.

The reason it's called a "volcano" is because when most proteins have small fold changes and only a few have large changes, the points tend to form a shape that resembles a volcano. This shape can help us diagnose whether our differential expression comparison is working properly. If everything is clustered around zero or there are no significant values, it may indicate problems like low statistical power or technical noise.

To make a volcano plot, we need several pieces of information for each protein: * the log2 fold change * the adjusted p-value (padj) * the –log10(padj) for plotting on the y-axis * the protein name or label * the color (based on significance and direction of change)

Rather than calculating and tracking each of these separately, it's cleaner and easier to put everything into a single data frame. This allows us to work with the data in a structured way, label the most extreme values, and use that data frame directly in our plot.

```
volcano_df = data.frame(log2fc = prot$WT_TP_vs_WT_baseline_log2fc,
    padj = prot$padj, name = row.names(prot))
```

Calculate -log10(padj)

```
volcano_df$neglog10padj = -log10(volcano_df$padj)
```

Assign colors by significance and direction

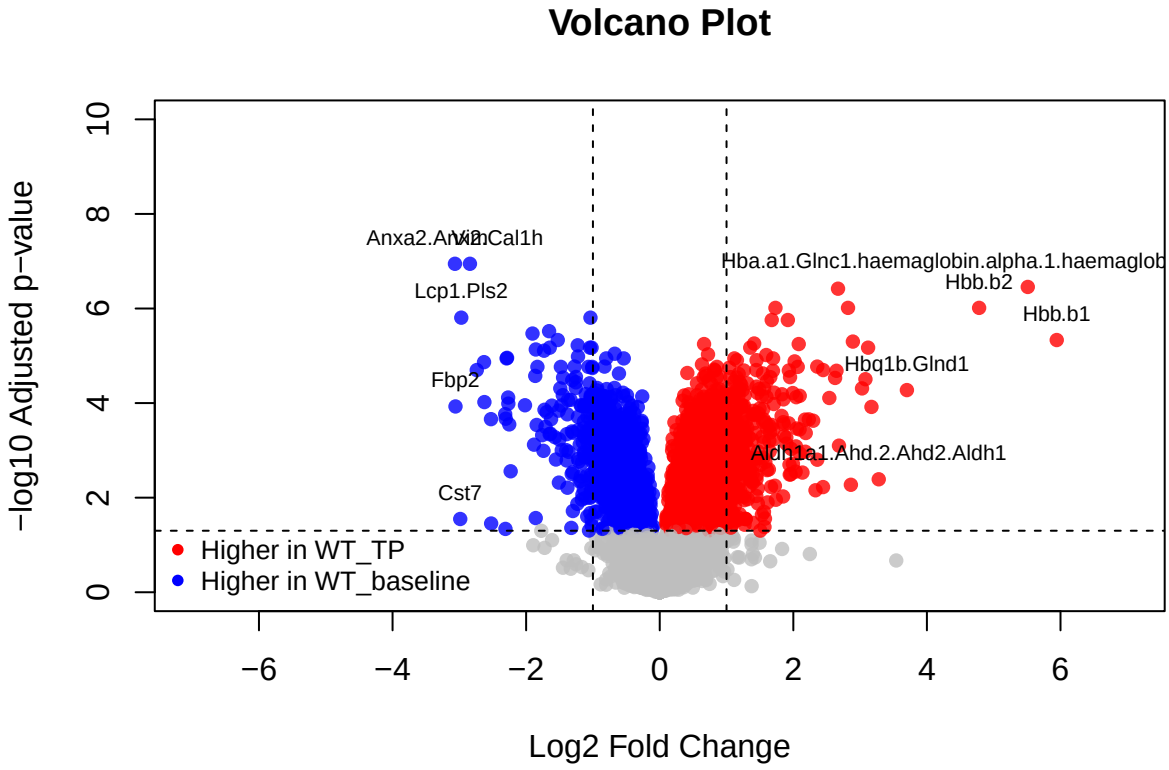
```
volcano_df$color = "gray"
volcano_df$color[volcano_df$padj < 0.05 & volcano_df$log2fc >
    0] = "red"
volcano_df$color[volcano_df$padj < 0.05 & volcano_df$log2fc <
    0] = "blue"
```

Get the gene symbols for the most differentially expressed proteins so we can display them in the plot

```
# Filter for significant proteins only
sig_df = volcano_df[volcano_df$padj < 0.05, ]

# Find top 5 significant upregulated (most positive log2fc)
top5_up_idx = order(sig_df$log2fc, decreasing = TRUE)[1:5]

# Find top 5 significant downregulated (most negative
# log2fc)
top5_down_idx = order(sig_df$log2fc)[1:5]

# Get the original row numbers for labeling; this will be
# used for text() momentarily
label_idx = match(sig_df$name[c(top5_up_idx, top5_down_idx)],
    volcano_df$name)
```

Create the volcano plot using base R

```
# Plot the volcano
plot(volcano_df$log2fc, volcano_df$neglog10padj, col = adjustcolor(volcano_df$color,
    alpha.f = 0.8), pch = 16, xlab = "Log2 Fold Change", ylab = "-log10 Adjusted p-value",
    xlim = c(-7, 7), ylim = c(0, 10), main = "Volcano Plot")

# Add dashed lines
abline(h = -log10(0.05), lty = 2)
abline(v = c(-1, 1), lty = 2)

# Label the selected proteins
text(volcano_df$log2fc[label_idx], volcano_df$neglog10padj[label_idx],
    labels = volcano_df$name[label_idx], pos = 3, cex = 0.7)

# Add a legend

# Add a legend for direction of change
legend("bottomleft", legend = c("Higher in WT_TP", "Higher in WT_baseline"),
    col = c("red", "blue"), pch = 16, bty = "n", cex = 0.8)
```



Volcano plot with ggplot2

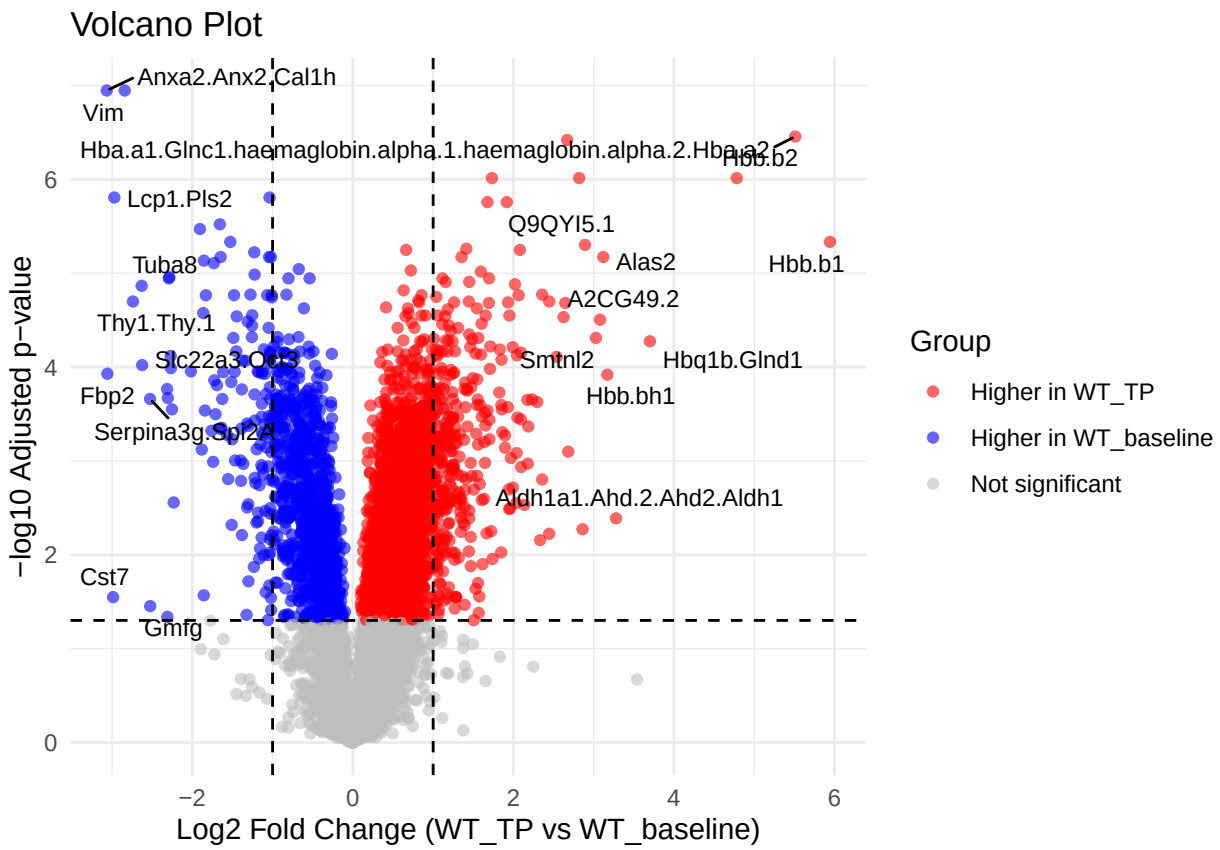
To explore the same data using ggplot2, we will create another volcano plot so we can see some of the aesthetic differences. First we need the ggrepel library to make it so our labels will not overlap:

```
install.packages("ggrepel")
```

Then we make the volcano:

```
# Load libraries (ggplot2 should already be loaded from
# before)
library(ggrepel)

# Create volcano_df_gg
volcano_df_gg = data.frame(log2fc = prot$WT_TP_vs_WT_baseline_log2fc,
    padj = prot$padj, name = row.names(prot))

# Calculate -log10 adjusted p-value
volcano_df_gg$neglog10padj = -log10(volcano_df_gg$padj)

# Assign group labels for coloring
volcano_df_gg$group = "Not significant"
volcano_df_gg$group[volcano_df_gg$padj < 0.05 & volcano_df_gg$log2fc >
    0] = "Higher in WT_TP"
volcano_df_gg$group[volcano_df_gg$padj < 0.05 & volcano_df_gg$log2fc <
    0] = "Higher in WT_baseline"

# Set factor levels for consistent legend order
volcano_df_gg$group = factor(volcano_df_gg$group, levels = c("Higher in WT_TP",
    "Higher in WT_baseline", "Not significant"))
```

```
# Filter for complete, plottable data
volcano_df_gg_clean = subset(volcano_df_gg, is.finite(log2fc) &
    is.finite(neglog10padj))

# Identify top 10 significant up/down by fold change
sig_df = subset(volcano_df_gg_clean, padj < 0.05)
top10_up = sig_df[order(sig_df$log2fc, decreasing = TRUE)[1:10],
    ]
top10_down = sig_df[order(sig_df$log2fc)[1:10], ]
label_df = rbind(top10_up, top10_down)

# Make the volcano plot
ggplot(volcano_df_gg_clean, aes(x = log2fc, y = neglog10padj,
    color = group)) + geom_point(alpha = 0.6) + geom_vline(xintercept = c(-1,
    1), linetype = "dashed") + geom_hline(yintercept = -log10(0.05),
    linetype = "dashed") + geom_text_repel(data = label_df, aes(label = name),
    color = "black", size = 3, max.overlaps = Inf, show.legend = FALSE) +
    scale_color_manual(values = c(`Higher in WT_TP` = "red",
        `Higher in WT_baseline` = "blue", `Not significant` = "gray")) +
    labs(title = "Volcano Plot", x = "Log2 Fold Change (WT_TP vs WT_baseline)",
        y = "-log10 Adjusted p-value", color = "Group") + theme_minimal()
```



Pathway enrichment analysis

With so many proteins differentially expressed, it is difficult to make sense of them one by one. Pathway enrichment analysis helps by grouping proteins into biological processes or signaling pathways and identifying

whether any of these groups are statistically overrepresented in our results. This gives us a higher-level view of the biological systems that may be active or disrupted, making it easier to interpret complex omics data. Further reading: https://pubmed.ncbi.nlm.nih.gov/22383865/.

Before we can run enrichment, we need to clean up our identifiers. In proteomics data, protein groups often represent multiple possible gene products. For example, "Actb.Fos.Tpm4" could refer to any of those three gene symbols. Unfortunately, enrichment tools like enrichR expect single gene symbols, and they cannot handle names with multiple entries or special characters.

To address this, we will use a quick string-cleaning trick: keep only the first gene symbol and discard anything after the first ".". This is a coarse solution, but it works well for exploratory analysis.

We will use the stringr package to perform this cleanup. stringr is part of the tidyverse collection of R packages and provides user-friendly functions for working with text, including tools for pattern matching using regular expressions. We will use a regular expression (also called regex) to define the pattern we want to remove, in combination with a stringr function to apply it.

```
install.packages("stringr")
```

Regular expressions are a compact way to describe patterns in text. In R, they are frequently used to extract, match, or clean strings, especially during data preprocessing steps like this one.

Here's how the pattern works in our case:

```
library(stringr)
# \\. means 'a literal dot' (the dot is a special character
# in regular expressions, so we have to tell R to ignore it
# with a double backslash) . means 'any single character'
# (not just a literal dot).  * means 'zero or more
# repetitions' of the thing before it.  Together, \\..*
# means: 'the first dot and everything after it'

example_names = c("Actb.Fos.Tpm4", "Myc", "Cd44.Tubb3")
str_replace(example_names, "\\..*", "")
```

```
## [1] "Actb" "Myc"  "Cd44"
```

Now that we understand the pattern, we can apply it to our actual protein names and run enrichment analysis.

Clean protein identifiers and run enrichment

```
# Load libraries
library(enrichR)
```

```
## Welcome to enrichR
## Checking connections ...
```

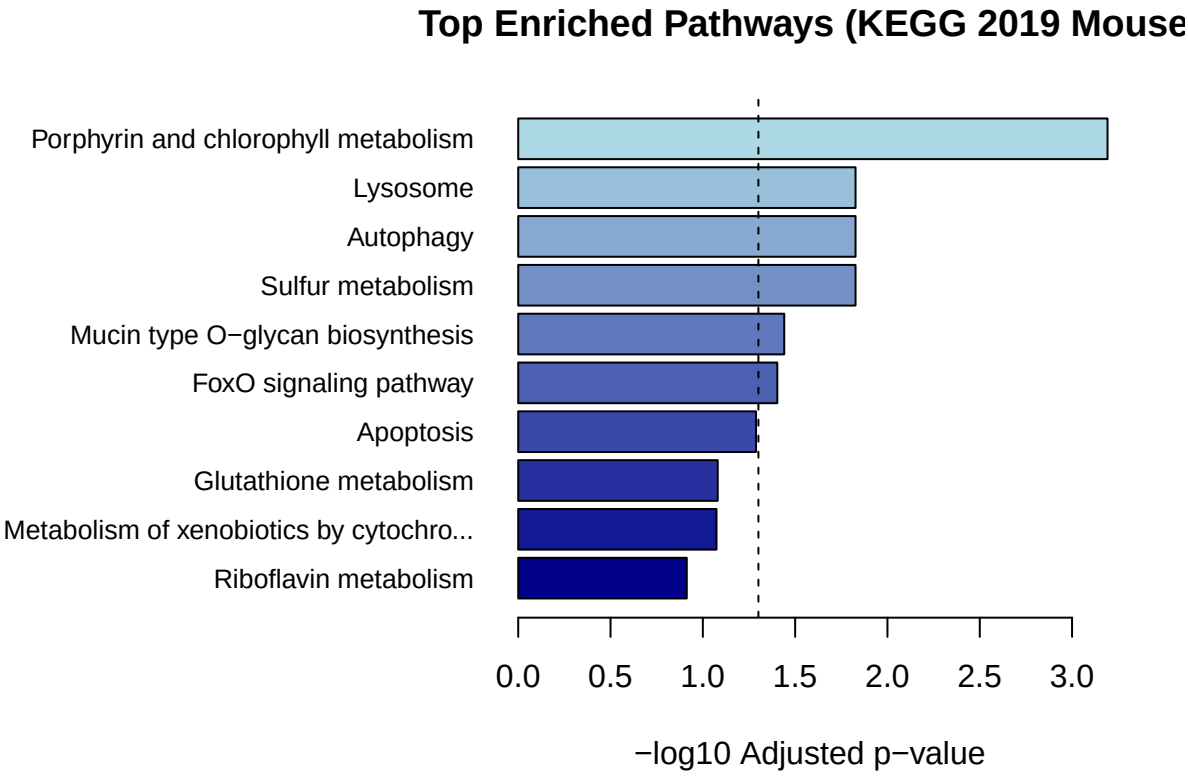
```
## Enrichr ... Connection is Live!
## FlyEnrichr ... Connection is Live!
## WormEnrichr ... Connection is Live!
## YeastEnrichr ... Connection is Live!
## FishEnrichr ... Connection is Live!
## OxEnrichr ... Connection is Live!
```

```
library(stringr)

# Use KEGG 2019 Mouse database
dbs = c("KEGG_2019_Mouse")
```

```r
# Extract row names of significantly upregulated proteins
protein_names = row.names(prot)[prot$differentially_expressed &
    prot$WT_TP_vs_WT_baseline_log2fc > 0]

# Use regex to strip anything after the first '.'
gene_list_clean = str_replace(protein_names, "\\..*", "")

# Run enrichment
enrichment_results = enrichr(gene_list_clean, dbs)
```

```
## Uploading data to Enrichr... Done.
##   Querying KEGG_2019_Mouse... Done.
## Parsing results... Done.
```

```r
# Extract results
kegg_results = enrichment_results[["KEGG_2019_Mouse"]]
# Visualize results using base R Top N terms and
# significance scores
top_n = 10
top_kegg = head(kegg_results[order(kegg_results$Adjusted.P.value),
    ], top_n)
top_kegg$neglog10padj = -log10(top_kegg$Adjusted.P.value)

# Shorten long pathway names
short_labels = ifelse(nchar(top_kegg$Term) > 40, paste0(substr(top_kegg$Term,
    1, 37), "..."), top_kegg$Term)

# Color gradient
bar_colors = colorRampPalette(c("lightblue", "darkblue"))(top_n)

# Adjust plot margins: bottom, left, top, right
par(mar = c(5, 15, 4, 2))  # left margin increased from default (usually 4) to 15

# Horizontal barplot
barplot(rev(top_kegg$neglog10padj), names.arg = rev(short_labels),
    horiz = TRUE, col = rev(bar_colors), las = 1, xlab = "-log10 Adjusted p-value",
    main = "Top Enriched Pathways (KEGG 2019 Mouse)", cex.names = 0.8)

# Add vertical line at padj = 0.05 (approx -log10(0.05) =
# 1.3)
abline(v = -log10(0.05), lty = 2, col = "black")
```



Save the results

Save the updated prot with gene symbols and differential expression results:

```
write.table(prot, "work/prot_gene_symbols_differential_exp_2025-07-17.txt",
    sep = "\t", quote = F, row.names = T, col.names = NA)
```

Similarly for the pathway enrichment:

```
write.table(kegg_results, "work/kegg_higher_WT_TP_2025-07-17.txt",
    sep = "\t", quote = F, row.names = F)
```

Questions

1. How many proteins pass both the adjusted p-value and fold-change cutoffs?
2. What happens if you change the log2 fold-change cutoff from 1 to 0.5? How does this affect the number of differentially expressed proteins?
3. Try making a histogram of all log2 fold-changes using hist(). Does the distribution look symmetric?
4. What is the 42nd most differentially expressed protein (by adjusted p-value) in WT_TP?
5. Perform pathway enrichment analysis on the significantly downregulated proteins, and make a barplot to visualize the results.
6. Optional homework! Take some of your own data, use this markdown file as a template, and see how far you can get with processing and visualizing it. Start by changing meta to match your samples and experimental data.

Convert R Markdown to PDF

To convert an R Markdown file into a PDF, R requires an additional program called LaTeX. One simple and lightweight option is TinyTeX, which installs just the components needed for R Markdown and works behind the scenes to generate PDF output.

Installing TinyTeX is optional and can be done later if you are interested in generating PDFs. The code chunk below shows how to install it, but it is tagged with eval = FALSE so it will not run automatically. The installation is several hundred megabytes and may take a little time. In some cases, installation issues can occur that require additional troubleshooting, so we recommend doing this on your own time rather than during the workshop.

If you decide to install TinyTeX, simply remove, eval = FALSE from the chunk header and run the code. After installation, you should see a "Knit" button at the top of your RStudio window. You can use this button to convert your R Markdown file to PDF.

```
install.packages(c("rmarkdown", "knitr", "tinytex"))
tinytex::install_tinytex()
```

Further reading

If you would like to continue building your skills in R and data analysis, a great place to start is Roger D. Peng's open-access (free) book *R Programming for Data Science*: https://bookdown.org/rdpeng/rprogdatascience/. It introduces many foundational ideas in R programming and data science in a practical and approachable way.

For topics more specific to genomics and bioinformatics workflows, *Computational Genomics with R* by Altuna Akalin is another excellent resource: https://compgenomr.github.io/book/. It covers a range of common bioinformatics tasks, including working with sequencing data, visualization, and downstream analysis using R and Bioconductor.

Looking for inspiration on how to visualize your data in R? Explore the R Graph Gallery: https://r-graph-gallery.com

Want to see some of these concepts applied in cutting-edge proteogenomics research? Take a look at recent CPTAC papers, such as this one: https://pubmed.ncbi.nlm.nih.gov/38359819/

Although this workshop focuses on proteomics, many of the same principles and tools apply across bioinformatics domains. Bioinformatics is, at its core, biological data science. Most well-written data science resources are directly relevant and can help you continue to grow your skills.

How to get help

Many common questions and answers: https://stackoverflow.com (top Google search hits for error messages and problems with R usually have some links to here).

Amazing for debugging code and troubleshooting error messages https://chatgpt.com.

Need to ask someone for help? Make a SSCCE https://sscce.org.