Session 4: Data Visualization

I. Ali and M. Wells

2023-02-23

# Introduction to Plotting

One of the most powerful features of R is its functionality with data visualization and plotting. Various developers in the R community have created robust tools for data visualization and plotting. Some plotting libraries are immensely extensive, allowing you to create nearly any kind of data driven graphic you would like and more.

The R developers community has created a variety of packages that are useful for plotting. “Base” R has a highly customizable plotting function and is used on occasion (see plot()). Searching forums is a great way to learn about the ways you can customize features needed for your own data visualization approaches. See: [r-bloggers.com/some-basics-for-base-graphics](https://www.r-bloggers.com/some-basics-for-base-graphics/) More recently, other plotting libraries have been developed and implemented for various applications. This includes plotly, leaflet, pheatmap, EnhancedVolcano and more.

By the end of this session you should be able to

* Recognize the 3 main components of the grammar of graphics.
* Create three or more different kinds of plots.
* Recognize modifications to plot themes and colors
* Save plots you create to export as figures.

The most commonly used visualization library in R is ggplot2 package which has immense customizability and the most extensive documentation and instructional resources of the plotting libraries. ggplot2 is based on what is called the ‘grammar of graphics’. This is the idea that you can build nearly every graph from the same components: (1) a data set, (2) a coordinate system and (3) ‘geoms’ which are visual marks that represent data points. ggplot2 is able to create a diverse set of plots using these three characteristics. Here we will explore how to construct a few example plots using ggplot2. For a more exhaustive list of ggplot2 functions please review the [ggplot2 cheat sheet](https://hbctraining.github.io/Intro-to-R-flipped/cheatsheets/data-visualization-2.1.pdf).

It is worth fully exploring ggplot2 documentation. There are many features that you can customize for each geometry making ggplot2 one of the most comprehensive packages for data visualization in R.

The art of data visualization requires both an understanding of the data that you are working with and also an awareness of the important features of the data that you want to highlight. Experimenting with different visualization strategies can help you better pin down the most informative ways of displaying your data. When working with your own data it can be important to look at mean or median values, the variance and spread, correlations and clusters.

Through our exercises in this session, we aim to highlight some of the more common plots you might come across in your research. We will cover how to make these plots using ggplot, or other relevant plotting libraries.

## Loading Packages

Don’t forget to always install or load packages at the beginning of your document.

Remember: There are multiple ways to create code chunks.

1. Look at the top right corner of your R markdown window. There is a green button with the letter C and a green plus symbol. This will insert a code chunk pre-formatted for the programming language that is selected.
2. Use the “Code” menu button and select “Insert Chunk”. This will default to creating a code chunk using R.
3. Use the shortcut for pc (ctrl+alt+i) or mac (command+option+i). This will insert a code chunk using R.

In this lesson it may be useful to create new code chunks when you are creating new plots

## Plotting One Variable

We are going to start with plotting individual variables and gradually increase the complexity and the number of layers of the plot. We created a fictional “dummy” data set containing read counts per million of 15 human genes measured at 8 time points. We are going to use these read counts and genes to practice some plotting basics.

ggplot2 requires data to be “tidy” which is also called “long” format. We have pre-formatted the data to be in long form. Use View() to check to see what the data looks like.

# Using the read.csv() command import the GeneExpressionData.csv file  
SampleGenes <- read.csv("../Datasets/GeneExpressionData.csv", header = TRUE, sep = ",")

All ggplot commands follow a similar syntax. In the first line, you call the ggplot() command. In this command the first argument is the data frame, and the second argument maps the **aesthetics** using the aes() command. These are the visual attributes of the geometric object created in your plot. It could map to x-position, y-position, color, shape, and size. Each attribute within the aesthetics can be mapped to a variable in our data set, increasing the density of data we can include in our plots. Aesthetics defined in the first line are passed down to all geometries you add to your plot.

After including the first two components, you can add any number of geometries to your ggplot() command which we will get to practice in a number of ways.

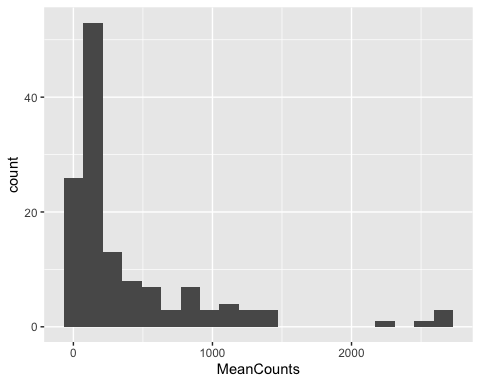
### Histograms

**Histograms** are particularly useful in statistics as they reveal characteristics about sample distribution. Statistical tests often assume certain things about the sample distribution. Therefore reviewing the distribution of your sample can help you select the correct statistical test to use.

Histograms measure the number of values that fall into pre-defined bins over a sample distribution. Below we will use the geom\_histogram() geometry to look at the distribution of counts per-million for all the genes in our data set. We can set a fixed number of bins across a data set, or we can set a fixed bin width to visualize the data.

geom\_histogram() will plot only one variable distributed across the x or the y axis.

# Use geom\_histogram to create a histogram to show the gene expression counts  
# for all of the data  
  
# Notice that you can add layers and features to ggplots with a + symbol  
  
ggplot(SampleGenes, aes(x = MeanCounts)) +   
 geom\_histogram(bins = 20)

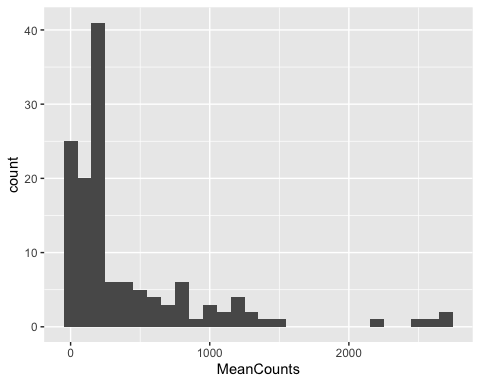


Notice that when you write a ggplot command, the first argument is the long form data set that it will use to create the plot. The second argument includes the aesthetics. We use x = MeanCounts to view the read counts for each gene. All aesthetics are inherited from the first line of the ggplot code. This means for every geometry that you add, it will use MeanCounts for the x variable unless otherwise noted inside a geometry.

To add a layer, we use the + symbol. Certain layers require additional information in order for them to show in your plot. If certain information is missing, or nothing is added, the command will use it’s built in defaults to guess what you are trying to see. If there is no default set, it will throw out an error indicating that some default parameter is missing.

In this case, we set bins = 20 to create 20 bins across the range of MeanCounts. You can also modify bin width to a fixed size.

ggplot(SampleGenes, aes(MeanCounts)) +   
 geom\_histogram(binwidth = 100)



### Modifying Fill and Color

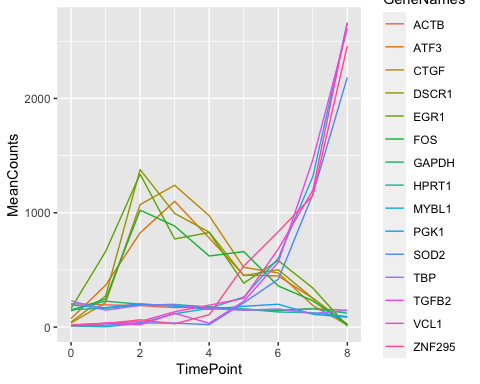
ggplot can systematically assign fill and outline colors based on assigned categories. In this case, we have a cluster variable that we can use to assign colors to different clusters. To assign a fill color you can assign fill = Cluster as an argument in your ggplot aesthetics. You can easily change the fill assignment to whatever variable you are most interested in separating by color. all you have to do is change the value of fill to GeneNames.

## Visualizing Relationships Between Two Variables

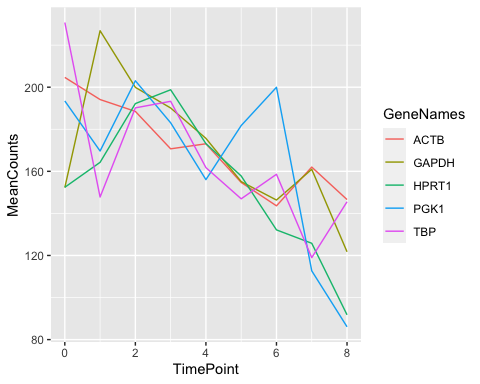
Using our example we can take a look at how gene expression changes over time using line plots. There are multiple approaches for visualizing relationships between variables that you can use from ggplot(). geom\_point() uses quantitative x and y aesthetics to create a scatterplot comparing two variables. geom\_line() uses quantitative x and y aesthetics to “connect the dots” for related variables. This geometry is best for looking at individual trajectories, but is not as effective at looking at groups. geom\_smooth() uses grouping categories to create linear models, or a line of best fit, for data within each category. This geometry is better for groups, but can get messy if looking at many individual trajectories.

Most illustration software allows you to change the fill color of object and also the outline color of object. R follows these same principals. You may notice that for our histograms we assigned colors using fill =. In order to assign line colors you must use the color = argument. We will use color = to assign the line colors based on GeneNames for our line plots.

# Plot Mean Counts (y) over time (x). Use color = to distinguish between genes  
ggplot(SampleGenes, aes(x = TimePoint, y = MeanCounts, color = GeneNames)) +   
 geom\_line()



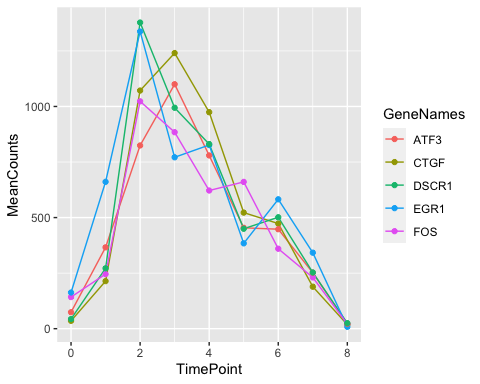
# create a data frame from the Gene expression set that only contains  
# housekeeping genes and plot their gene names  
  
HousekeepingGenes <- SampleGenes %>%  
 filter(Cluster == "Housekeeping")  
  
# Recreate our previous line plot for our filtered data set. Notice if we do not  
# include "x = " or "y = " ggplot will assume you list x first and y second.  
# What happens if you set color = cluster?  
ggplot(HousekeepingGenes, aes(TimePoint, MeanCounts, color = GeneNames)) +   
 geom\_line()



### Adding Additional Layers

Add additional layers to emphasize features of the plot. In the following exercises we will create a line plot from the “early” genes and add points to emphasize transition points. We will create a line plot from the early genes and add a geom\_smooth() layer to emphasize the overall trend in the data.

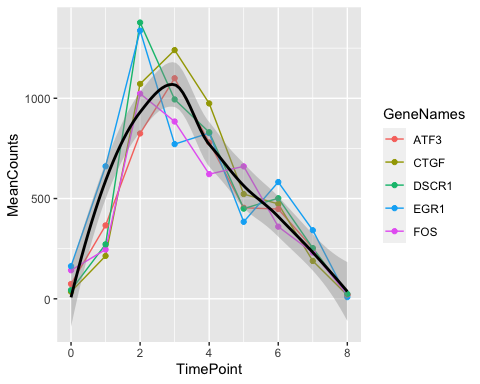
# Create a data frame from the gene expression set that only contains early genes  
EarlyGenes <- SampleGenes %>%  
 filter(Cluster == "Early")  
  
# Add geom\_point to add points to your line plot  
ggplot(EarlyGenes, aes(x = TimePoint, y = MeanCounts, color = GeneNames)) +   
 geom\_line() +   
 geom\_point()



Use geom\_smooth() instead of geom\_line() to create a smoothed line of the data. This is best for visualizing data with a linear relationship since it will create a line of best fit for all data points in the group

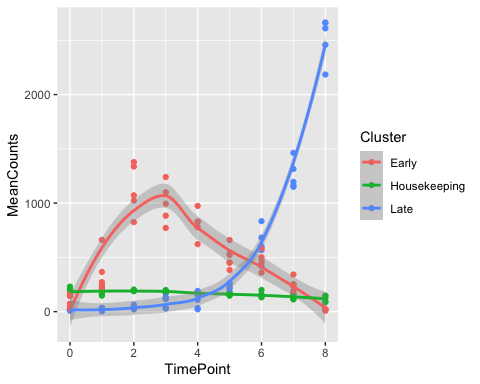
# Use geom\_smooth() to create a smoothed line of the data. This is best for visualizing data with a linear relationship since it will create a line of best fit around all data points in the group  
ggplot(EarlyGenes, aes(x = TimePoint, y = MeanCounts, color = GeneNames)) +   
 geom\_line()+  
 geom\_point()+  
 geom\_smooth(color = "black")

## `geom\_smooth()` using method = 'loess' and formula = 'y ~ x'



# Try it out with the SampleGenes data set, grouping color by cluster.  
ggplot(SampleGenes, aes(x = TimePoint, y = MeanCounts, color = Cluster)) +   
 geom\_point()+  
 geom\_smooth()

## `geom\_smooth()` using method = 'loess' and formula = 'y ~ x'

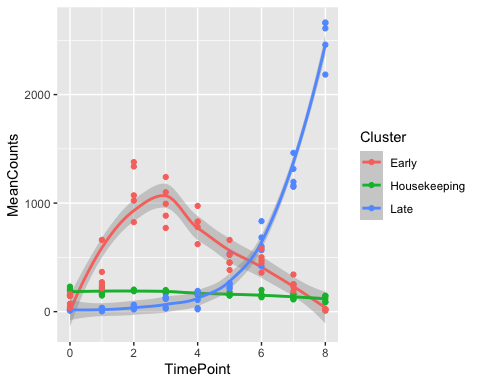


geom\_smooth() uses statistical methods to create a line of best fit. It uses one of several methods for line smoothing added as an argument in the command.

* method = "lm" creates a basic linear model
* method = "glm" creates a generalized linear model
* method = "gam" creates a generalized additive model
* method = "loess" creates a locally weighted scatterplot smoothing model

In addition to depicting a line of best fit, the model will depict the confidence interval for the line shown. The confidence interval is shown as a greyed area around the line of best fit. Depending on the number of layers or the complexity of the plot, you may or may not want to show the confidence interval. To remove the confidence interval you may add “se = FALSE” as an argument in the command. See the examples below.

## `geom\_smooth()` using formula = 'y ~ x'

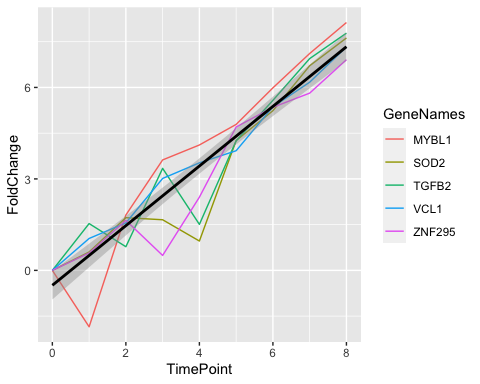
 ### Challenge 1

You will rarely need to visualize raw read counts for genes over time. Instead, most data is normalized to its baseline condition (in this case TimePoint = 0). Use dplyr to calculate a new column with the log2 fold change compared to baseline.

Plot a line graph of the log2 fold change of the Late genes over the 8 time points. Include a layer with a line of best fit shown in black.

T0 <- SampleGenes %>%  
 group\_by(TimePoint) %>%  
 filter(TimePoint == 0)  
  
NormalizedData <- SampleGenes %>%  
 group\_by(TimePoint) %>%  
 mutate(FoldChange = log2(MeanCounts/T0$MeanCounts)) %>%  
 filter(Cluster == "Late")  
  
ggplot(NormalizedData, aes(x = TimePoint, y = FoldChange, color = GeneNames)) +  
 geom\_line()+  
 geom\_smooth(method = "lm", color = "black")

## `geom\_smooth()` using formula = 'y ~ x'



## Visualizing Distributions for Multiple Groups

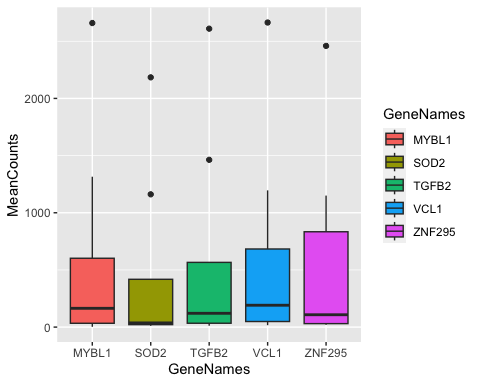
Another extremely important aspect of data visualization includes **depicting sample ranges**, spread and variability. There are number of ways to do this. In this section of our lesson we will cover **box plots**, **violin plots**, **dot plots** and **bar plots** as these are some of the most common used for plotting sample distributions.

### Boxplots

A default boxplot is constructed based on five points. The minimum value, the quartile 1 value (25th percentile), the median, the quartile 3 value (75th percentile) and the maximum value. The box in a box plot is created around quartile 1 (Q1) and quartile 3 (Q3) and is considered the “interquartile range” or IQR.

For normally distributed data, outliers are defined as values that are 1.5xIQR less than Q1 or 1.5xIQR more than Q3. Observations outside the range of the IQR are shown as dots separated from the rest of the box plot. Let’s create boxplots from our data set using geom\_boxplot().

# Create a boxplot showing the distribution of gene expression for the Late genes in our sample.  
LateGenes <- SampleGenes %>%  
 filter(Cluster == "Late")  
  
ggplot(LateGenes, aes(x = GeneNames, y = MeanCounts, fill = GeneNames)) +   
 geom\_boxplot()

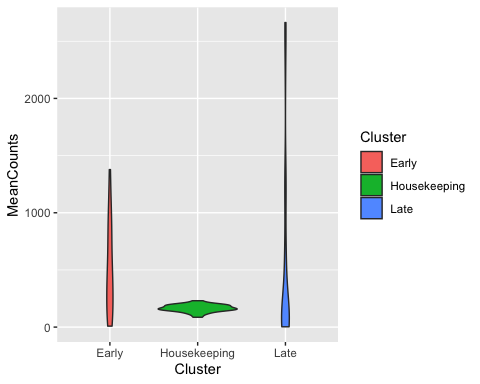
 ### Violin Plots and Dot Plots

Violin plots are a means of depicting data distribution for multiple categorical variables simultaneously. Violin plots depict the distribution of quantitative values calculated for each group. In our practice data set, we have three groups with distinct overall read counts and distributions. Let’s use violin plots to view these overall distributions. The command to create a violin plot is geom\_violin()

geom\_violin() will work as a command if all the correct aesthetics are included, but one key method of customization for violin plots is the scale = parameter. There are three main scales you can include in the geom\_violin() command.

* scale = "area scales the plot so that each group has an equal area
* scale = "width" scales the plot so that all groups have an equal maximum width
* scale = "count" scales the plot so that each group is scaled based on the total number of observations.

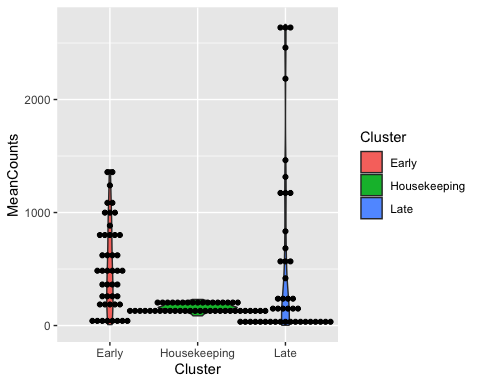
# Use our SampleGenes set and geom\_violin to create violin plots of our data  
ggplot(SampleGenes, aes(x = Cluster, y = MeanCounts, fill = Cluster)) +   
 geom\_violin(scale = "area")

 Add a dotplot layer with geom\_dotplot() on top of the violin plot layer to see how violin distributions are constructed. In the example below we have defined several arguments inside geom\_dotplot() to make sure the data is readable. Dotplots depict distributions similar to histograms in that they create bins where the dots can fall so the plot looks organized. Adding a dotplot layer on top of the violin plot allows you to see all of the individual values that are used to create the violin plot with a distribution scaled based on the total number of points.

* binaxis = "y" This argument sets which axis to bin the data on. Options are read as characters, and you can inclide the “x” or “y” axis
* stackdir = "center" Dot plots stack dots within each bin. They can stack the dots from a “left”, “center”, or “right” direction.
* dotsize = 0.5 The size of the dots is scaled based on the size of the bin. A value of 0.5 means the dot is half the size of the bin, a value of 2 means the dot is twice the size of the bin.

ggplot(SampleGenes, aes(x = Cluster, y = MeanCounts, fill = Cluster)) +   
 geom\_violin(scale = "area") +   
 geom\_dotplot(binaxis = "y", stackdir = "center", fill = "black", dotsize = 0.5)

## Bin width defaults to 1/30 of the range of the data. Pick better value with  
## `binwidth`.

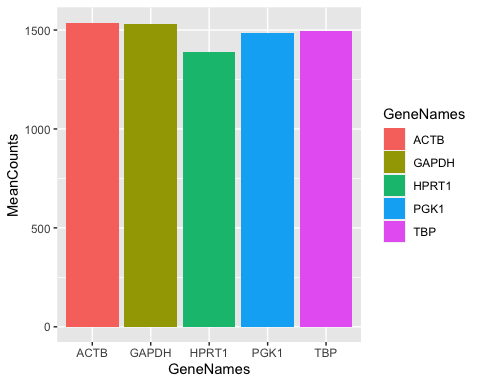


### Bar plots

Bar and column plots are some of the most common plots that you will see presented in research. If you are used to data visualization in Microsoft Excel, you will quickly notice that the commands for bar and column plots geom\_bar() and geom\_col() behave differently than when you create a bar plot in Excel. In Excel, you can select values and it will automatically calculate averages and display standard deviation values based on groups in the data set. In contrast, R will try to depict every single value offered. So instead of showing averages and standard deviations, bar plots will show sums.

In our previous session we calculated means for groups using the mutate() function in dplyr. In order to create bar or column plots that show mean and standard deviations, we must calculate those before plotting. In the example below we will first look to see what happens if we use geom\_col() to plot the MeanCounts for Housekeeping genes.

ggplot(HousekeepingGenes, aes(x = GeneNames, y = MeanCounts, fill = GeneNames)) +   
 geom\_col()



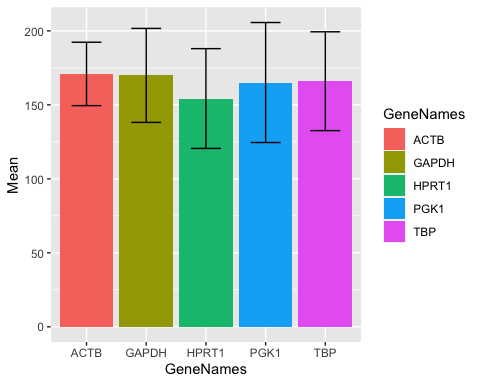
You can see in the example above that all of the observations are simply stacked on top of one another. We are not seeing average counts per million as we might expect if the command would function as it would in Excel. Fortunately, we can quickly calculate mean and standard devition and format our table using the summarize() function.

The summarize() function works similarly to mutate() in that each argument sets a column name equal to a formula which is used to fill in the values for that column. Summarize works particularly well with the group\_by() function to calculate summary statistics like mean, median, standard deviaiton and more. To plot the mean and standard devotion of the read counts for each housekeeping gene we would group by GeneNames and calculate the mean and standard deviation using the summarize() function.

* Mean = mean(MeanCounts) - this will create a new column called Mean and report the average of everything indicated by group\_by()
* StanDev = sd(MeanCounts) - this will create a new column called StanDev and report the standard deviation in the indicated groups.

In order to *show* the standard deviations in your plot, you can add a layer geom\_errorbar(). geom\_errorbar() requires two additional aesthetics ymin and ymax. These values set the limits to the error bars that you calculate. sd() will calculate the value for one standard deviation, which is typically what is shown in research publications. Therefore, we can simply tell ggplot the ymin = Mean - Standev and ymax = Mean + Standev and ggplot will automatically calculate visualize that for each group. Review the example below which visualizes what we described in the section above.

HousekeepingMeans <- HousekeepingGenes %>%  
 group\_by(GeneNames) %>% # group the data by category of interest   
 summarize(Mean = mean(MeanCounts), StanDev = sd(MeanCounts))  
  
ggplot(HousekeepingMeans, aes(x = GeneNames, y = Mean, fill = GeneNames,   
 ymin = Mean-StanDev, ymax = Mean + StanDev)) +   
 geom\_col() +   
 geom\_errorbar(width = 0.5)



### Plotting Summary

In this section we have described that the grammar of graphics relies on three essential components. First a data set to inform the plot. Second a coordinate system (x, y and z mapping) so that the computer can identify which elements should go where. Third geometries used for depicting the patterns, distributions or summaries you want to report.

ggplot() commands follow this exact logic. The **first** element in a ggplot command is the data frame used to inform the plot. The **second** element is called the aesthetics which helps ggplot map the data to its coordinate system. Aesthetics are included with aes() argument where you indicate what it should use as the x variable, the y variable, and if any other variables should be mapped to the data like z, fill, color, ymin, ymax, etc. Including aesthetics is **required** for ggplots to be constructed. The **third** element are the geom commands multiple geom’s can be added to each plot separated by the + symbol. Below we include a table of the geoms covered in this section.

| Geometry | Command | Useful Arguments (See Help tab for full list) |
| --- | --- | --- |
| Histogram | geom\_histogram() | bins = , binwidth = |
| Line Plot | geom\_line() | none required, see Help tab for full list |
| Scatter Plot | geom\_point() | none required, see Help tab for full list |
| Smooth Plot | geom\_smooth() | method =, se = |
| Box Plot | geom\_boxplot() | none required, see Help tab for full list |
| Violin Plot | geom\_violin() | scale = |
| Dot Plot | geom\_dotplot() | binaxis =, stacdir =, dotsize = |
| Bar Plot | geom\_col() | none required, see Help tab for full list |
| Error Bars | geom\_errorbar() | ymin =, ymax = in aes and width = |

### Challenge 2

Create a boxplot showing the read count distribution for genes in each respective cluster.

Create a bar chart showing the mean and standard deviation of the read counts for all genes that fall under each cluster.

## Customizing Theme and Colors

One principle for effective data visualizations is that you want a reader to require as little time as possible to understand as much data as possible. This often means having very clear plot titles, labels for x and y axis, adding or removing figure legends, modifying font sizes, and much more. In this section we will cover how to include plot labels using the labs() command, modifying colors with RColorBrewer, and making axis text modifications with theme().

### Adding plot labels

The command to customize plot labels is labs(). This command is added using a + symbol and most commonly includes arguments for the title, subtitle, x and y axis labels. Let’s review the boxplot we used to view the mean read count distribution for LateGenes, and give it some proper labels.

ggplot(LateGenes, aes(x = GeneNames, y = MeanCounts, fill = GeneNames)) +  
 geom\_boxplot() +  
 labs(title = "Read Count Distribution for Late Genes",  
 x = "Genes",  
 y = "Read Counts")



# Copy the code above and add an argument assigning a subtitle.

If you mapped the color or fill to certain characteristics you can use the labs() command to modify how that is represented in the figure legend. To do that you can include fill = or color = as comma separated arguments inside the labs() command. See the example below.

ggplot(LateGenes, aes(x = GeneNames, y = MeanCounts, fill = GeneNames)) +  
 geom\_boxplot() +  
 labs(title = "Read Count Distribution for Late Genes",  
 x = "Genes",  
 y = "Read Counts",  
 fill = "Genes")



Using the labs() command is one of the quickest ways to clean up the labels in your plots. When we get to the theme() section, we will cover more customization for labels and legends.

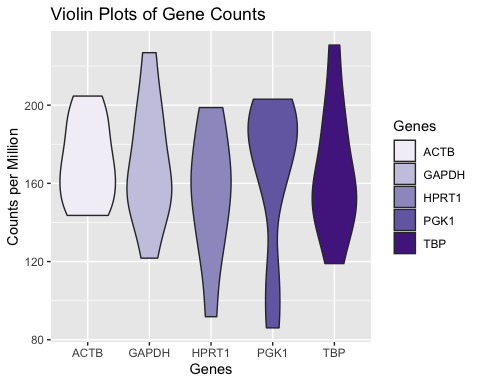
### Modifying plot colors with RColorBrewer

In the plots throughout the lesson you might have noticed that the default color palette is **erhm**, not the cutest. Best practices on using colors in plotting are extensively documented. Color modification is also important when considering accessibility, and ensuring you use an inclusive visual design.

RColorBrewer has set itself apart as the standard for color modification and customization. The documentation is not always easy to find, but we will cover the basics in this section. The way it works is that if you have assigned a fill value then you would use the scale\_fill\_brewer() command. If you have assigned a color value, then you would use the scale\_color\_brewer() command to modify those colors. Within these commands, the only required argument is to indicate which color palette to use. The palette you use can be one of many that are built into RColorBrewer. [Check out RColorBrewer’s default color palettes.](https://r-graph-gallery.com/38-rcolorbrewers-palettes.html)

In the example below, we modify the color with RColorBrewer on the violin plot of the Early Genes.

ggplot(HousekeepingGenes, aes(x = GeneNames, y = MeanCounts, fill = GeneNames)) +   
 geom\_violin(scale = "area") +   
 scale\_fill\_brewer(palette = "Purples") +  
 labs(title = "Violin Plots of Gene Counts",   
 x = "Genes",   
 y = "Counts per Million",  
 fill = "Genes")

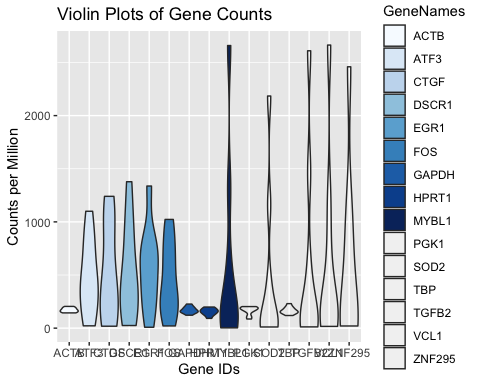


One limitation of RColorBrewer is that each default palette has a fixed number of shades built into the palettes. However, you can work around this limitation by using the scale\_fill\_manual() or the scale\_color\_manual() commands in conjunction with the ColorRampPalette() These commands allow you to assign the number of shades that you want, and R will systematically create a gradient that accommodates the number of shades requested.

In the example below lets look at what happens when there are not enough colors for the number of groups included in the violin plot.

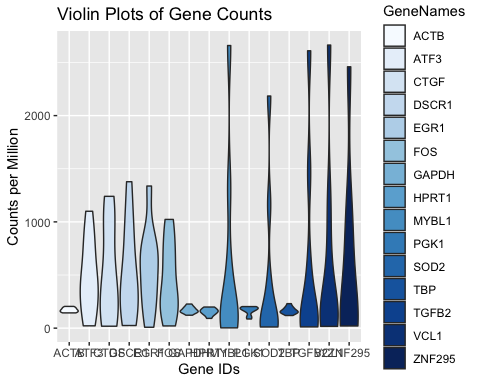
ggplot(SampleGenes, aes(x = GeneNames, y = MeanCounts, fill = GeneNames)) +   
 geom\_violin(scale = "width") +   
 scale\_fill\_brewer(palette = "Blues") +   
 labs(title = "Violin Plots of Gene Counts",   
 x = "Gene IDs",   
 y = "Counts per Million")

## Warning in RColorBrewer::brewer.pal(n, pal): n too large, allowed maximum for palette Blues is 9  
## Returning the palette you asked for with that many colors



This plot is obviously a mess, it will take a few steps to clean it up. First, lets teach RColorBrewer the number of shades that we need for our plot. In this case, we create a scalar called “ncolors”. If we want a color for each gene in our data set, we must count the number of unique times a particular gene name is mentioned. You can use unique() to remove duplicates from a vector and use length() to count the number of values. Since our data frame has the names for all the genes, we can pull the data directly from the data frame.

# identify the number of times a unique gene name occurs in your SampleGenes data.  
ncolors <- length(unique(SampleGenes$GeneNames))  
  
# Once we have the number of colors that we need. We can use the scale\_fill\_manual()  
# command to create the color palette needed. Notice the syntax below  
# Getting the parenthesis right is crucial for this code to work  
  
ggplot(SampleGenes, aes(x = GeneNames, y = MeanCounts, fill = GeneNames)) +   
 geom\_violin(scale = "width") +   
 scale\_fill\_manual(values = colorRampPalette(brewer.pal(9, "Blues"))(ncolors)) +   
 labs(title = "Violin Plots of Gene Counts",   
 x = "Gene IDs",   
 y = "Counts per Million")



Notice that the scale\_fill\_manual() command has a complex argument. It wants a series of HEX codes which are commonly used in graphic design software to define colors. To generate the 15 hex codes needed to satisfy the values = argument you can use colorRampPalette() which will return a vector of HEX codes given an RColorBrewer color palette, the range of the palette to use, and a number of groups in the plot.

In the console below run each component of the command written below and see what it returns.

* brewer.pal(9, "Blues")
* colorRampPalette(brewer.pal(9, "Blues"))
* colorRampPalette(brewer.pal(9, "Blues"))(ncolors)

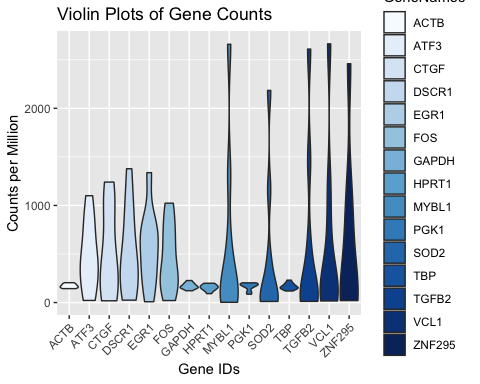
There is far too much built into RColorBrewer for us to cover in class. When in doubt about how to modify the colors of your plot, Ask your favorite search engine and check to see if anyone has asked (and answered) your plotting question on sites like Stack Overflow or R Bloggers.

### Modifying Theme and Details

Just because we were able to change the colors, doesn’t mean our plot is in good shape. The most expansive set of customizatinos in ggplot come from the theme() command. It is not required to create a plot, but many detail modifications are done within this command. Below we will cover what to do to clean up our example plot, but our lesson is hardly scratching the surface of what you can do with theme().

If you review the plot above you can see that the gene names on the x axis are all overlapping and there are so many names in the figure legend that they are flowing off the plot. In the example below we use the theme() command to angle the text of the x axis, and set the horizontal justification to align with the plot.

ggplot(SampleGenes, aes(x = GeneNames, y = MeanCounts, fill = GeneNames)) +   
 geom\_violin(scale = "width") +   
 scale\_fill\_manual(values = colorRampPalette(brewer.pal(9, "Blues"))(ncolors)) +   
 theme(axis.text.x = element\_text(angle = 45, hjust = 1)) +   
 labs(title = "Violin Plots of Gene Counts",   
 x = "Gene IDs",   
 y = "Counts per Million")



To modify parameters for the text on the x axis we add the axis.text.x = argument. Regarding the axis text we use element\_text() to set one of several parameters used to modify that text. Below we include a list of the most commonly editable parameters.

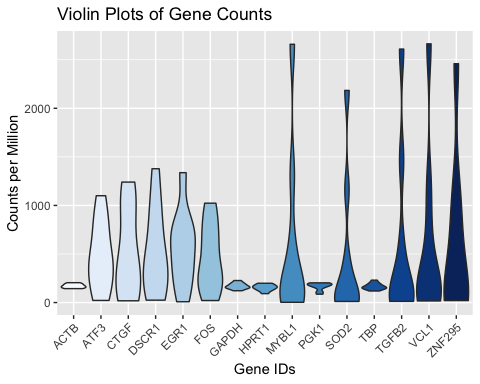
* color = - sets the color for the element (use a HEX code for this)
* size = - sets the font size for the element (an number larger than 0)
* hjust = - sets the horizontal justification (a number between 0 and 1)
* vjust = - sets the vertical justification (a number between 0 and 1)
* angle = - sets the angle of the text (a number between 0 and 180)

You can also modify more than 90 other aspects of the plot (use the Help tab for an exhaustive list of possible editable theme features). Below are some common features to modify

* title - modify the title for the plot
* axis.title.x - modify the label for the x axis
* axis.title.y - modify the label for the y axis
* axis.text.x - modify the text for the x axis values
* axis.text.y - modify the text for the y axis values
* legend.text - modify the text for the legend
* legend.title - modify the title for the legend
* legend.position - modify the position of the legend, or remove it completely

In addition to modifying any of the above aspects listed, you can remove them completely as well. For aspects relating to text you can simply set the value of the aspect to element\_blank(). Since the x axis text and the figure legend labels are redundant, we can either remove the x axis text or remove the legend completely to clean up our plot. In this case, the legend might be the bigger problem for our plot. Set legend.position = "none" to remove the legend completely.

ggplot(SampleGenes, aes(x = GeneNames, y = MeanCounts, fill = GeneNames)) +   
 geom\_violin(scale = "width") +   
 scale\_fill\_manual(values = colorRampPalette(brewer.pal(9, "Blues"))(ncolors)) +   
 theme(axis.text.x = element\_text(angle = 45, hjust = 1),  
 legend.position = "none") +   
 labs(title = "Violin Plots of Gene Counts",   
 x = "Gene IDs",   
 y = "Counts per Million")

 ## Exporting figures

We can export publication-quality figures with ggsave(). The review the arguments below to see

ggsave(  
 "Heatmap.pdf", # a character with a file type suffix (ie: pdf, png, jpg)  
 plot = last\_plot(), # default is last plot or any plot assigned as an object.  
 device = "pdf", # R will assume the correct format if it is in the name  
 width = 10, # width value  
 height = 10, # height value  
 units = c("cm"), # units of measurement  
 dpi = 300 # dots per inch resolution  
)

### Challenge 3

Immediate early gene: FOS is a gene that transiently increases expression in neurons after stimulation before returning to baseline levels. It has been theorized that this activity plays a role in learning and memory via synaptic plasticity, which may be defective in certain diseases like intellectual disability. Using the “IEG.csv” dataset in your data folder for this lesson, generate the following plots and answer the questions below.

* Create a line plot that shows FOS expression over time in controls and patient cells. Be sure to normalize this data to the GAPDH housekeeping gene. Do patient cells show the expected increase in FOS expression?
* One patient line is clearly an outlier when it comes to FOS expression at time zero. Choose the appropriate plot that shows this and plot the data.