Advanced Lab 2: Introduction to ggplot

Introduction

In this lab we will delve more deeply into the ggplot2[1] graphics package and how we can use it to create elaborate graphs.

ggplot2 builds on the idea of a grammar of graphics[2]. The basic concept in the grammar of graphics is that we build a graph iteratively by specifying various components of it:

data	Specify the the data set to be used.
aesthetics	Determine which variables will be shown on the graph and how. For example one variable could be assigned to correspond to the x axis, another to correspond to a color, one to correspond to size and so on. Aesthetics can either be **mapped** to variables, using the aes (...) syntax, or they can be **set** to constant values, e.g. geom_smooth(color="red"), as part of a specific component indicating the aesthetic to be used for that component.
geometries	Usually abbreviated as **geom**s. Determine the kinds of shapes we want to have included in the graph (points, lines, bars etc). We can have multiple geoms on the same graph, creating various *layers*.
stats	Determine what statistic of the variable will be visualized. This often is done at the same time as specifying a geom, but it can also be done separately.
facets	Determine if we should create separate panels based on the levels of a factor variable. These would be called “panel variables” in SPSS.
scales	Control how data values are mapped to visual values. Often default scales work just fine, but some times you may want to adjust those, especially when you choose colors to represent a variable’s levels.
annotations	Add text and similar elements to a graph.
themes	Control the overall graph appearance (axes ticks, labels, legend etc).

This will all make more sense as we move along.

You can load ggplot2 itself directly, or you can load it as part of the hanoverbase package:

```
library(hanoverbase)
```

A first example: Color vs Price for diamonds

We will use the built-in diamonds data for this lab. You may want to start a new project for this, and put most of this work in an R Markdown document.

```
data(diamonds)
View(diamonds)    # Do the View in the console
?diamonds         # If you want to open up the dataset documentation
```

[1] https://ggplot2.tidyverse.org/
[2] http://vita.had.co.nz/papers/layered-grammar.html

We would like to investigate how the color relates to the price of the diamonds. We start by defining the dataset to use:

```
ggplot(diamonds)
```

This will not show much (an empty plot), but it tells ggplot to use the diamonds dataset. Next we will specify the "aesthetics"; in this case we will tell it to use the color variable for the x-axis and the price variable for the y-axis:

```
ggplot(diamonds) + aes(x=color, y=price)
```

Notice the syntax: we add new components to our graph via the "plus" operation.

This should have produced axes, but it has plotted no data yet. This is because we have not specified any geom layer yet. But specifying the aesthetics was enough for ggplot to work out some scales for the axes.

We now add a geom layer with another "plus". We will spread this onto a new line to keep the lines short. To do that, you need to end each previous line with the plus sign, to instruct R that there is more to this command.

```
ggplot(diamonds) +
    aes(x=color, y=price) +
    geom_boxplot()
```

Note that the price tends to increase as the color gets *worse* (J is worse than D). This sounds opposite to what it should be, and we'll examine that later. If you'd like to see a different kind of plot before moving on, try using violin instead of boxplot in the above expression.

Now let's add one more layer, namely a point for each color, representing the mean price:

```
ggplot(diamonds) +
    aes(x=color, y=price) +
    geom_boxplot() +
    geom_point(stat="summary", fun.y=mean, color="red", fill="red", size=3)
```

Things just got a lot more complicated in the code! Let's walk through it.

We are here telling the graph to add "points" (geom_point), where it uses a specific stat function, namely "summary", to compute the y coordinates of these points. The default stat is the identity, which would not work very well here (but give it a go!); it would in effect plot a point for every data row, at the corresponding x and y coordinates. It will in effect be drawing a scatterplot. This summary stat will instead take all y values for the same x value, and *summarize* them using the provided function, mean, specified via the fun.y argument. Under the hood the system actually calls the function stat_summary, and you can look at its documentation for more details.

The remaining parameters merely customize the look and feel of the points, namely a specific border *color* and *fill* value for all the points, and a *size* for the points.

Let's also add lines connecting the means:

```
ggplot(diamonds) +
    aes(x=color, y=price) +
    geom_boxplot() +
    geom_line(stat="summary", group=1, fun.y=mean, size=1, linetype="dashed") +
    geom_point(stat="summary", fun.y=mean, color="red", fill="red", size=3)
```

This is similar, except for one new bit, namely the group=1 argument. geom_line and other line-drawing geoms need to be told a way to decide which group of values is part of the same line/path. The group parameter specifies exactly that; all points with the same group value will be joined together. In this instance we tell it that all values should correspond to the same group, namely group "1". Oftentimes in other graphs we may have subjects with a specific ID, and then we would specify group=subject or something similar. Or we might have some data on the economies of countries over many years, and we want one line drawn for each country; we would then use group=country.

Note that we added the geom_line *before* the geom_point. The elements will be drawn in that order, and that order may affect the appearance.

As the distribution of prices is quite skewed, let's adjust the y scale to use a logarithmic scale:

```
ggplot(diamonds) +
    aes(x=color, y=price) +
    geom_boxplot() +
    geom_line(stat="summary", group=1, fun.y=mean, size=1, linetype="dashed") +
    geom_point(stat="summary", fun.y=mean, color="red", fill="red", size=3) +
    scale_y_log10()
```

This performed a log transform on the prices *before* it computed the boxplot, then drew the graph with a logarithmic y-scale but showing the original price values there. This is important to keep in mind: these scale transformations occur *before* any statistics are performed on the values. This may not always be what you want, and we will see later how we can change the scale on a graph *after* all computations are performed.

Next, let's specify some more tick marks. We do this by setting the breaks parameter of the scale transform (the values we specify are our un-transformed values):

```
ggplot(diamonds) +
    aes(x=color, y=price) +
    geom_boxplot() +
    geom_line(stat="summary", group=1, fun.y=mean, size=1, linetype="dashed") +
    geom_point(stat="summary", fun.y=mean, color="red", fill="red", size=3) +
    scale_y_log10(breaks=seq(1000, 13000, by=2000))
```

Now, let's use a custom theme to change the look and feel of the graph. We'll remove the vertical grid lines, set the background to white and the horizontal grid lines to light green:

```
ggplot(diamonds) +
    aes(x=color, y=price) +
    geom_boxplot() +
    geom_line(stat="summary", group=1, fun.y=mean, size=1, linetype="dashed") +
    geom_point(stat="summary", fun.y=mean, color="red", fill="red", size=3) +
    scale_y_log10(breaks=seq(1000, 13000, by=2000)) +
    theme_light() +
    theme(
        panel.grid.major.y = element_line(color="lightgreen"),
        panel.grid.major.x = element_blank()
    )
```

We added two components here: theme_light() set some default settings for a light-colored theme, while theme(...) refined the theme settings further.

It's worth mentioning that you can save any part of this graph-building process, and reuse it in other graphs. For example, we can create our own theme and store in a variable. In order to do that, we place the components we want to use into a *list*. ggplot2 will unpack the list and put the components with pluses, when needed.

```
ourTheme <- list(
  theme_light(),
  theme(
      panel.grid.major.y = element_line(color="lightgreen"),
      panel.grid.major.x = element_blank()
  )
)

ggplot(diamonds) +
    aes(x=color, y=price) +
    geom_boxplot() +
    geom_line(stat="summary", group=1, fun.y=mean, size=1, linetype="dashed") +
    geom_point(stat="summary", fun.y=mean, color="red", fill="red", size=3) +
    scale_y_log10(breaks=seq(1000, 13000, by=2000)) +
    ourTheme
```

A list is a bit like c, but can actually contain any kinds of elements, even other sublists. c on the other hand is really only meant for concatenating like items to produce a sequence of homogeneous values.

We can also save our own boxplot with the extra mean lines added:

```
boxplotWithMeans <- list(
    geom_boxplot(),
    geom_line(stat="summary", group=1, fun.y=mean, size=1, linetype="dashed"),
    geom_point(stat="summary", fun.y=mean, color="red", fill="red", size=3)
)

priceLogScale <- scale_y_log10(breaks=seq(1000, 13000, by=2000))

ggplot(diamonds) +
    aes(x=color, y=price) +
    boxplotWithMeans +
    priceLogScale +
    ourTheme
```

Practice: Do a similar graph to compare price and clarity as well as price and cut. What would be your initial observations about the relations?

Scatterplots and faceting

Let's now examine the relation between price and carats:

```
ggplot(diamonds) +
    aes(carat, price) +
    geom_point()
```

This relation will look better in logarithmic scale:

```
ggplot(diamonds) +
    aes(carat, price) +
    geom_point() +
    scale_x_log10() +
    scale_y_log10()
```

Before moving on, let's store the basic parameters and scaling in an intermediate variable, so we don't have to type them every time:

```
priceVsCaratPlot <- ggplot(diamonds) +
    aes(carat, price) +
    scale_x_log10() +
    scale_y_log10()

priceVsCaratPlot + geom_point()
```

Let's try to add a transparency to each point:

```
priceVsCaratPlot + geom_point(alpha=0.3)
```

Or we can try to use single dots for each point rather than circles:

```
priceVsCaratPlot + geom_point(shape=".")
```

Given the large number of data points in this plot, a better approach is to use bin2d, which is like a two-dimensional histogram; it breaks the area in small rectangles, counts the number of points that fall in each rectangle, then uses color intensity to signify areas with more values:

```
priceVsCaratPlot + geom_bin2d(bins=50)
```

Now let's add a linear regression line with error estimates:

```
priceVsCaratPlot +
    geom_bin2d(bins=50) +
    geom_lm(interval="prediction")
```

Or we might prefer to fit a smooth line instead:

```
priceVsCaratPlot +
    geom_bin2d(bins=50) +
    geom_smooth(color="red")
```

We will now use facets to create a panelled graph and investigate this relation based on a third variable (e.g. cut, clarity, or color):

```
priceVsCaratPlot +
    geom_bin2d(bins=50) +
    geom_smooth(color="red") +
    facet_wrap(~cut)
```

We can do two factors:

```
priceVsCaratPlot +
    geom_bin2d(bins=50) +
    geom_smooth(color="red") +
    facet_wrap(~cut + color)
```

With two factors, it is best to use facet_grid (we also used the ggtitle annotation to add a title):

```
priceVsCaratPlot +
    geom_bin2d(bins=50) +
    geom_smooth(color="red") +
    facet_grid(cut ~ color) +
    ggtitle("Diamond price vs carat separated by cut and color")
```

Practice: Use these techniques to compare the width (y) and length (x) of diamonds. You may have to filter the data to get a better view.

Barcharts for categorical variables

We can use bar charts to investigate relations between the categorical variables. Let's start with individual bar charts. Here we look at a bar chart of the color variable:

```
ggplot(diamonds) +
    aes(x=color) +
    geom_bar(fill="blue")
```

We can add different colors depending on which diamond color the bar represents. This requires a new aesthetic:

```
ggplot(diamonds) +
    aes(x=color) +
    geom_bar(aes(fill=color))
```

Note that the aes here shows up within the geom_bar. This means that it would only affect that geom_bar and not other geoms that we might add.

Note also the difference between the two examples. In the first we used fill ="blue" to specify a fixed fill color for all bar charts. In the second, we wanted the fill color to actually correspond to a variable in the data set. This required an "aesthetic" specification.

Let's briefly talk about choosing the color palette to use. This is basically setting a **scale** for the fill variable. These would be added as new components, via the "plus" operator. It would be a discrete scale. There are four different ways of specifying a discrete color scale:

- Directly listing the colors:

```
scale_fill_manual(values=c("blue", "green", "orange", "red",
                           "magenta", "purple", "pink"))
```

- Using one of the preset "color brewer" palettes (see options with display .brewer. all ())

```
scale_fill_brewer(palette="Set1")
```

- Using a grey-scale:

```
scale_fill_grey(start=0.3, end=0.7)
```

- Using a scale based on the HCL color wheel:

```
scale_fill_hue(l = 40)
```

Practice: Try adding one of these now.

Let's compare two categorical variables, say `cut` and `clarity`:

```
ggplot(diamonds) +
    aes(x=cut, fill=clarity) +
    geom_bar()
```

Let's we could have specifically specified that it is stacked:

```
ggplot(diamonds) +
    aes(x=cut, fill=clarity) +
    geom_bar(position="stack")
```

And now 100% stacked:

```
ggplot(diamonds) +
    aes(x=cut, fill=clarity) +
    geom_bar(position="fill")
```

We could have done a clustered bar chart using `position="dodge"`.

You can also try to combine this with a facet:

```
ggplot(diamonds) +
    aes(x=cut, fill=clarity) +
    geom_bar(position="fill") +
    facet_wrap(~color)
```

We can also add lines at the bar locations, to follow the trends (not meant as an example of a good graph, just an illustration of the package's capabilities):

```
ggplot(diamonds) +
    aes(x=cut, fill=clarity) +
    geom_bar(position="fill") +
    geom_line(aes(group=clarity), stat="count", position="fill") +
    facet_wrap(~color)
```

We can control the "angle" of the x axis labels via a theme parameter:

```
ggplot(diamonds) +
    aes(x=cut, fill=clarity) +
    geom_bar(position="fill") +
    geom_line(aes(group=clarity), stat="count", position="fill") +
    facet_wrap(~color) +
    theme(
        axis.text.x = element_text(angle=-45, vjust=0.5)
    )
```

Pie Charts

Creating a pie chart is somewhat more elaborate. The main idea is as follows: We create a bar chart of one stacked bar, and then we make a change to polar coordinates, which wraps the y axis into a circle. This will in effect use the y axis as an angle specification, and the x axis as a radius specification.

To be clear, we are not advocating use of pie charts; there are typically better ways to demonstrate the needed information. But examining how to do pie charts in ggplot2 does demonstrate some interesting and useful ideas.

As a first step, let's create a bar chart of one stacked bar based on cut. Each bar chart however needs an x variable. We "fake" such a variable by using factor(1), which creates a single common level (labeled 1) for all the data rows.

```
ggplot(diamonds) +
    aes(x=factor(1), fill=cut) +
    geom_bar()
```

The current graph has extra space on the bar sides, which is the default behavior for bar charts as the bars are not supposed to connect to each other. In our case this extra space will present problems when we turn the graph to polar coordinates, as it will create a hole in the middle. We will therefore eliminate the space by specifying the bar width. We will also remove the extraneous top and bottom spacing by setting the expand parameter of the y scale, which tells the scale how much to expand beyond its range of values (typically a minimum of 4% expansion is advisable, but again for pie charts that would cause problems):

```
ggplot(diamonds) +
    aes(x=factor(1), fill=cut) +
    geom_bar(width=1) +
    scale_y_continuous(expand=c(0, 0))
```

Next we will remove the axis labeling and ticks, by specifying a theme:

```
ggplot(diamonds) +
    aes(x=factor(1), fill=cut) +
    geom_bar(width=1) +
    scale_y_continuous(expand=c(0, 0)) +
    theme_void()
```

Lastly, we convert to polar coordinates. We also add borders to the bars/slices. We will store the graph in a variable for later use.

```
ourPieChart <- ggplot(diamonds) +
    aes(x=factor(1), fill=cut) +
    geom_bar(width=1, color="white", size=0.2) +
    scale_y_continuous(expand=c(0, 0)) +
    theme_void() +
    coord_polar(theta="y")
ourPieChart
```

To incorporate the percentages as slice labels, we need a bit more work. We start by creating a new data frame that computes the needed counts and proportions:

```
cutCounts <- diamonds %>% group_by(cut) %>%
             summarize(count = n()) %>%
             mutate(percent = 100 * count / sum(count))
cutCounts
```

We will want to format that percent so that it has fewer digits and a percent symbol after it:

```
cutCounts <- diamonds %>% group_by(cut) %>%
             summarize(count = n()) %>%
             mutate(percent = 100 * count / sum(count)) %>%
             mutate(percent = format(percent, digits = 2)) %>%
             mutate(percent = paste(percent, "%", sep=""))
cutCounts
```

Then we add a new geom to our graph, which draws from the cutCounts dataset:

```
ourPieChart +
    geom_text(cutCounts,
               aes(x=1.2, y=count, label=percent),
               position=position_stack(vjust=0.5))
```

Solving the mystery

So let's now try to answer the question of why diamonds with worse color/clarity are actually more expensive. The answer lies in the effect of the diamond size (carat) on the price: bigger diamonds are more expensive but also harder to get in the best color/clarity.

So we would like to condition on the size as we examine the relationship between price and color. In order to do that we need to "bin" the carat variable to turn it into a categorical variable. We will do this in a moment.

First let's restrict the carat range to exclude outliers. Here is the carat range:

```
ggplot(diamonds) +
    aes(x=carat) +
    geom_histogram(bins=40)
```

We will restrict our analysis to diamonds up to 2.5 carats. The filter method from dplyr can help us here.

```
diamonds %>%
    filter (carat <= 2.5) %>%
    ggplot() +
        aes(x=carat) +
        geom_histogram(bins=40)
```

Let's look at how these are distributed amongst the different colors:

```
diamonds %>%
    filter(carat <= 2.5) %>%
    ggplot() +
        aes(x=carat, fill=color) +
        geom_density() +
        scale_fill_brewer(palette="Blues")
```

This graph shows us the overlapping density curves for each color. A better view will stack the curves (you can also try position="stack" if you like):

```
diamonds %>%
    filter(carat <= 2.5) %>%
    ggplot() +
        aes(x=carat, fill=color) +
        geom_density(position="fill") +
        scale_fill_brewer(palette="Blues")
```

We can see from this graph that most of the best-color diamonds are small.

Practice: What if we used clarity instead of color?

Now we will group the carat variable into "bins". We can use one of the cut_ methods for this:

- cut_interval lets you specify how many bins to use.
- cut_width lets you specify how wide the bins will be.
- cut_number lets you specify a uniform bin frequency.

Let's facet the a boxplot of price vs color based on a carat grouping, using the cut_interval method:

```
diamonds %>%
    filter(carat <= 2.5) %>%
    ggplot() +
        aes(x=color, y=price) +
        geom_boxplot() +
        facet_wrap(~cut_interval(carat, 6))
```

The vast difference in value ranges on the various panels makes it hard to clearly see the patterns. To that end, we would like to set each panel to have independent scales. reading the documentation of facet_wrap we learn about the scales parameter:

```
diamonds %>%
    filter(carat <= 2.5) %>%
    ggplot() +
        aes(x=color, y=price) +
        geom_boxplot() +
        facet_wrap(~cut_interval(carat, 6), scales="free")
```

We may return to this example in our regression session.

Practice: Make a similar paneled boxplot using cut in place of color. Adjust the x-axis labels (45 degree angle).

Further Practice

In this section we provide a series of practice questions based on the gapdata dataset from the previous session. If you do not have that dataset available, you can download the data from this file: datasets/gapdata.RData[3]. Save the file and then upload it to your project directory, and use a command like load("gapdata.RData") within R to load the data into the environment.

The load and save methods allow us to store R objects and reload them at a later time. They store information in an internal R-specific data format that is not human-readable, so for data sets it is best to use something like write_csv or write_excel_csv to export the data in a more broadly shareable format.

The following questions assume that you have the gapdata dataset active.

1. Filter the data to focus on the most recent year, and draw a scatterplot of life expectancy compared to income, with the population determining the point size and with region determining the point fill. Use shape=21 to have points that allow a boundary color via color="black" for example, as well as an interior/fill color via fill=region. You should try to do this in steps:

[3] ../datasets/gapdata.RData

- Get a basic point graph of life expectancy against income for 2015, using geom_point.
- Add a logarithmic scale on x (should be the income variable).
- Add a fill =region aesthetic to geom_point. This will not really have a visible effect until the next step.
- Add a shape=21 setting to geom_point.
- Set the size aesthetic to equal the population.
- Try a different color palette, such as scale_color_brewer (palette ="Set1").
- Because population is used in an aesthetic, the population will now show up on the legend, and we may or may not want to do that. We can control this via a scale_size_continuous layer. For example scale_size_continuous (guide="none") will remove it from the legend.

2. Still focusing on the most recent year, use group_by and summarize to compute average life expectancy, population and income values for each region. Draw a similar graph to the previous exercise using this grouped data, making any necessary adjustments to the variable names. The resulting graph should have one point per region.
3. Now, including *all* the years, group by region and year, and then use summarize as in 2. Use arrange to make sure the data is in chronological order. Then draw a similar graph as 2, but now including all the years. You can also add a geom_line component, using group=region, to show the evolution of these averages over time. Make sure to include it in the code before the geom_point. You can adjust the lines sizes by setting the size parameter.
4. Use filter (country %in% c("Colombia", "Honduras", "Nicaragua", " Haiti ", "Mexico")) to restrict gapdata to these countries (you can add more if you like but there are some interesting patterns in the above), then arrange by year and graph life expectancy against year, using line plots grouped and colored by country. You should notice spikes in the graph at two particular points in time, related to natural disasters (one earthquake, one hurricane). You may want to filter further, to focus on the years since 1990.
5. Carry out a similar graph where we use filter (region=="South Asia"). You should see a another number of interesting spikes related to natural disasters.
6. A nice graph of life expectancy vs year can be drawn for the middle east region:
   - Filter the data with filter (region=="Middle East & North Africa"), and restrict to years since 1980.
   - Draw line and point plots (you may need to adjust the point size), colored and grouped by country, and using country as a facet_wrap. You should see some interesting patterns, both of spikes and of small slope. You may want to add a theme(legend. position ="none") layer to omit the legend.
7. For this exercise, we start by computing for each country the relative increase in per-capita income from 2000 to 2010. This is mostly dplyr work. Since this rather complex, you should pause after after each step to run the code and examine the results. You can pipe the result to a View for a quick look at it.
   - Filter the data to only include those two years. The predicate year %in% c(2000, 2010) can be used for that.
   - Select the country, region, income and year variables only.
   - Use the spread method to spread the income information over two columns, one for each year. The spread method expects two arguments, the "key" to use for the column names (here year) and the "value" from which to draw the values (here income).
   - Use mutate to create a new variable, relincr that computes the relative increase in income from 2000 to 2010.
   - Arrange the resulting data on this new variable.

- Give this result the relIncreases name for easier future use.

Now we will make some graphs using this relIncreases dataset.

- Start with a basic histogram of the relative increases, and notice a number of countries at 1 (100%) or more, meaning that the income more than doubled in that decade.
- You may also want to try geom_dotplot(binwidth=0.05).
- Now make a plot of relIncreases using geom_point, using reorder(country, relincr) as a y-aesthetic to order the countries by their increase. Give geom_point an aesthetic to color the points by region. Use custom figure size (R chunk option) to get a better view.
  - Variation 1: Add a facet_wrap by region with scales="free_y" to allow each y-scale to vary independently of the others.
  - Variation 2: Add a facet_grid by region: facet_grid(region ~., scales="free_y", space="free_y"). Notice how the output from facet_grid differs from that of facet_wrap.
  - Variation 3: Add a facet_wrap by cut_number(relincr) to split the range of values into three bins of equal frequency and display the resulting data in three panels.
- Finally, here's one possible end result graph to check out:

```
relIncreases %>%
    ggplot() +
        aes(x=relincr, y=reorder(country, relincr), color=region) +
        geom_segment(aes(xend=0, yend=reorder(country, relincr))) +
        geom_point() +
        facet_wrap(~cut_number(relincr, 3), scales="free_y") +
        theme(legend.position="bottom")
```

Appendix

Available Aesthetics

You can learn more about the available aesthetics for each kind of geom here[4]. Inside an aes call these can be bound to data variables. Outside of it they can be given specific values.

color	The color of the elements (boundary color when applicable)
fill	The fill color for elements that allow it
size	The relative size of the element
stroke	The size of line strokes
shape	The shape of the points
weight	The weight to assign to counts in bar charts
group	The values to use to group points that are part of the same line
linetype	The type of line to be drawn (solid, dashed etc)
linejoin	The way that the line segments are to join where they meet
lineend	The way that the line segments end
x	The values assigned to the x-axis
y	The values assigned to the y-axis
xend	The x coordinate of the end of the segment
yend	The y coordinate of the end of the segment

[4] https://ggplot2.tidyverse.org/articles/ggplot2-specs.html

label	The label to be used for text elements
family	The font family to use
fontface	The emphasis that is put on the font (normal, italic, bold)
hjust	Horizontal justification of the text center relative to coordinates
vjust	Vertical justification of the text center relative to coordinates

Available Geometries

See this page[5] for a full list and more details. This is by no means a complete list, check the webpage for more.

Empty Graph	geom_blank
Reference Lines	geom_abline, geom_hline, geom_vline
Bar Chart	geom_bar, geom_col
2d bin count heatmap	geom_bin2d
Box and whiskers plot	geom_boxplot
2d Contours	geom_contour
Point Plot	geom_point, geom_count, geom_jitter
Density Plot	geom_density
Dot Plot	geom_dotplot
Horizontal Error Bar	geom_errorbarh
Histogram	geom_histogram, geom_freqpoly
Vertical Intervals and Error Bars	geom_crossbar, geom_errorbar, geom_linerange, geom_pointrange
Maps	geom_map
Line Plot	geom_path, geom_line, geom_step, geom_segment
Fits	geom_smooth, geom_lm

Some extra Stats

Most stat functions are simply tied to a geom and used that way. But some stats are worth mentioning:

stat_ecdf	Compute empirical cumulative distribution
stat_ellipse	Compute normal confidence ellipses
stat_function	Compute function for each x value
stat_identity	Leave data as is
stat_summary_2d	Bin and summarise in 2d (rectangle)
stat_summary_hex	Bin and summarise in 2d (hexagons)
stat_summary	Summarise y values at unique x
stat_summary_bin	Summarise y values at binned x
stat_unique	Remove duplicates

[5]https://ggplot2.tidyverse.org/reference/index.html#section-layer-geoms

Positioning adjustments

These are useful for moving objects around to avoid their overlaps. This is all from this page[6].

position_dodge, position_dodge2	Dodge overlapping objects side-to-side
position_identity	Don't adjust position
position_jitter	Jitter points to avoid overplotting
position_jitterdodge	Simultaneously dodge and jitter
position_nudge	Nudge points a fixed distance
position_stack , position_fill	Stack overlapping objects on top of each another

Annotations

Annotations[7] use fixed values, rather than data points.

Reference Lines	geom_abline, geom_hline, geom_vline
Generic annotation layer	annotate
Log tick marks	annotation_logticks
Maps	annotation_map, borders
Rectangular Tiling	annotation_raster

Scales

See here[8]. Each color method also has a corresponding fill method, and each x has a corresponding y.

Axis Labels, Legend, Titles	labs, xlab, ylab, ggtitle
Scale Limits	lims, xlim, ylim, expand_limits
Transparency Scales	scale_alpha, scale_alpha_continuous, scale_alpha_discrete , scale_alpha_ordinal
Color Brewer Package Scales	scale_color_brewer, scale_fill_brewer , scale_color_distiller , scale_fill_distiller
Continuous Color Scales	scale_color_continuous , scale_fill_continuous
Position Scales	scale_x_continuous, scale_x_log10, scale_x_reverse, scale_x_sqrt
Date/Time Scales	scale_x_date, scale_x_datetime, scale_x_time
Discrete Data Scales	scale_x_discrete
Gradient Color Scales	scale_color_gradient , scale_color_gradient2 , scale_color_gradientn
Gray Scale	scale_color_grey
Hue	scale_color_hue
Line Patterms	scale_linetype , scale_linetype_continuous , scale_linetype_discrete
Manual Scales	scale_color_manual, scale_size_manual, scale_shape_manual, etc
Shape Scale	scale_shape
Radius Scales	scale_radius , scale_size , scale_size_area

[6]https://ggplot2.tidyverse.org/reference/index.html#section-layer-position-adjustment
[7]https://ggplot2.tidyverse.org/reference/index.html#section-layer-annotations
[8]https://ggplot2.tidyverse.org/reference/index.html#section-scales

Coordinate Systems

See here[9]

Cartesian coordinates	coord_cartesian
Cartesian coordinates with fixed "aspect ratio"	coord_fixed
Cartesian coordinates with x and y flipped	coord_flip
Map projections	coord_map coord_quickmap
Polar coordinates	coord_polar
Transformed Cartesian coordinate system	coord_trans

[9] https://ggplot2.tidyverse.org/reference/index.html#section-coordinate-systems