Lab: 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 We start each graph by simply specifying 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.
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 Can be used to 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
```

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)
```

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

This will not show much (and 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 color for the x axis and price 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, go remind R that there is more to this command.

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

Note that the price seems 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 having violin instead of boxplot in the above expression.

Now let us 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 plot the points exactly where they are. This summary stat instead will 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 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. The group parameter specifies exactly that. 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 i.d., 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.

As the price range is quite skewed, let us 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 actual price values there. This is important to keep in mind, these scale transformations occur *before* and statistics are performed on the values. This may not 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 us 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=c(1000, 3000, 5000, 70000, 9000, 11000, 13000))
```

Now, let us 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=c(1000, 3000, 5000, 70000, 9000, 11000, 13000)) +
    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:

```
ourTheme <- 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=c(1000, 3000, 5000, 70000, 9000, 11000, 13000)) +
```

```
    ourTheme
```

We can also save our own boxplot with the extra mean lines added. For that we will need to to store the three components that we want to include into a list. ggplot2 will unpack the list and put the components with pluses:

```
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=c(1000, 3000, 5000, 70000, 9000, 11000, 13000))

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 us 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 us store the basic parameters and scaling in an intermediate variable, so we don't have to type them every time:

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

caratVsPricePlot + geom_point()
```

Let us try to add a transparency to each point: r caratVsPricePlot + geom_point(alpha=0.3)‘ ‘ Or we can try to use single d caratVsPricePlot + geom_point(shape=“.”) “‘

Given the large number of values in this example, 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:

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

Now let us add a linear regression line with error estimates:

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

Or we might prefer a "smooth" line fit instead:

```
caratVsPricePlot +
    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):

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

We can do two factors:

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

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

```
caratVsPricePlot +
    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 barcharts to investigate relations between the categorical variables. Let's start with individual barcharts. Here we look at a bar chart of the color variable:

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

Question: How would we add color to the bars?

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

Let us briefly talk about choosing the color palette to use. This is basically setting a **scale** for the fill variable. 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 `scale_fill_brewer(palette="Set1")` (see options with `display.brewer.all()`)
- 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`

Practice: Try adding one of these now.

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

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

Let's make it stacked:

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

And now 100% stacked:

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

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 of the possibilities):

```
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 us 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 spaces 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 whole 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 turn into 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 add percentages, a lot more work needs to be done. We can obtain the percentages as follows:

```
diamonds$cut %>% table() %>% prop.table()
```

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(), percent= 100 * n()/nrow(diamonds))
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(), percent= 100 * n()/nrow(diamonds)) %>%
             mutate(percent = format(percent, digits = 2)) %>%
             mutate(percent = paste(percent, "%", sep=""))
cutCounts
```

Then we add a new geom to our graph, which draws from this new different dataset rather than the dataset used in the pie chart:

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

Solving the mystery

So let us try to now answer the question on 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 would need to "bin" the carat variable to turn into a categorical variable. We will do this in a moment.

First let us 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.

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

Let us 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.

Now we will facet the graph 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.

Further Practice

In this section we will list a series of practice questions based off of the gpdata dataset from the previous session. If you do not have that dataset available, you can download the data from this file: datasets/gpdata.RData[3]. Save the file and then upload it to your project directory, and use a command like load("gpdata.RData") within R to load the data into the environment. The load and save methods allow us to store R objects like that and load them on the other 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 gpdata 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:
    - Get a basic point graph of life expectancy against income, using geom_point.
    - Add logarithmic scale on x.
    - Add a fill =region aesthetic.
    - Change the shape to 21, and add a fixed color="black" setting to the geom_point.
    - Set the size aesthetic to equal the population.

[3] ../datasets/gpdata.RData

- The population will now show up on the legend, and we may or may not want to have 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, then draw a similar graph to the previous exercise.
3. 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 will also want to set size=1 or something like that to keep the lines small.
4. Use filter(country %in% c("Colombia", "Honduras", "Nicaragua", "Haiti", "Mexico")) to restrict gpdata 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).
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 can be drawn for the middle east regions:
   - Filter the data with filter(region=="Middle East & North Africa"), an 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 will follow a more complex process: We start by computing for each country the relative increase in per-capita income from 2000 to 2010. This is mostly dplyr work:
   - 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 turn the income variable back in 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 that computes the relative increase from 2000 to 2010.
   - Arrange the resulting data on this new variable.
   - You may want to store the result of this process in a variable for easier future access.
   - You may want to start with a basic histogram of the relative increases, and notice a number of countries at 1 or more.
   - You may also want to try geom_dotplot(binwidth=0.05).
   - Another interesting plot is a geom_point where we use reorder(country, relincr) as a y aesthetic (relincr is our name for the relative increase variable, yours may vary). You may also want to facet wrap by the region and using scales="free_y" on it. Another possible facet is by cut_number(relincr, 3) to split the range of values into three pieces with equal number of elements.
   - 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")
```