AVML 2012: ggplot2

–Josef Fruehwald, AVML, University of York, 2012

- AVML 2012: ggplot2
  - Plotting principles
  - Installation
  - Grammar of Graphics
  - ggplot2 Materials
  - ggplot2 Basic Concepts
    - Layers
      - The data layer
      - The geometries layer
      - The statistics layer
      - Scale transformations
      - Cosmetic alterations
    - Aesthetics
      - Inheritance
      - Grouping
      - More aesthetics and their use.
    - Geometries
      - Some special geoms
        - geom_line() vs geom_path()
        - geom_text()
      - Positioning
        - Jitter
        - Dodge, Stack and Fill
        - Some new and exciting geoms
    - Statistics
      - Statistics and their Geometries
      - Particularly useful Statistics
        - stat_smooth()
        - stat_summary()
      - Values created by statistics
    - Scales
      - x and y scales
        - scale_ arguments
        - scale_[xy]_continuous
      - color and fill scales
        - Categorical color scales
        - Gradient color scales

- Guides
- shape and linetype
- Other scales
- More on Guides
- Coordinate systems
- Faceting
- facet_wrap
- facet_grid
- Additional options
- Building plots

Plotting principles

We are here to learn the basics of `ggplot2`. `ggplot2` *will* be useful for producing complex graphics relatively simply. It won't be of any use for figuring out what is a sensible, useful, or accurate plot. To get a good handle on those, I'd advise simply reading a lot about data visualization.

Installation

If you haven't already, take this opportunity to install the latest version of `ggplot2`.

```
install.packages("ggplot2")
```

```
library(ggplot2)
```

Right now, I'm running version `0.9.1`, and some of the default behaviors of `ggplot2` changed with version 0.9.0.

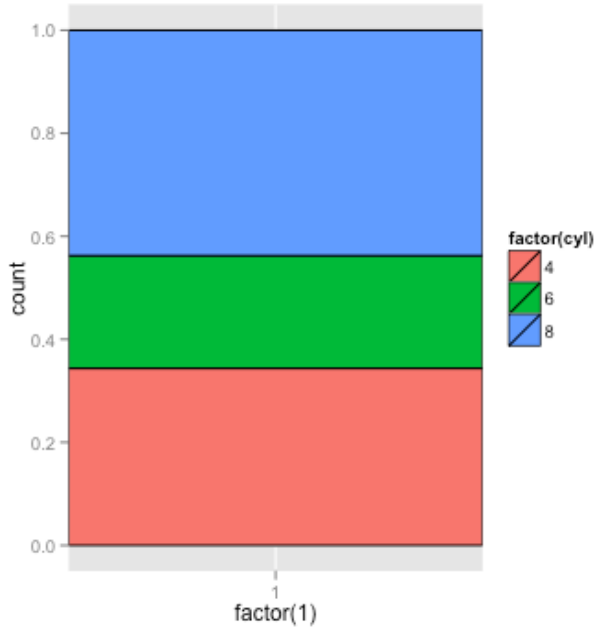
Grammar of Graphics

`ggplot2` is meant to be an implementation of the Grammar of Graphics, hence *gg* plot. The basic notion is that there is a grammar to the composition of graphical components in statistical graphics. By direcly controlling that grammar, you can generate a large set of carefully constructed graphics from a relatively small set of operations. As Wickham (2010), the author of `ggplot2` said,

> A good grammar will allow us to gain insight into the composition of complicated graphics, and reveal unexpected connections between seemingly different graphics

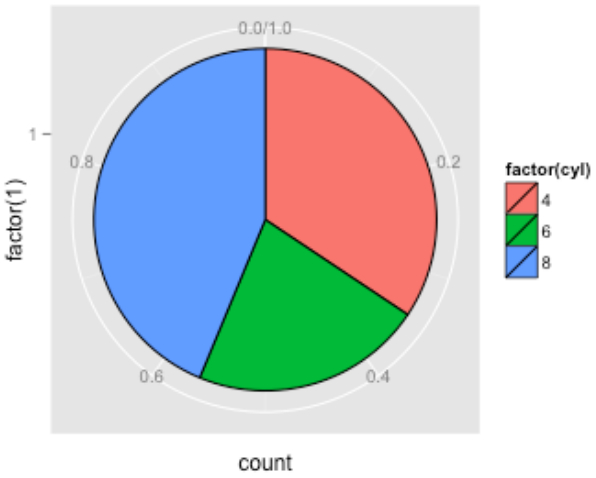
A good example of an unexpected connection would be that pie charts are just filled bar charts…

```
pie <- ggplot(mtcars, aes(x = factor(1), fill = factor(cyl))) +
       geom_bar(width = 1, position = "fill", color = "black")
pie
```



…in polar coordinates.

```
pie + coord_polar(theta = "y")
```



ggplot2 Materials

There are quite a few ggplot2 materials out there to guide you when you're not sure what to do next. First and foremost are the internal help pages for every piece of ggplot2 we'll cover here. They tend to be pretty useful.

```
?geom_jitter
```

- ggplot2 website
  - The main website has a pretty useful online reference manual.
- ggplot2 book
  - The ggplot2 book is an invaluable resource for beginners
- My own ggplot2 notes from 2010
- 0.9.0 transition guide (pdf)
  - This documents a lot of the new features, and some restructuring that was implemented in version 0.9.0.
- Learning R
  - The Learning R blog had a whole series of posts where they recreated almost all of the plots from the lattice book in ggplot2
- opts() list
  - A comprehensive list of possible cosmetic adjustments.

ggplot2 Basic Concepts

There are a few basic concepts to wrap your mind around for using ggplot2. First, we construct plots out of **layers**. Every component of the graph, from the underlying data it's plotting, to the coordinate system it's plotted on, to the statistical summaries overlaid on top, to the axis labels, are layers in the plot. The consequence of this is that your use of `ggplot2` will probably involve iterative addition of layer upon layer until you're pleased with the results.

Next, the graphical properties which encode the data you're presenting are the **aesthetics** of the plot. These include things like

- x position
- y position
- size of elements
- shape of elements
- color of elements

The actual graphical elements utilized in a plot are the **geometries**, like

- points
- lines
- line segments
- bars
- text

Some of these geometries have their own specific aesthetic settings. For example,

- points
  - point shape
- text
  - text labels
- lines
  - line weight
  - line type

You'll also frequently want to plot **statistics** overlaid on top of, or instead of the raw data. Some of these include

- Smoothing and regression lines
- One and two dimensional binning
- Mean and medians with confidence intervals.

The **aesthetics**, **geometries** and **statistics** constitute the most important **layers** of a plot, but for fine tuning a plot for publication, there are a number of other things you'll want to adjust. The most common one of these are the **scales**, which encompass things like

- A logarithmic x or y axis

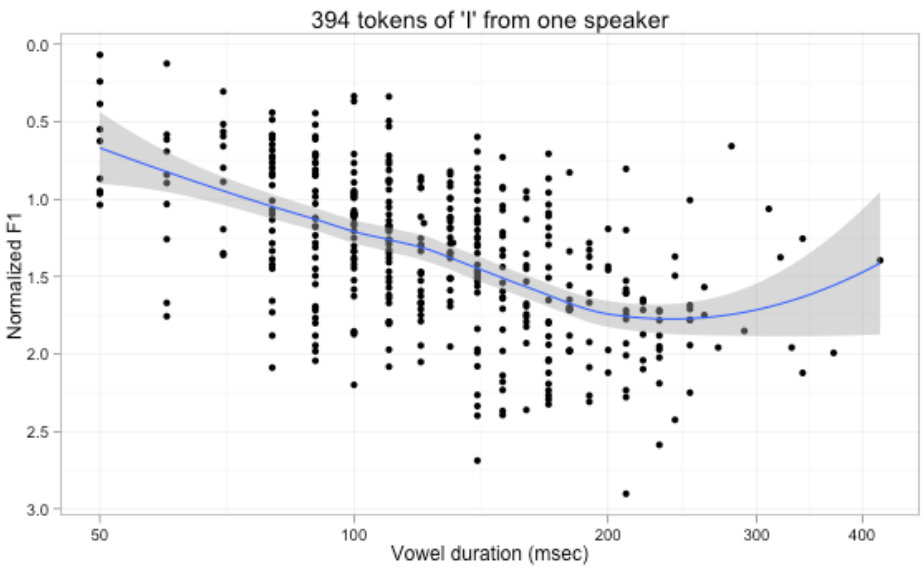
- Customized color scales
- Customized point shapes, or linetypes

The following sections are devoted to some of these basic elements in `ggplot2`.

Layers

We'll be constructing plots with `ggplot2` by building up “layers”. The layering of plot elements on top of each other is perhaps the most powerful aspect of the `ggplot2` system. It means that relatively complex plots are built up of modular parts, which you can iteratively add or remove. For example, take this figure, which plots the relationship between vowel duration and F1 for 394 tokens of the lexical item “I”.

```
I_jean <- read.delim("http://bit.ly/avml_ggplot2_data")
```



This plot is composed of nine layers, which can be subdivided into five layer types.

The data layer

Every `ggplot2` plot has a data layer, which defines the data set to plot, and the basic mappings of data to aesthetic elements. The data layer created with the functions `ggplot()` and `aes()`, and looks like this

```
ggplot(data, aes(...))
```

The first argument to `ggplot()` is a data frame (it *must* be a data frame), and its second argument is `aes()`. You're never going to use `aes()` in any other context except for inside of other `ggplot2` functions,

so it might be best not to think of `aes()` as its own function, but rather as a special way of defining data-to-aesthetic mappings.

For the plot from above, we'll be using data from the `I_jean` data frame, which looks like this:

```
head(I_jean)
```

```
##    Name Age Sex Word FolSegTrans Dur_msec     F1   F2
## 1 Jean   61   f  I'M           M      130  861.7 1336
## 2 Jean   61   f    I           N      140 1010.4 1349
## 3 Jean   61   f I'LL           L      110  670.1 1293
## 4 Jean   61   f  I'M           M      180  869.8 1307
## 5 Jean   61   f    I           R       80  743.0 1419
## 6 Jean   61   f I'VE           V      120  918.2 1581
##     F1.n    F2.n
## 1 1.6609 -0.8855
## 2 2.6883 -0.8536
## 3 0.3370 -0.9873
## 4 1.7168 -0.9536
## 5 0.8407 -0.6897
## 6 2.0512 -0.3068
```

I've decided that an interesting relationship in this data is between the vowel duration (`Dur_msec`) and the normalized F1 of the vowel (`F1.n`). Specifically, I'd like to map `Dur_msec` to the x-axis, and `F1.n` to the y-axis. Here's the `ggplot2` code.

```
p <- ggplot(I_jean, aes(x = Dur_msec, y = F1.n))
```

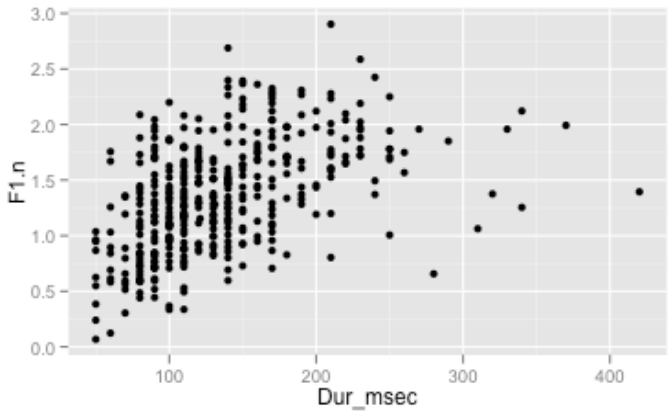
Right off the bat, we can see one way in which `ggplot2` is similar to `lattice`, and different from base graphics. If you've only ever used `plot()` in R, then you might be surprised to see me assigning the output of `ggplot2` to an object, because `plot()` just creates a plot, not an object. `ggplot2` plots, on the other hand, are objects, which you can assign, save, and manipulate.

If you try to print p right now though, you'll get an error. Right now, p is a `ggplot2` plot that's all data, but no graphical elements. Adding graphical elements, or *geoms* is the next step.

The geometries layer

The next step, after defining the basic data-to-aesthetic mappings, is to add geometries to the data. We'll discuss geometries in more detail below, but for now, we'll add one of the simplest: points.

```
p <- p + geom_point()
p
```

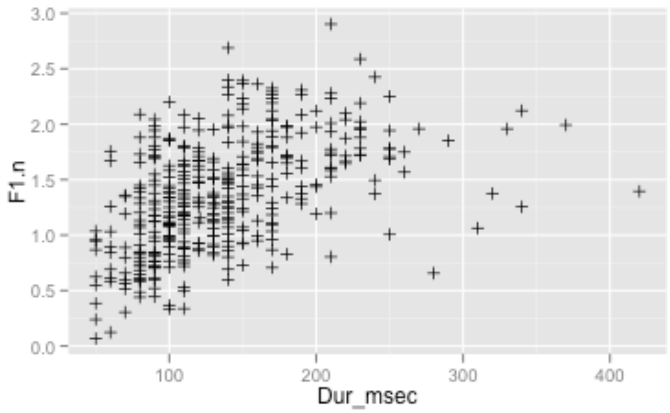


There are a few things to take away from this step. First and foremost, the way you add new layers, of any kind, to a plot is with the + operator. And, as we'll see in a moment, there's no need to only add them one at a time. You can string together any number of layers to add to a plot, separated by +.

The next thing to notice is that all layers you add to a plot are, technically, functions. We didn't pass any arguments to `geom_point()`, so the resulting plot represents the default behavior: solid black circular points.

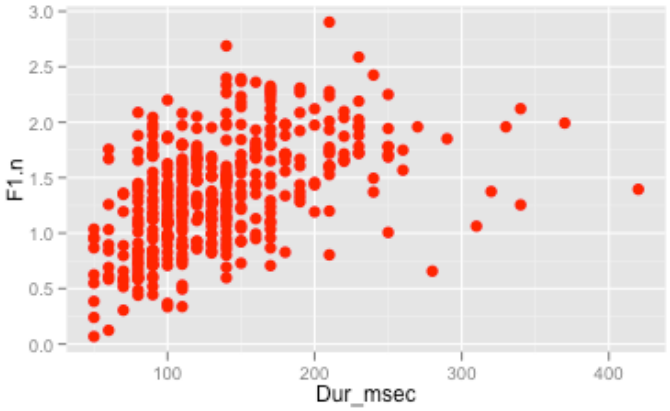
If for no good reason at all we wanted to use a different point shape in the plot, we could specify it inside of `geom_point()`.

```
ggplot(I_jean, aes(x=Dur_msec, y=F1.n)) +
  geom_point(shape = 3)
```



Or, if we wanted to use larger, red points, we could specify that in `geom_point()` as well.

```
ggplot(I_jean, aes(x=Dur_msec, y=F1.n)) +
  geom_point(color = "red", size = 3)
```



Speaking of defaults, we can see a few of the default setting of `ggplot2` on display here. Most striking is the light grey background, with white grid lines. Opinion varies on whether or not this is aesthetically or technically pleasing, but don't worry, it's adjustable.

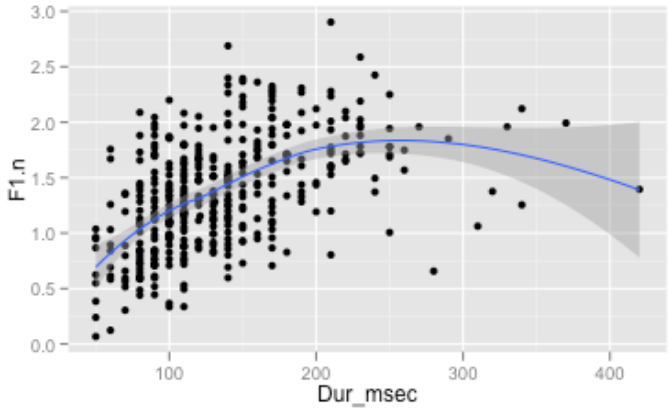
Another default is to label the x and y axes with the column names from the data frame. I'll inject a bit of best practice advice here, and tell you to *always* change the axis names. It's nearly guaranteed that your data frame column names will make for very poor axis labels. We'll cover how to do that shortly.

Finally, note that we didn't need to tell `geom_point()` about the x and y axes. This may seem trivial, but it's a really important, and powerful aspect of `ggplot2`. When you add any layer at all to a plot, it will **inherit** the data-to-aesthetic mappings which were defined in the data layer. We'll discuss inheritance, and how to override, or define new data-to-aesthetic mappings within any geom.

The statistics layer

The final figure also includes a smoothing line, which is one of many possible statistical layers we can add to a plot.

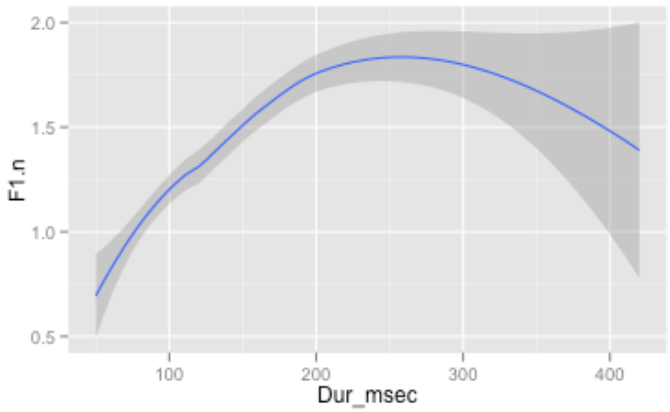
```
p <- p + stat_smooth()
p
```



We'll go over the default behavior of `stat_smooth()` below, but in this plot, the smoothing line represents a loess smooth, and the semi-transparent ribbon surrounding the solid line is the 95% confidence interval.

One important thing to realize is that it's not necessary to include the points in order to add a smoothing line. Here's what the plot would look like with the points omitted.

```
ggplot(I_jean, aes(x = Dur_msec, y = F1.n))+
  stat_smooth()
```

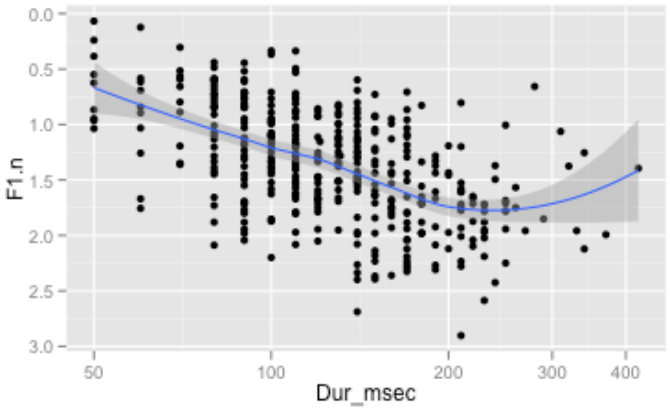


Notice how the y-axis has zoomed in to just include the range of the smoothing line and standard error.

Scale transformations

I also wanted to make some alterations to the default x and y axis scales. For example, the y-axis is currently running in reverse to the intuitive direction of F1. *Higher* vowels have *lower* F1 values, so we want to flip the y-axis. Additionally, durations are typically best displayed along a logarithmic scale, so we should convert the x-axis as well.

```
p <- p + scale_x_log10(breaks = c(50, 100,200,300,400))+
         scale_y_reverse()
p
```

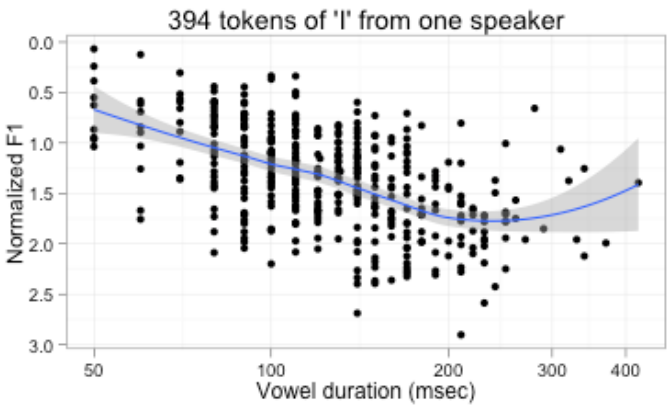


It's worth noting that the smoothing line here is calculated over the *transformed* data.

Cosmetic alterations

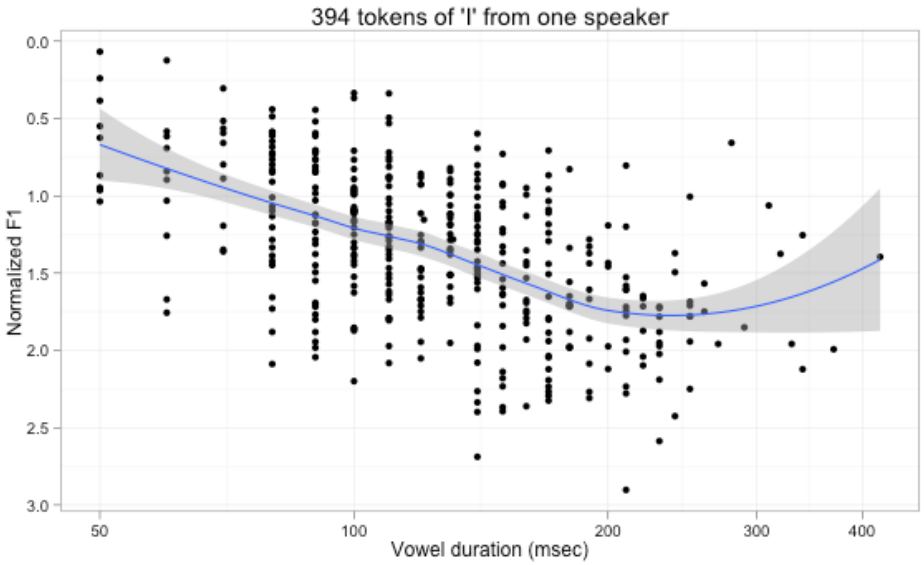
Finally, I wanted to make some cosmetic adjustments to the plot. For example, the x-axis label “Dur_msec” is not quite as useful as “Vowel duration (msec)” would be. I also added a title to the plot, and changed the color theme to black and white.

```
p <- p + ylab("Normalized F1")+
         xlab("Vowel duration (msec)")+
         theme_bw()+
         opts(title = "394 tokens of 'I' from one speaker")
p
```



Here's all the layers, put together all at the same time.

```
ggplot(I_jean, aes(x=Dur_msec, y=F1.n))+
  geom_point()+
  stat_smooth()+
  scale_x_log10(breaks = c(50, 100,200,300,400))+
  scale_y_reverse()+
  ylab("Normalized F1")+
  xlab("Vowel duration (msec)")+
  theme_bw()+
  opts(title = "394 tokens of 'I' from one speaker")
```



Aesthetics

In `ggplot2`, **aesthetics** are the graphical elements which are mapped to data, and they are defined with `aes()`. To some extent, the aesthetics you need to define are dependent on the geometries you want to use, because line segments have different geometric properties than points, for example. However, there is also a great deal of uniformity in the aesthetics used across geometries. Here is a list of the most common aesthetics you'll want to define.

- x
  - x-axis location
- y
  - y-axis location
- `color`
  - The color of lines, points, and the outside borders of two dimensional geometries (polygons, bars, etc.). Hadley Wickham,

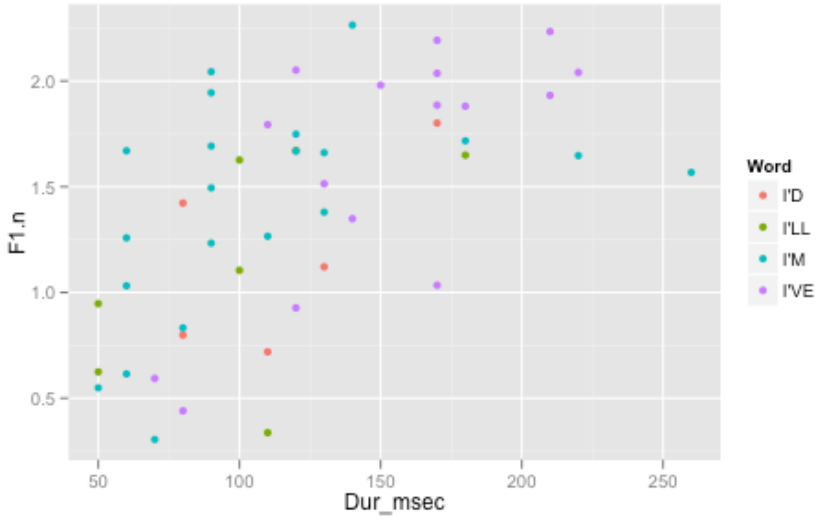
the primary `ggplot2` developer, is from New Zealand, so `colour` is also supported!

- `fill`
  - The fill color of two dimensional geometries.
- `size`
  - The size of points, or the weight of lines and borders of two dimensional geometries.
- `shape`
  - This is specific to points, and defines the point shape. This is one of the few aesthetics to which you can't map a continuous variable.
- `linetype`
  - This defines the line type of any kind of line, path, or border of a two dimensional geometry. This is another aesthetic which cannot be mapped to a continuous variable.
- `alpha`
  - This defines the opacity of any geometric property. It's less commonly mapped to data, and more often hard coded to a single value as a solution for overplotting.
- `xend`, `yend`
  - You'll use these more rarely, usually when plotting a line segment, or arrow. The beginning of the line segment will be located at `x`, `y`, and the end of the line segment will be located at `xend`,`yend`.
- `ymin`, `ymax`, (`xmin`, `xmax`)
  - `ymin` and `ymax` are reserved for geometries which are devoted to representing *ranges* of data, like error bars, and ribbons. For the most part, these will be expressed along the y-axis, but `xmin` and `xmax` are utilized for some geometries as well.

The most important thing to keep in mind about aesthetics is not what they're called, though, but how they are inherited by the layers. Let's start by mapping the `Word` to `color`. 88% of the tokens are just “I”, so lets create a subset of the data that excludes “I” so it doesn't visually swamp the plot.

```
I_subset <- subset(I_jean, Word != "I")
```

```
ggplot(I_subset, aes(Dur_msec, F1.n, color = Word))+
  geom_point()
```



Each point is now colored according to the word it corresponds to. `ggplot2` has automatically generated a color palette of the right type and size, based on the data mapped to `color`, and created a legend to the side. As with everything, the specific color palette we use is adjustable, which will be discussed in more detail below under Scales. The default `ggplot2` color palette is rather clever, however. Every color is equidistant around an HSL color circle, and have equal luminance. The idea is that no category should be accidentally visually emphasized, but they can be hard for some colorblind readers, and they will all print to the same shade of grey!

But on to more pressing matters. The only possible way to map the `Word` data to the color of points in the plot is to do it within `aes()`. If you are used to working with base graphics, then a lot of your instincts are wrong for `ggplot2`. For example, you might think the following would work:

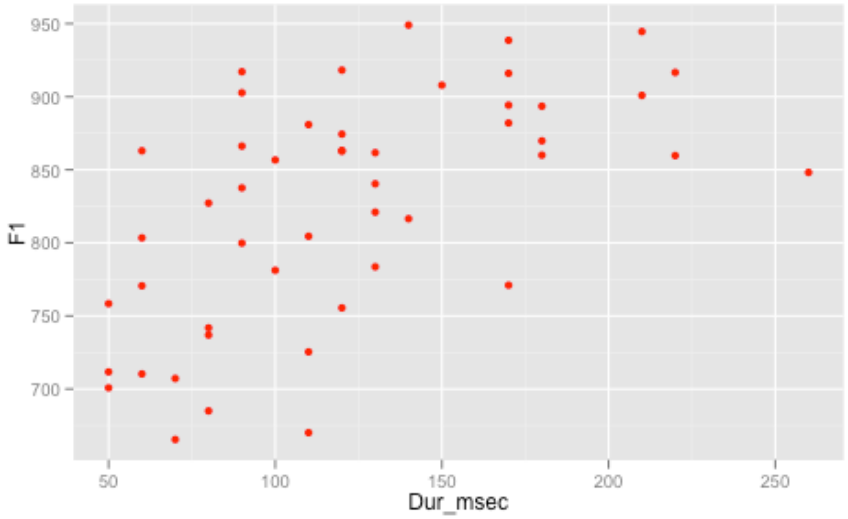
```
I_subset$Color <- c("black",
                    "red","blue",
                    "green","goldenrod")[I_subset$Word]

ggplot(I_subset, aes(Dur_msec, F1))+
  geom_point(color = I_subset$Color)
```

```
## Error: When _setting_ aesthetics, they may only take one
## value. Problems: colour
```

The error is yelling at us about the distinction between **setting** an aesthetic, and **mapping** an aesthetic. Recall from above that we were able to **set** the color of all points to red by saying so inside of `aes()`,

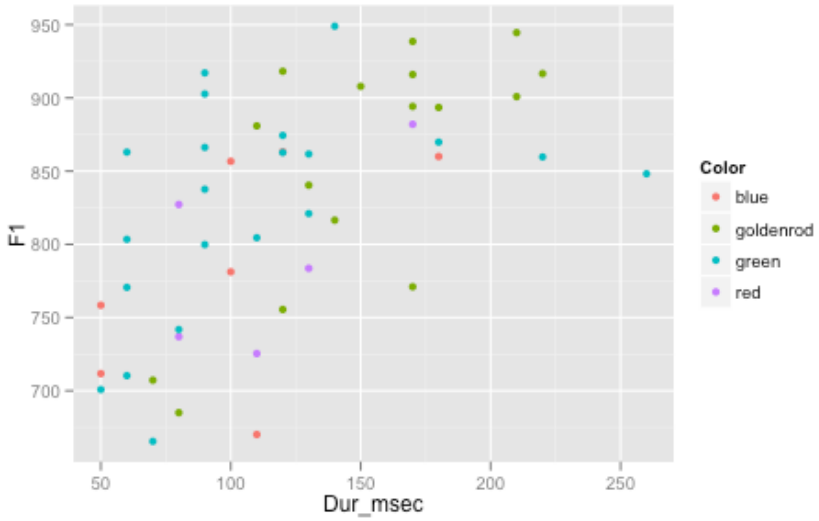
```
ggplot(I_subset, aes(Dur_msec, F1))+
  geom_point(color = "red")
```



But that's as far as you can go with that. Everything beyond **setting** a single color for all points in a plot constitutes **mapping** colors, and you have to use `aes()` for that.

But **mapping** data to specific color values is still not as simple as you might initially think. For example, mapping `Color` to `color` inside of `aes()` produces this stroop test.

```
ggplot(I_subset, aes(Dur_msec, F1, color = Color))+
  geom_point()
```

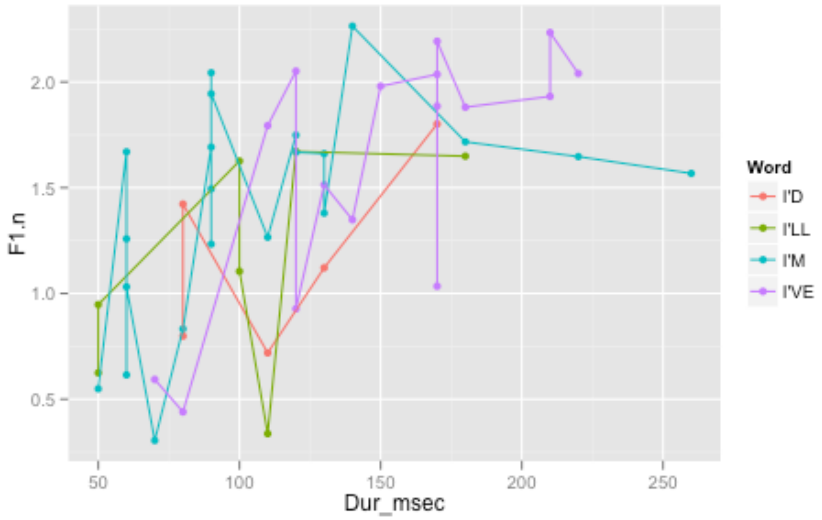


The lesson here is *not* to try this way of constructing your own custom color palettes. We'll go over how to construct custom palettes under Scales.

Inheritance

If we add one more geometry (a line), we see that it also inherits the mapping of `word` to `color`.

```
ggplot(I_subset, aes(Dur_msec, F1.n, color = Word))+
  geom_point()+
  geom_line()
```

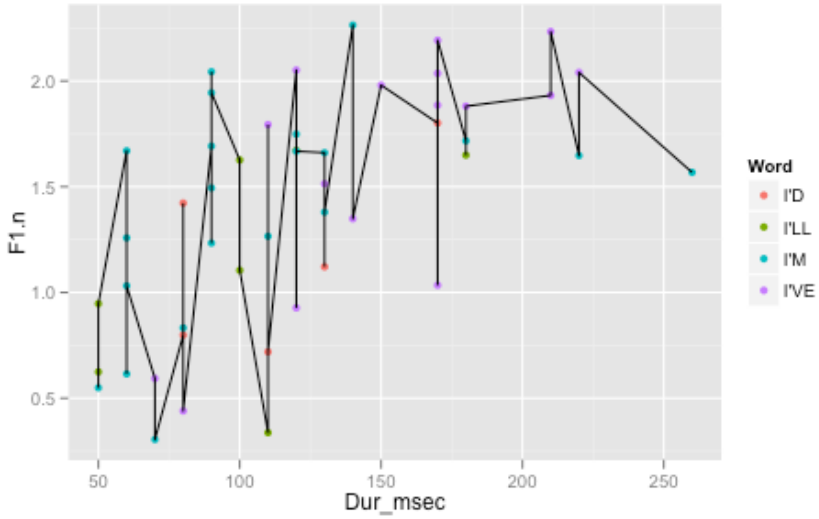


There are a few important things to take note of in this plot. First, you can see that we have actually added four lines to the plot, one for each color. In most cases, when you map categorical data to an aesthetic like `color`, you are also defining sub-groupings of the data, and `ggplot2` will draw a lines, calculate statistics, etc. separately for every sub-grouping of the data.

The second important thing to notice is that `geom_line()` joins up points as they are ordered along the x-axis, *not* according to their order in the original data frame. There is a geom which will join up points that way called `geom_path()`.

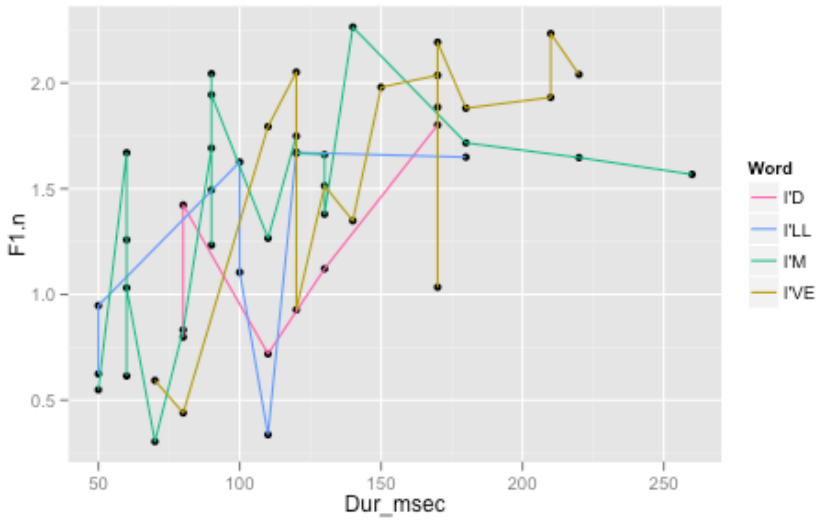
They point here, though, is that it is possible to define data-to-aesthetic mappings inside of geom functions, also by using `aes()`. Here, instead of mapping `Word` to `color` inside of `ggplot()`, we'll do it inside of `geom_point()`.

```
ggplot(I_subset, aes(Dur_msec, F1.n))+
  geom_point(aes(color = Word))+
  geom_line()
```



The points are still colored according to the word, but there is only one, black line. We can also try passing `aes(color = word)` to `geom_line()`.

```
ggplot(I_subset, aes(Dur_msec, F1.n))+
  geom_point()+
  geom_line( aes(color = Word))+
  scale_color_hue(direction = -1)
```



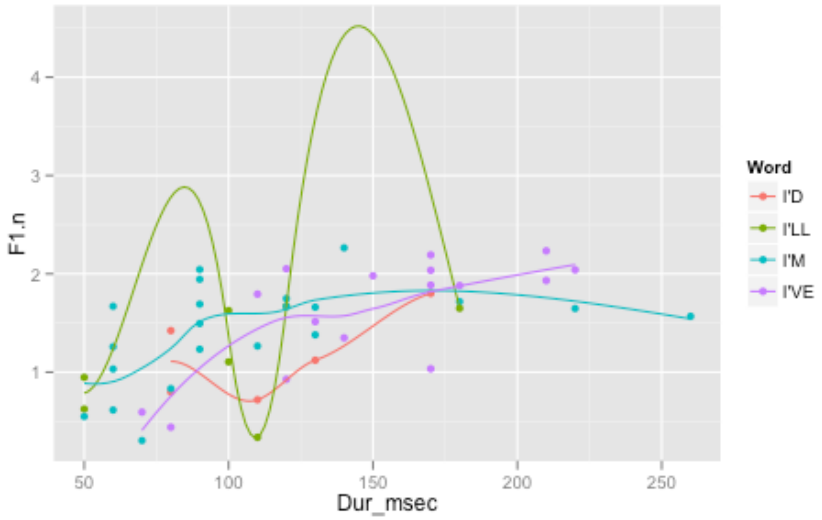
Now, the lines are colored according to the word, but the points are all black. This brings up the all important point about aesthetics:

Geoms inherit aesthetic mappings from the `ggplot()` data layer, and not from any other layer.

Grouping

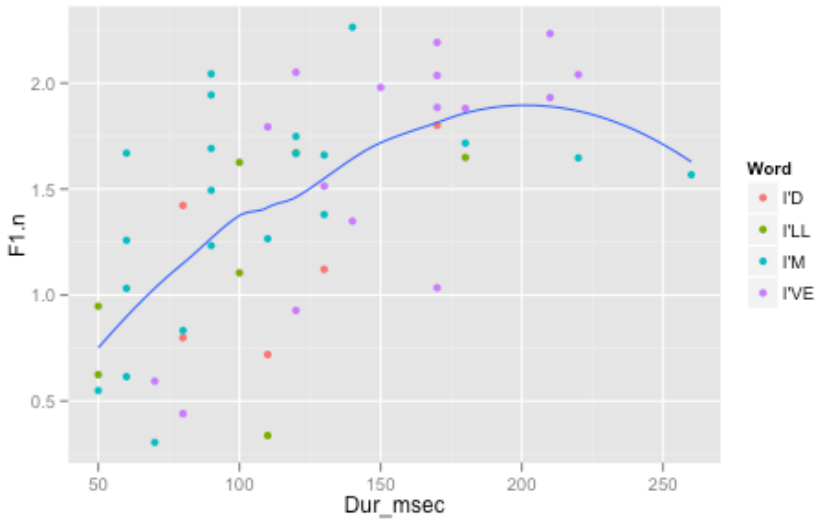
Let's look at the effect of mapping `Word` to `color` on the calculation of statistics, like smoothing lines. Note, inside of `stat_smooth()` I've said `se = F` to turn off the display of standard errors.

```
ggplot(I_subset, aes(Dur_msec, F1.n, color=Word))+
  geom_point()+
  stat_smooth(se = F)
```



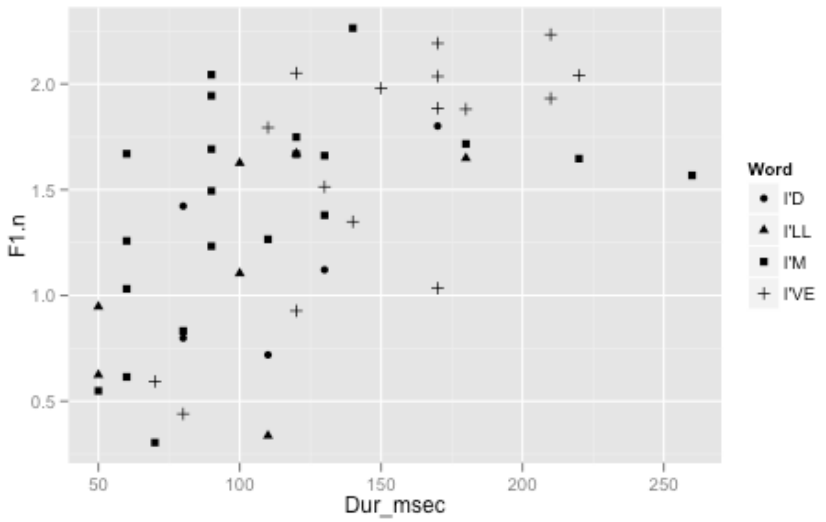
Just like separate lines were drawn for each group as defined by `color=Word`, `ggplot2` has calculated separate smoothers for each subset. If we had only passed `color=Word` to `geom_point()`, though, `stat_smooth()` would not have inherited that mapping, resulting in a single smoother being calculated.

```
ggplot(I_subset, aes(Dur_msec, F1.n))+
  geom_point(aes(color=Word))+
  stat_smooth(se = F)
```



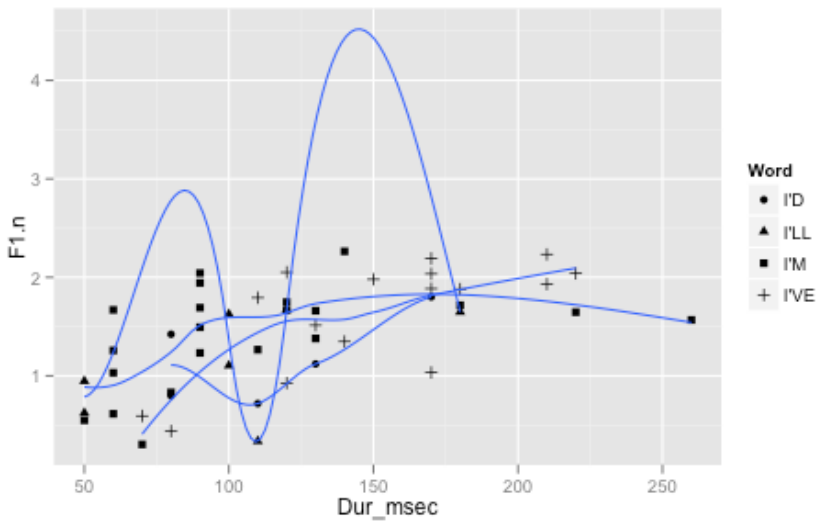
It's important to understand that when you map categorical variables to an aesthetic that you're also defining sub-groupings. For example, if we map `word` to `shape`, instead of `color`, the point shapes will now represent the word.

```
ggplot(I_subset, aes(Dur_msec, F1.n, shape=Word))+
  geom_point()
```



Now if we add a smoother to this plot, *even though* `shape` *isn't defined for lines*, the smoother will still plot a different smoothing curve for each sub-grouping.

```
ggplot(I_subset, aes(Dur_msec, F1.n, shape=Word))+
  geom_point()+
  stat_smooth(se = F)
```

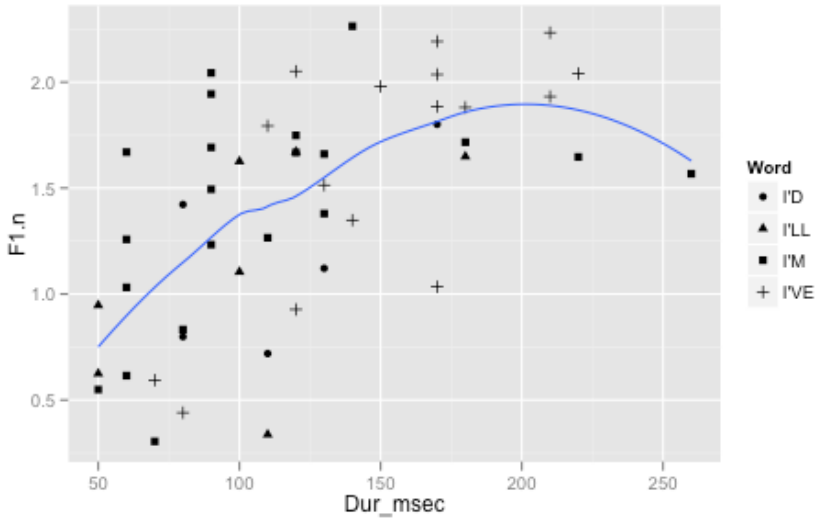


If you really only wanted a single smoother line for all of the data in this case, one solution would be to move the `shape=word` mapping from the data layer to the `geom_point()` layer. But in most cases, it's actually

more desirable to override the aesthetic mapping. We can do this with the special aesthetic `group`.

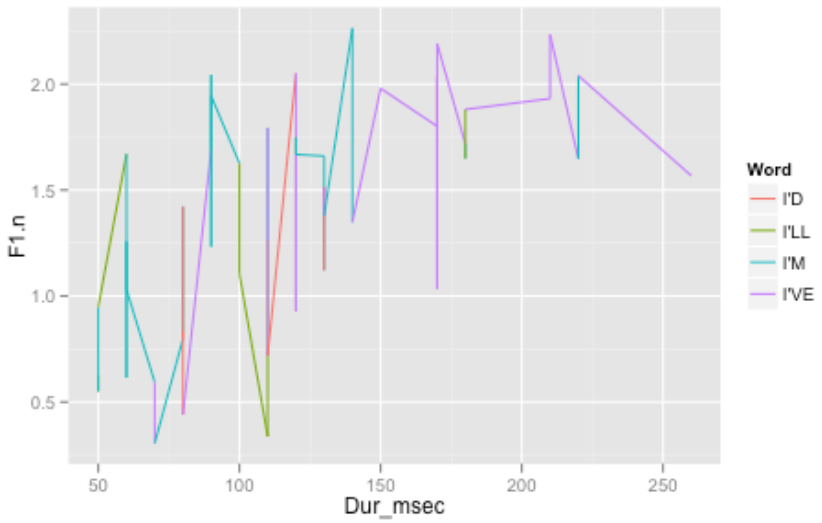
`group` does exactly what it sounds like it ought to: it defines groups of data. When you want to override groups defined in the data layer, you can do so by saying `group=1`.

```
ggplot(I_subset, aes(Dur_msec, F1.n, shape=Word))+
  geom_point()+
  stat_smooth(se = F, aes(group = 1))
```



The effect it has on `stat_smooth()` is that just a single smoother is calculated. If we come back to `color = word`, and then draw a line with `group = 1`, the effect is that we draw one line that varies in color.

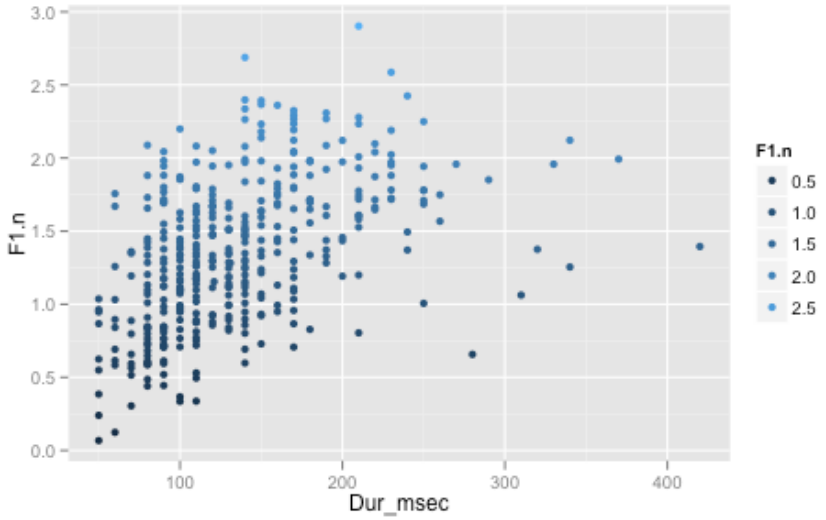
```
ggplot(I_subset, aes(Dur_msec, F1.n, color=Word))+
  geom_line(aes(group = 1))
```



More aesthetics and their use.

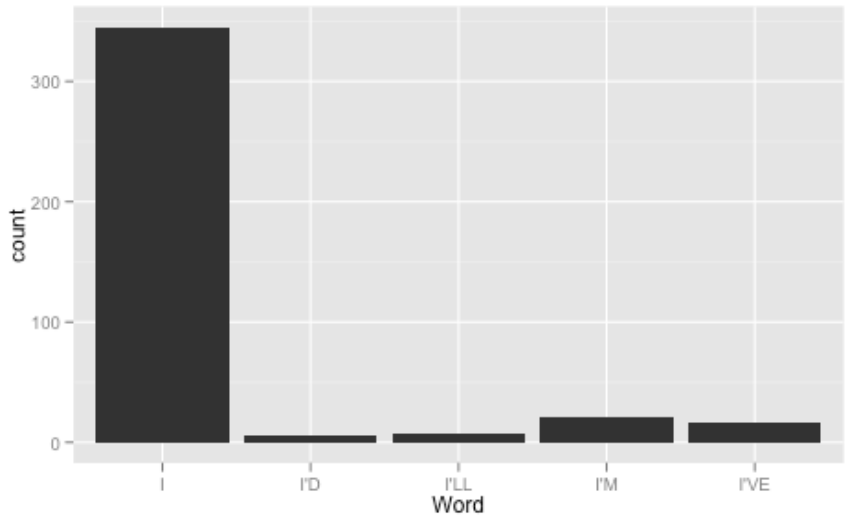
So far, we've only mapped categorical variables to `color`, but it's also possible to map continuous variables to `color`. Here we'll redundantly map `F1.n` to both y and `color`.

```
ggplot(I_jean, aes(Dur_msec, F1.n, color = F1.n))+
  geom_point()
```



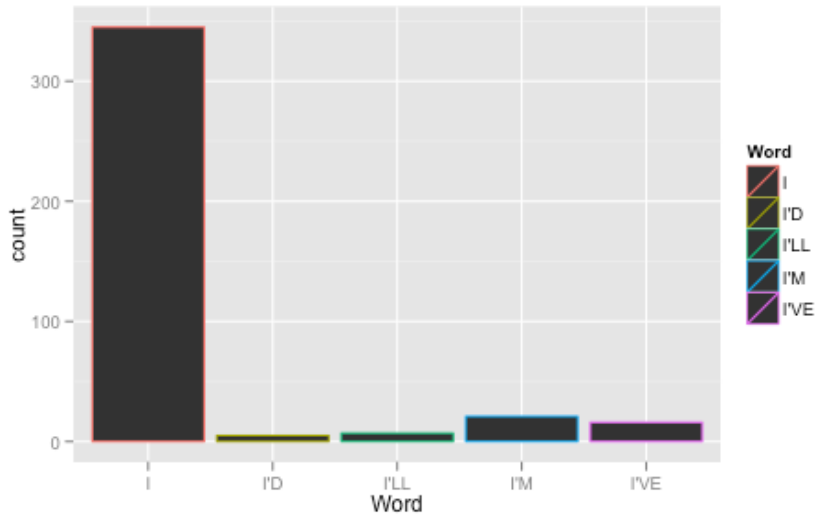
Another important aesthetics distinction is between `color` and `fill`. If we wanted to create a bar chart of word frequencies, we could do so by mapping `word` to the x-axis, and adding `geom_bar()` without any y-axis variable defined.

```
ggplot(I_jean, aes(Word))+
  geom_bar()
```



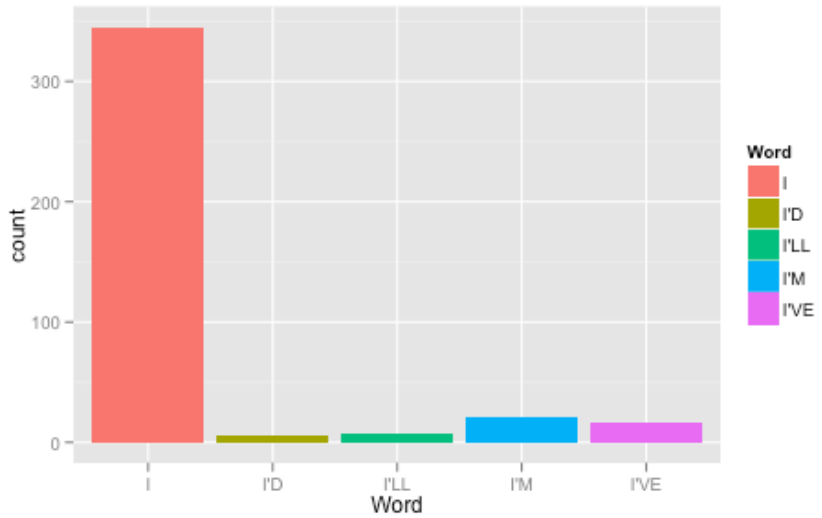
If you also wanted to color the bars according to the word, your first instinct would probably be to map `color = word`. But the result is that only the colors of the bars' outlines are mapped to `word`.

```
ggplot(I_jean, aes(Word, color = Word))+
  geom_bar()
```



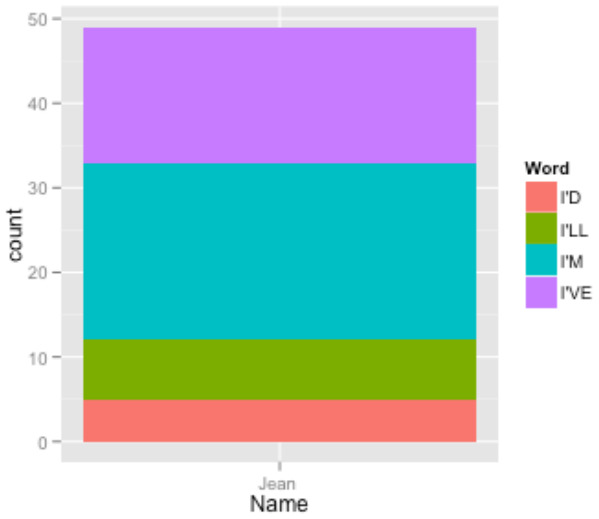
What is probably more advisable is to map `word` to `fill`, which control the filling color of two dimensional geoms.

```
ggplot(I_jean, aes(Word, fill = Word))+
  geom_bar()
```



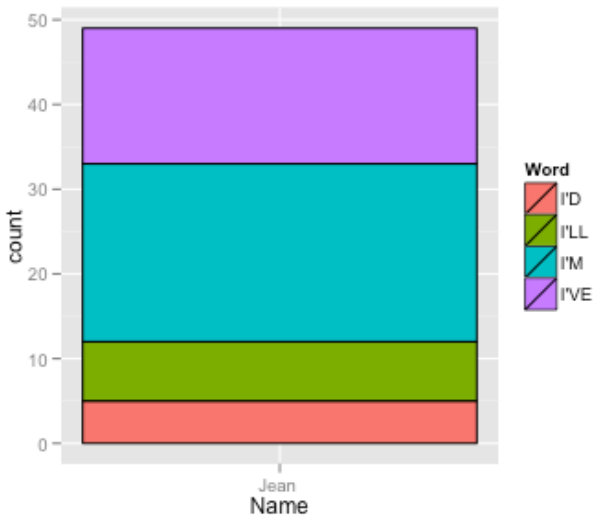
As you might have figured out now, it's technically possible to map the `fill` color of bars to one variable, and the outline `color` to different variable. My advice is to never do such a thing, because the results almost always come out a jumbled mess. Instead, I would suggest setting the `color` of the bars to black. I find it more pleasing to the eye, and helps to emphasize the divisions between bars when they're stacked. Compare this plot:

```
ggplot(I_subset, aes(Name, fill = Word))+
  geom_bar()
```



to this one.

```
ggplot(I_subset, aes(Name, fill = Word))+
  geom_bar(color = "black")
```



Geometries

So far, we've used the following geometries:

- `geom_point()`
- `geom_line()`
- `geom_bar()`

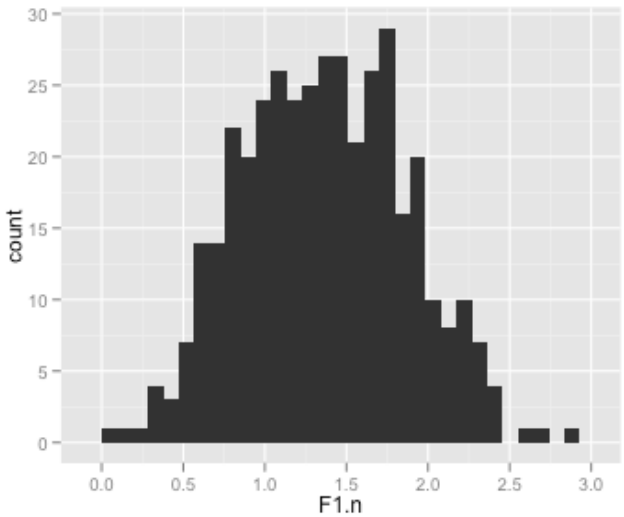
All geometries begin with `geom_`, meaning you can get a full list using `apropos()`.

```
apropos("^geom_")
```

```
##  [1] "geom_abline"     "geom_area"       "geom_bar"
##  [4] "geom_bin2d"      "geom_blank"      "geom_boxplot"
##  [7] "geom_contour"    "geom_crossbar"   "geom_density"
## [10] "geom_density2d"  "geom_dotplot"    "geom_errorbar"
## [13] "geom_errorbarh"  "geom_freqpoly"   "geom_hex"
## [16] "geom_histogram"  "geom_hline"      "geom_jitter"
## [19] "geom_line"       "geom_linerange"  "geom_map"
## [22] "geom_path"       "geom_point"      "geom_pointrange"
## [25] "geom_polygon"    "geom_quantile"   "geom_raster"
## [28] "geom_rect"       "geom_ribbon"     "geom_rug"
## [31] "geom_segment"    "geom_smooth"     "geom_step"
## [34] "geom_text"       "geom_tile"       "geom_violin"
## [37] "geom_vline"
```

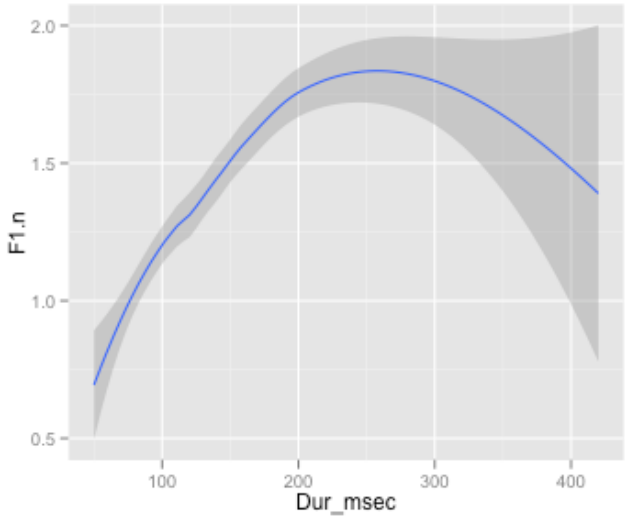
This is a quite extensive list, and we won't be able to cover them all today. Many of them are actually convenience functions for special settings of other geoms. For example, `geom_histogram()` is really just `geom_bar()` with special settings.

```
ggplot(I_jean, aes(F1.n))+
  geom_histogram()
```



Other geoms are just convenience functions for statistical layers. For example, you'll notice `geom_smooth()`, which if you add it to a plot will have the same behavior of `stat_smooth()`, which we've already been using extensively.

```
ggplot(I_jean, aes(Dur_msec, F1.n))+
  geom_smooth()
```



Some special geoms

Some geoms are both unique and common enough in their usage to warrant special mention.

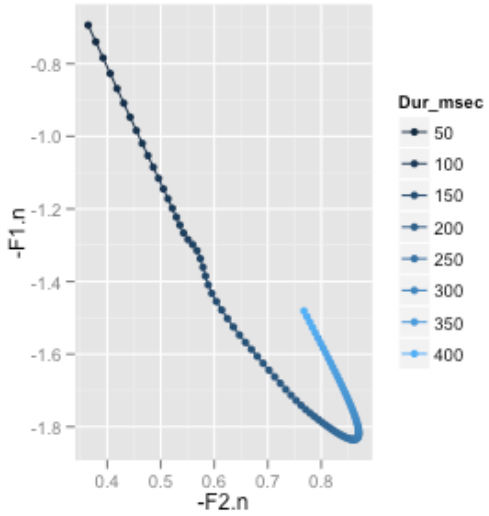
`geom_line()` **vs** `geom_path()`

As I said above, when you add `geom_line()` to a plot, it connects points up according to their order along the x-axis. If you want to connect points according to their order in a data frame (say, to illustrate a trajectory through two-dimensional space over time), you should use `geom_path()`.

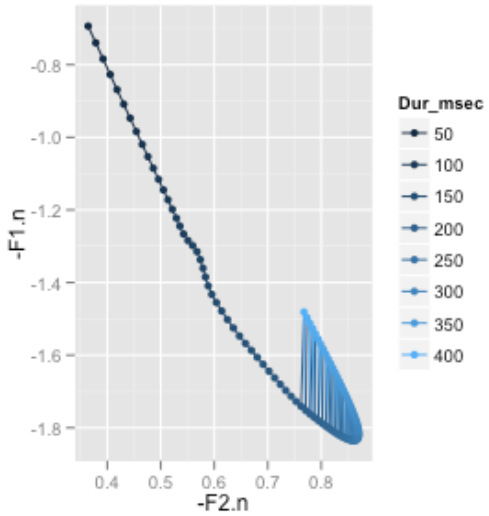
```
mod_F1 <- loess(F1.n ~ Dur_msec, data = I_jean)
mod_F2 <- loess(F2.n ~ Dur_msec, data = I_jean)

pred <- data.frame(Dur_msec = seq(50, 400, length = 100))
pred$F1.n <- predict(mod_F1, newdata = pred)
pred$F2.n <- predict(mod_F2, newdata = pred)
```

```
ggplot(pred, aes(-F2.n, -F1.n, color = Dur_msec))+
  geom_path()+
  geom_point()
```



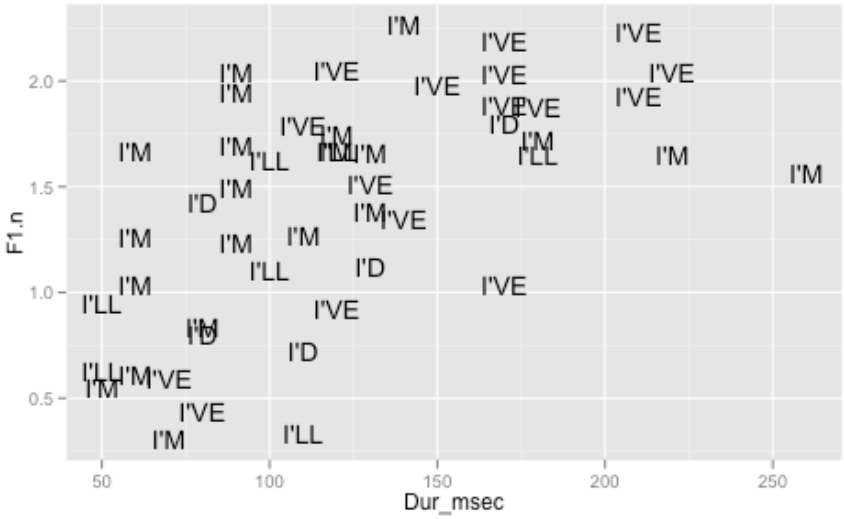

```
ggplot(pred, aes(-F2.n, -F1.n, color = Dur_msec))+
  geom_line()+
  geom_point()
```



`geom_text()`

Adding text, and text labels to a plot, is a very common task, and is done with `geom_text()`. There is a special aesthetic just for `geom_text()` called `label`, which defines the column that should be used as the text label.

```
ggplot(I_subset, aes(Dur_msec, F1.n))+
  geom_text(aes(label = Word))
```



Positioning

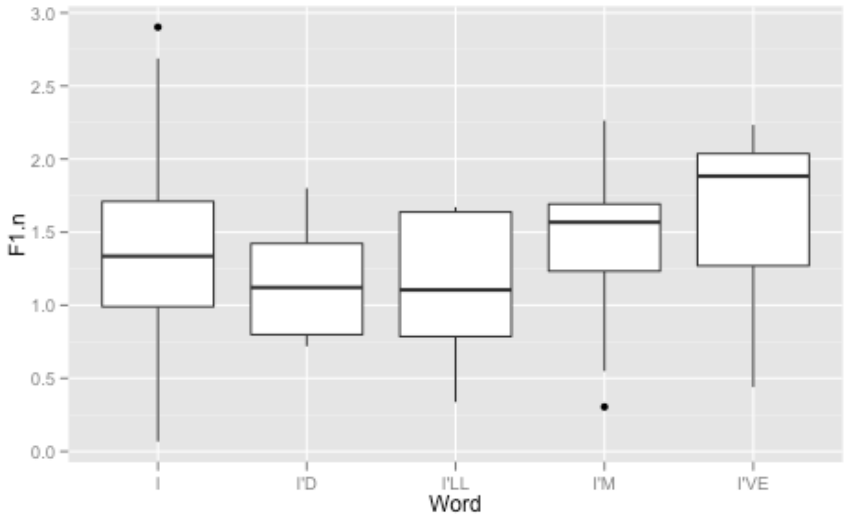
Just like inheritance was the big idea for aesthetics, positioning is the big idea for geoms. For various reasons, you may want to adjust where geometries are plotted. As a solution to overplotting, for example, you may want to add some jitter to points. When dealing with bars, you need to decide whether they should be stacked, or arranged next to each other. These small adjustments

- identity
  - This is the default in most cases, simply plotting geometries where they're defined by x and y.
- jitter
  - This adds some random noise either to the x position or the the y position, and is typically used just for points.
- stack
  - This stacks geometries on top of each other. This is the default for bars
- dodge
  - This pushes geometries out of each other's way, to the left and right.
- fill
  - This stacks geometries on top of each other, and expands or contracts them to fill the space between 0 and 1. Good for plotting proportions.

Jitter

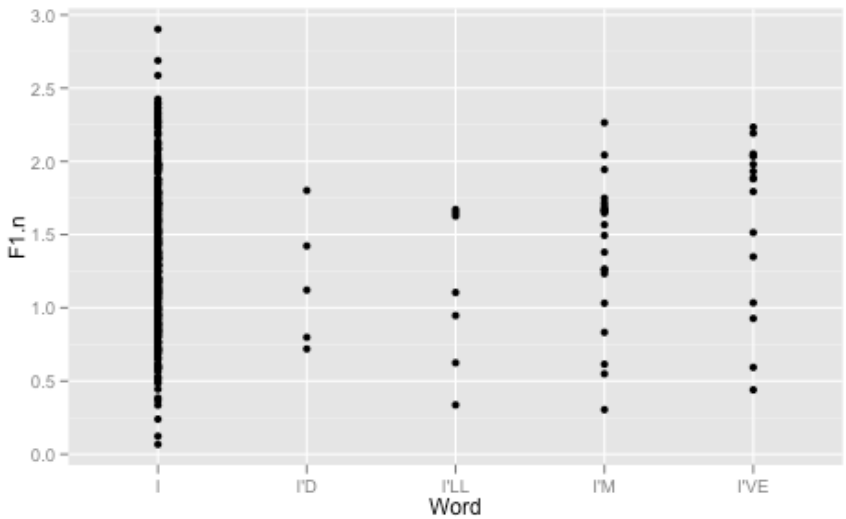
Some people, like Andrew Gelman, hate boxplots.

```
ggplot(I_jean, aes(Word, F1.n))+
  geom_boxplot()
```



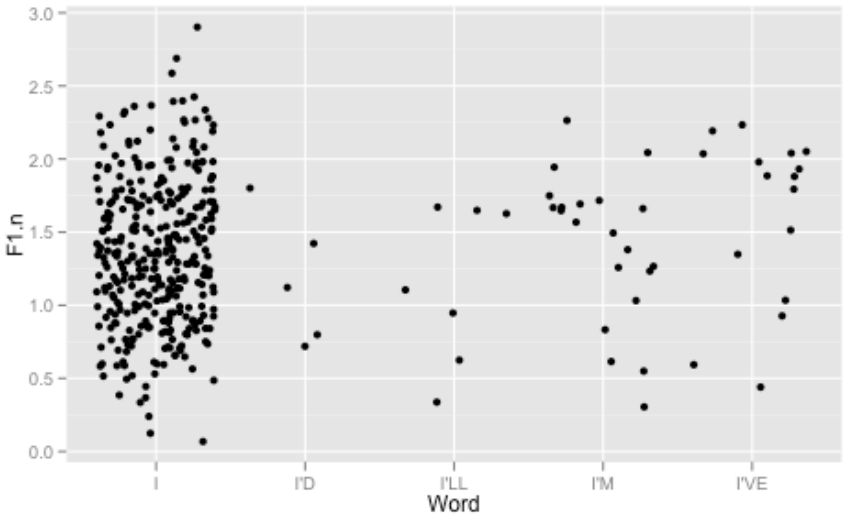
A frequent suggestion for a replacement to boxplots is just to plot the raw data points with some jitter. To get started, we'll replace `geom_boxplot()` with `geom_point()`.

```
ggplot(I_jean, aes(Word, F1.n))+
  geom_point()
```



And the add some jitter, by defining `position = "jitter"` inside of `geom_point()`.

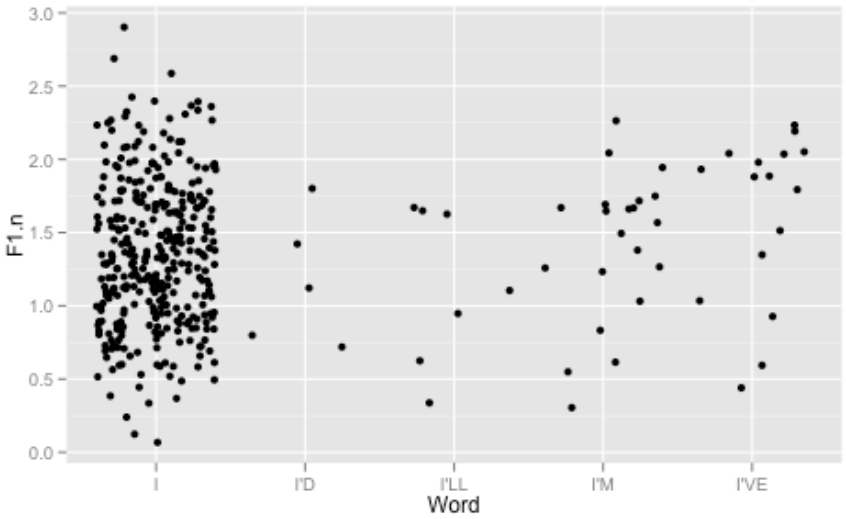
```
ggplot(I_jean, aes(Word, F1.n))+
  geom_point(position = "jitter")
```



In this example, you can see the benefit of jittered points over boxplots. With boxplots, there's no hint that one category, “I” has enormously more data than the others.

As a convenience, there's a geom called `geom_jitter()`, which is just a convenience function for `geom_point(position = "jitter")`.

```
ggplot(I_jean, aes(Word, F1.n))+
  geom_jitter()
```



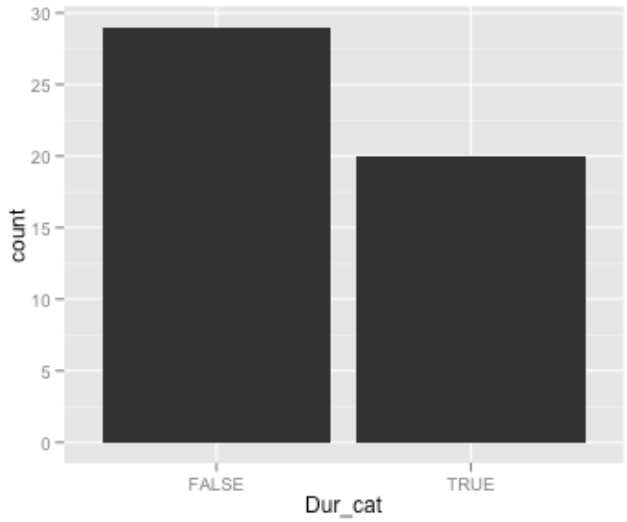
Dodge, Stack and Fill

To demonstrate how dodge, stack and fill work, let's create an additional categorical variable. `Dur_cat` will equal `TRUE` if the duration is greater than the mean, and `FALSE` otherwise.

```
I_subset$Dur_cat <- I_subset$Dur_msec > mean(I_subset$Dur_msec)
```

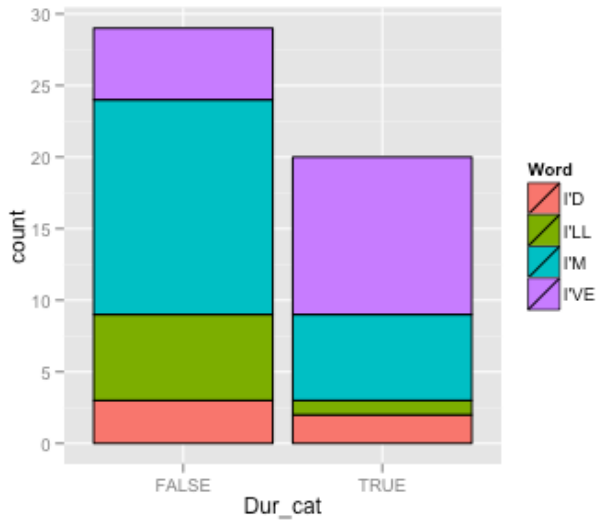
As a first step, we can create a bar plot where the height of the bars is equal to the number of observations in each category.

```
ggplot(I_subset, aes(Dur_cat))+
  geom_bar()
```



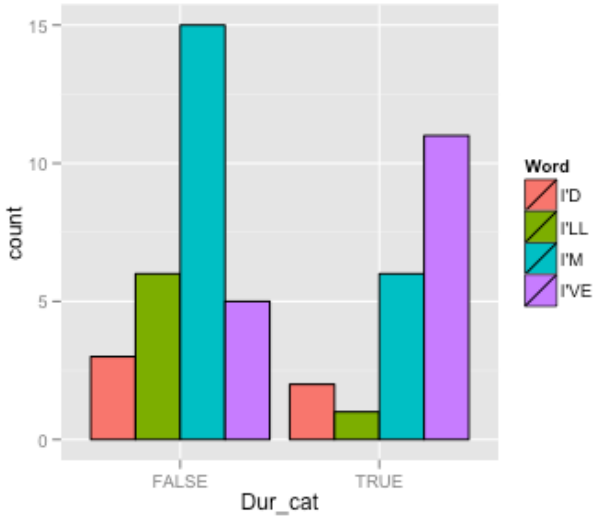
Next, we'll fill in the bars according to `Word`.

```
ggplot(I_subset, aes(Dur_cat, fill = Word))+
  geom_bar(color = "black")
```



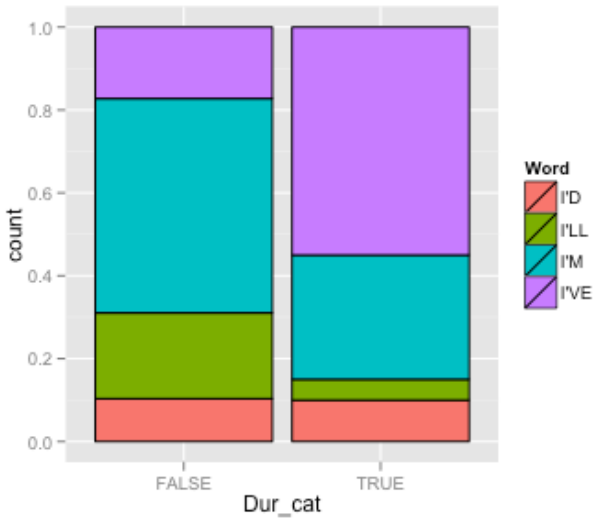
Here, we're seeing the default behavior of stacking bars on top of each other. But perhaps we'd like to see each colored segment lined up next to the others in a grouped bar chart. We can do this by defining `position = "dodge"`.

```
ggplot(I_subset, aes(Dur_cat, fill = Word))+
  geom_bar(position = "dodge", color = "black")
```



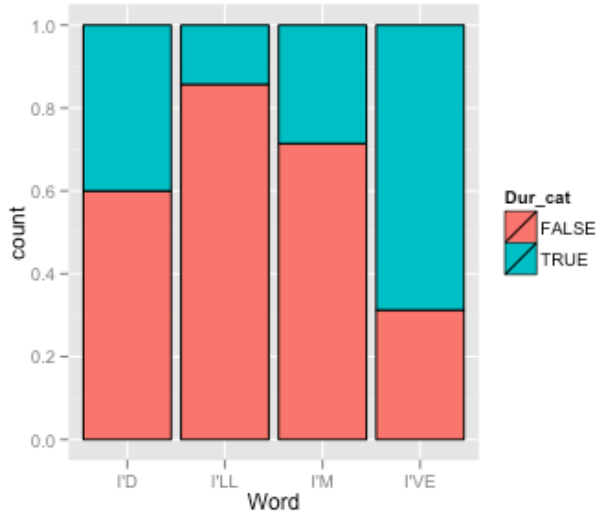
Or, perhaps we'd like to scale the stacked bars to be between 0 and 1, so we can more clearly see the proportional distribution of words into long and short categories. We can do this with `position = "fill"`.

```
ggplot(I_subset, aes(Dur_cat, fill = Word))+
  geom_bar(position = "fill", color = "black")
```



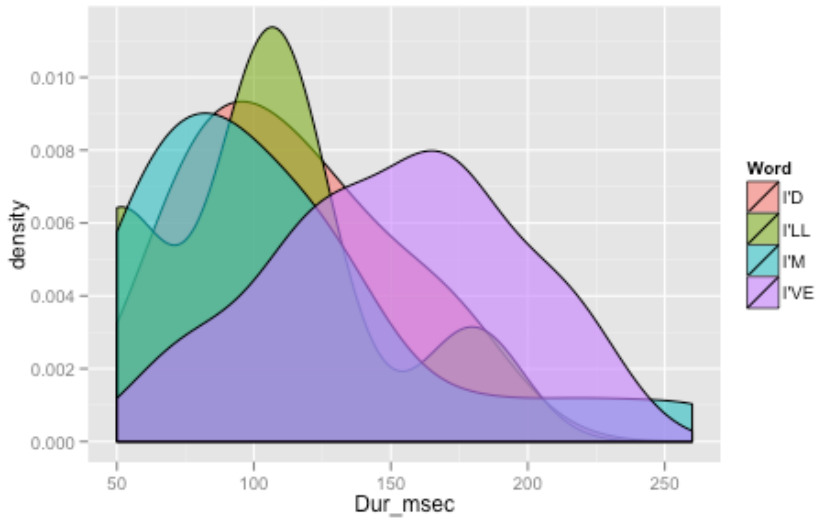
Or, depending on what kind of information you want to show:

```
ggplot(I_subset, aes(Word, fill = Dur_cat))+
  geom_bar(position = "fill", color = "black")
```



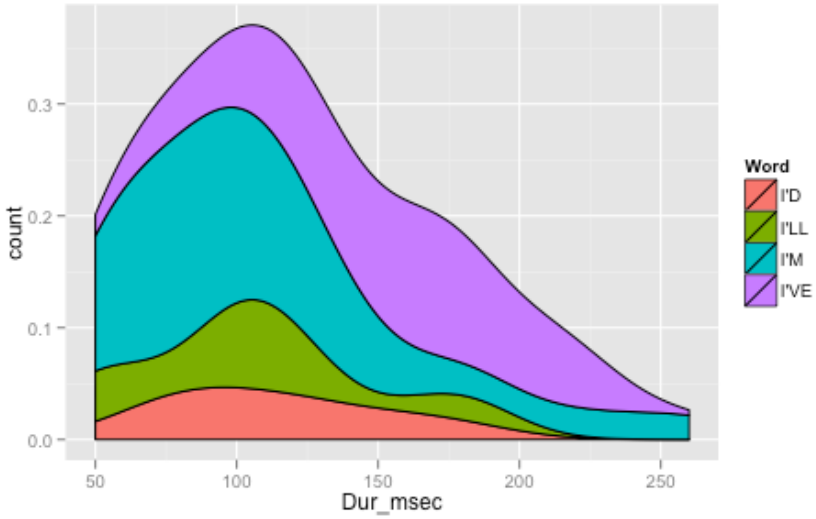
Stacking and filling are not just only applicable to bars. Things can get really interesting if you apply them to density plots. For example, here are kernel density estimates for`Dur_msec` for each word.

```
ggplot(I_subset, aes(Dur_msec, fill = Word))+
  geom_density(alpha = 0.6)
```



It's a bit of a mishmash, but it gets a lot more interesting if you stack them. The bit about `y = ..count..` will make more sense when we go over statistics.

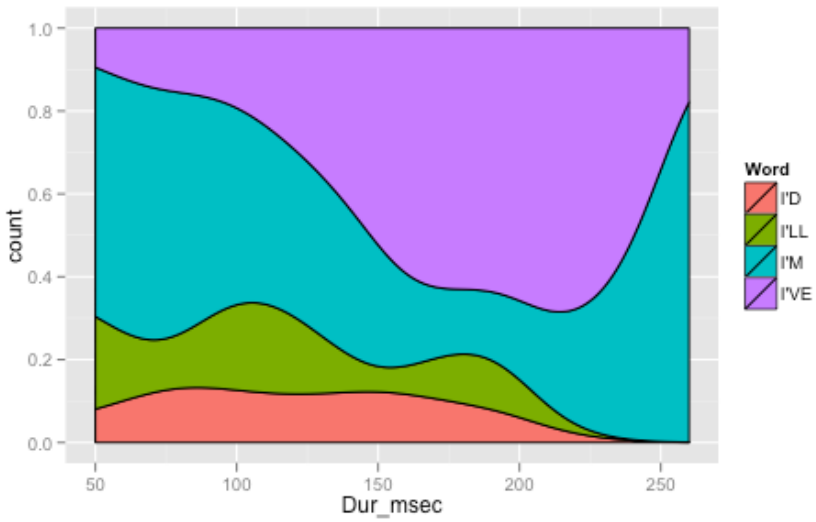
```
ggplot(I_subset, aes(Dur_msec, fill = Word))+
  geom_density(position = "stack", aes(y = ..count..))
```



Now we have a stacked density graph. The over-all height of the density displays the density of all data points, and the width of each band represents the portion of that density which is taken up by each word in that region. It's very similar to this illustration of global income distributions

Alternatively, we could set `position = "fill"` to display what proportion of observations at each duration are from which words.

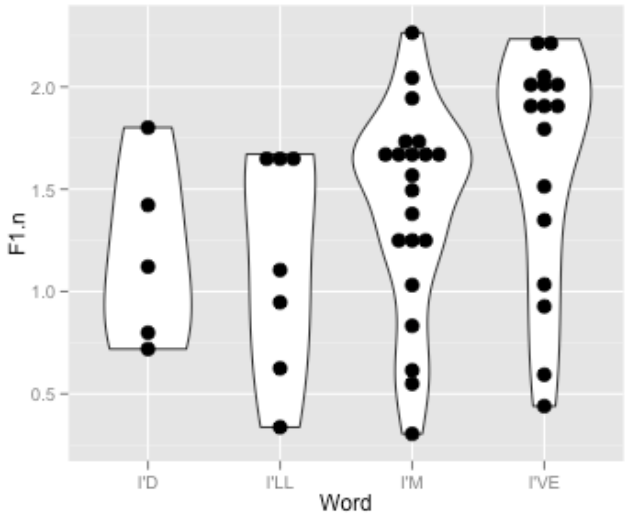
```
ggplot(I_subset, aes(Dur_msec, fill = Word))+
  geom_density(position = "fill", aes(y = ..count..))
```



Some new and exciting geoms

As of version 0.9.0 of `ggplot2`, there are some new and exciting geoms available, like `geom_dotplot()` and `geom_violin()`. Here's a combination of both `geom_violin()` and `geom_dotplot()` that creates a variant of a bean plot.

```
ggplot(I_subset, aes(Word, F1.n))+
  geom_violin()+
  geom_dotplot(binaxis="y", stackdir="center")
```



Statistics

After the layering paradigm, and the inheritance of aesthetic mappings, the next most powerful aspect of `ggplot2` is the ease with which statistical summaries can be incorporated into plots. Every statistical layer you can add to a plot starts with `stat_`, meaning we can list all statistical layers with `apropos()`.

```
apropos("^stat_")
```

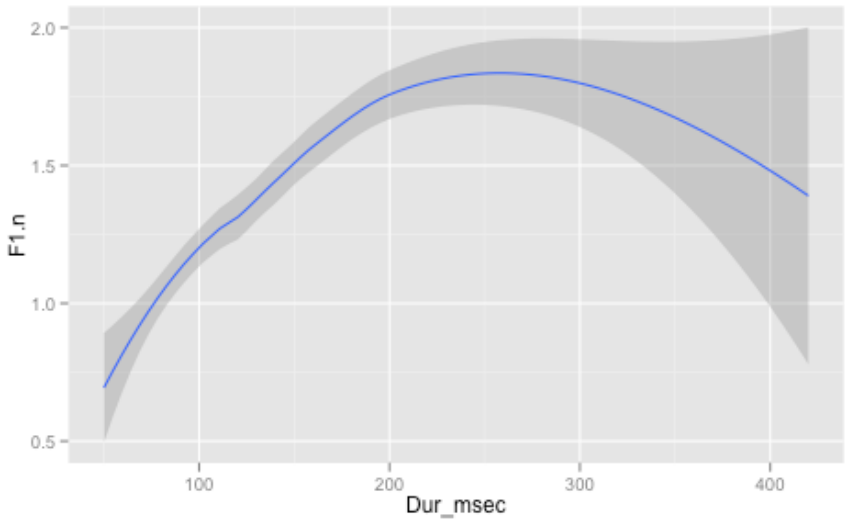
```
##  [1] "stat_abline"      "stat_bin"
##  [3] "stat_bin2d"       "stat_bindot"
##  [5] "stat_binhex"      "stat_boxplot"
##  [7] "stat_contour"     "stat_density"
##  [9] "stat_density2d"   "stat_function"
## [11] "stat_hline"       "stat_identity"
## [13] "stat_qq"          "stat_quantile"
## [15] "stat_smooth"      "stat_spoke"
## [17] "stat_sum"         "stat_summary"
## [19] "stat_summary2d"   "stat_summary_hex"
## [21] "stat_unique"      "stat_vline"
## [23] "stat_ydensity"
```

There are fewer statistical layers than geometries, but still too many to cover in their entirety today.

Statistics and their Geometries

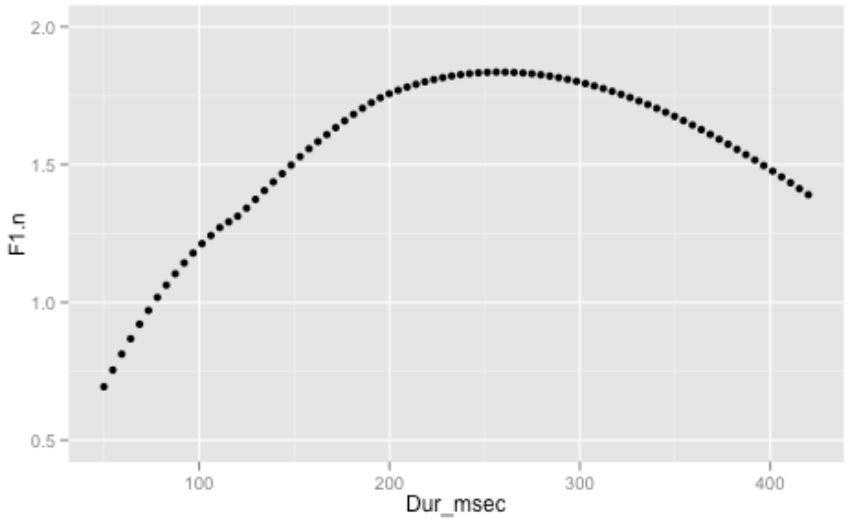
When you add a statistic to a plot, it will automatically plot with a default geom. For example, the default geom for `stat_smooth()` is `geom_smooth()`.

```
ggplot(I_jean, aes(Dur_msec, F1.n)) +
  stat_smooth()
```



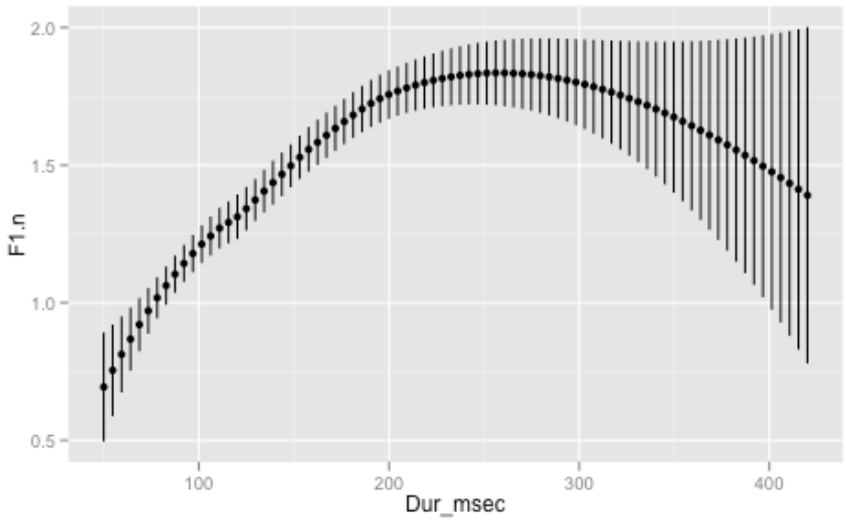
But there is nothing intrinsic to `stat_smooth()` requiring it to be represented this way. We can specify other geoms `stat_smooth()` to use, like points.

```
ggplot(I_jean, aes(Dur_msec, F1.n)) +
  stat_smooth(geom = "point")
```



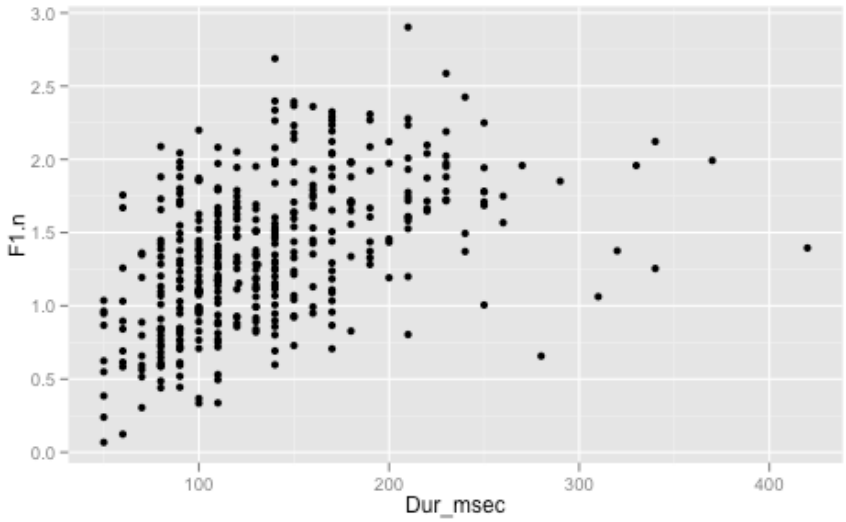
Or point ranges.

```
ggplot(I_jean, aes(Dur_msec, F1.n)) +
  stat_smooth(geom = "pointrange")
```

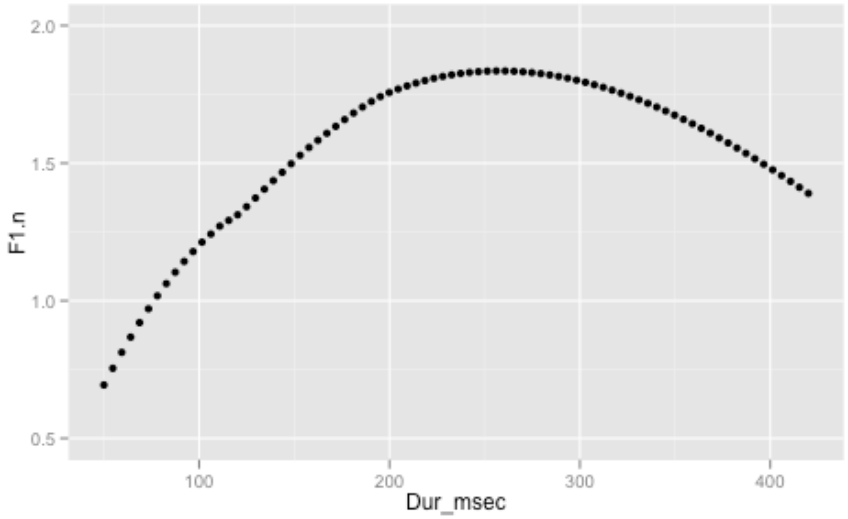


Coming from the other direction, you can specify special statistics for geometries.

```
ggplot(I_jean, aes(Dur_msec, F1.n)) +
  geom_point()
```




```
ggplot(I_jean, aes(Dur_msec, F1.n)) +
  geom_point(stat = "smooth")
```



They key point here is: **Every geometry displays a statistic, even if that's just the `identity` statistic. All geometries have a default statistic, which can be overrode. All statistics are displayed with a geometry. All statistics have a default geometry, which can be overrode.**

This intimate interrelationship between statistics and geoms is why there are so many statistics and geoms with the same name.

```
geoms <- gsub("geom_", "", apropos("^geom_"))
stats <- gsub("stat_", "", apropos("^stat_"))

stats[stats %in% geoms]
```

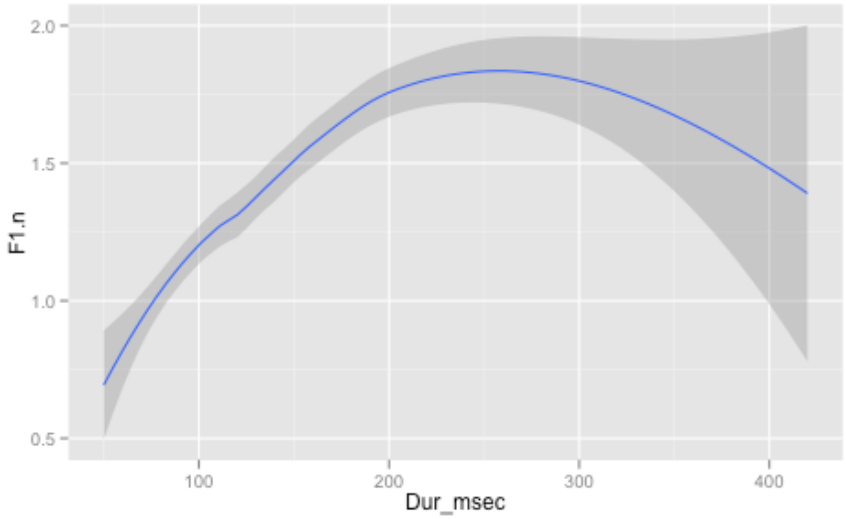
```
##  [1] "abline"    "bin2d"     "boxplot"   "contour"
##  [5] "density"   "density2d" "hline"     "quantile"
##  [9] "smooth"    "vline"
```

Particularly useful Statistics

`stat_smooth()`

The most common statistic to add to a plot is `stat_smooth()`. The default behavior for `stat_smooth()` is sensitive to how many data points there are in a plot. The plot below contains < 1000 points, meaning that its plotting a loess smooth. For points > 1000, it'll fit a generalized additive model with penalized cubic regression splines.

```
ggplot(I_jean, aes(Dur_msec, F1.n)) +
  stat_smooth()
```

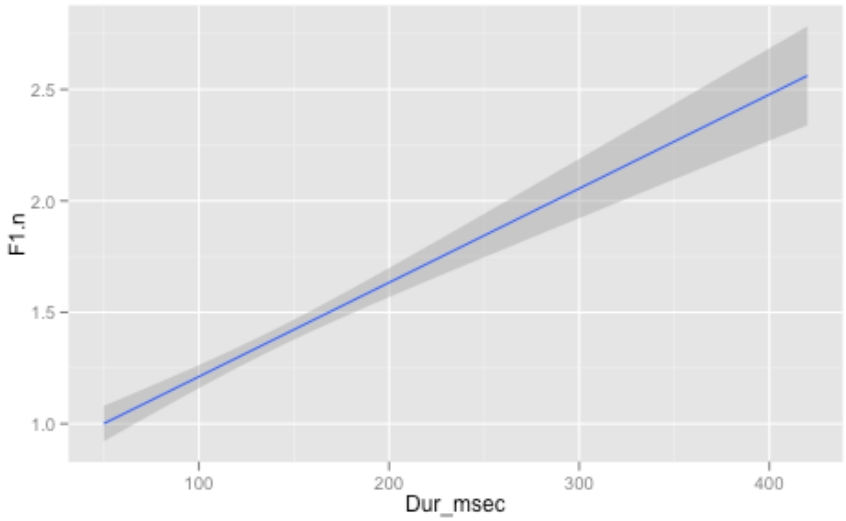


We can change the method of smoothing manually by specifying a regression function. Any regression method which

1. utilizes the formula specification, and
2. has a `predict()` method

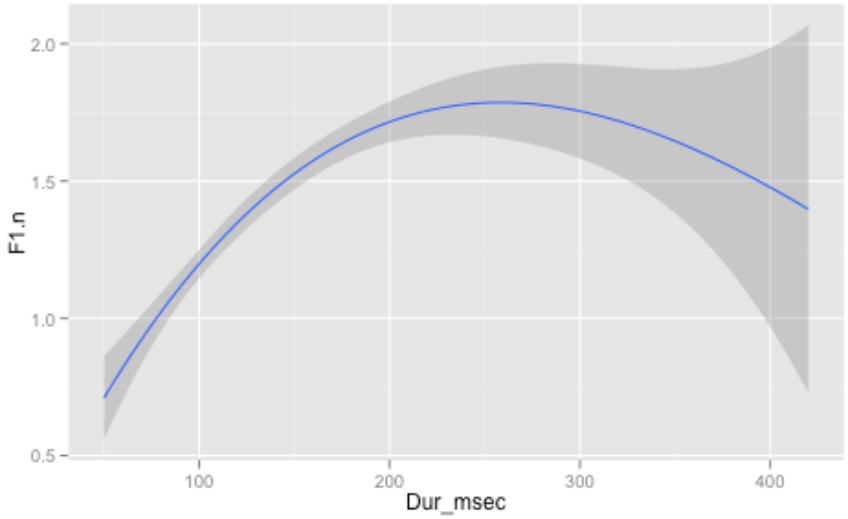
can be used with `stat_smooth()`. For example, we could plot an ordinary linear regression by saying `method = lm`.

```
ggplot(I_jean, aes(Dur_msec, F1.n)) +
  stat_smooth(method = lm)
```



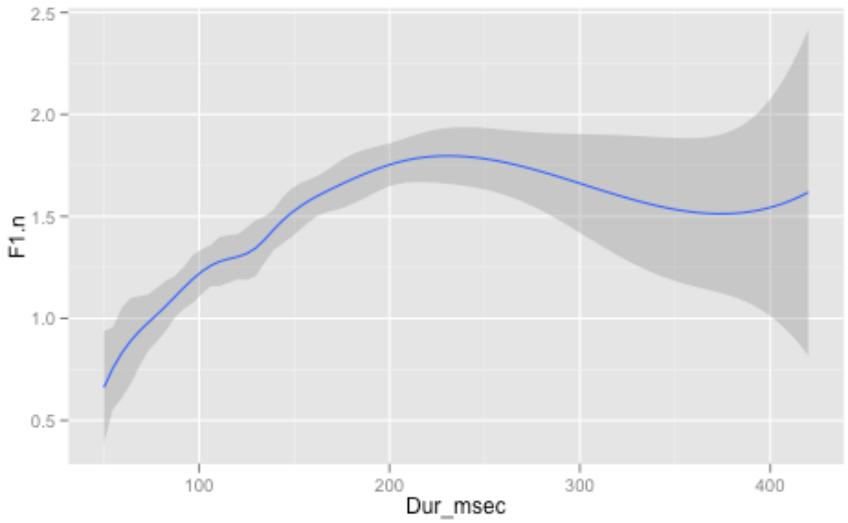
Any and all arguments that you might pass to `lm()` can be passed to `stat_smooth()`, including `formula`. So, for instance, if we wanted to fit a third-order polynomial curve to the data using `lm`, we would do so like this.

```
ggplot(I_jean, aes(Dur_msec, F1.n)) +
  stat_smooth(method = lm, formula = y ~ poly(x, 3))
```



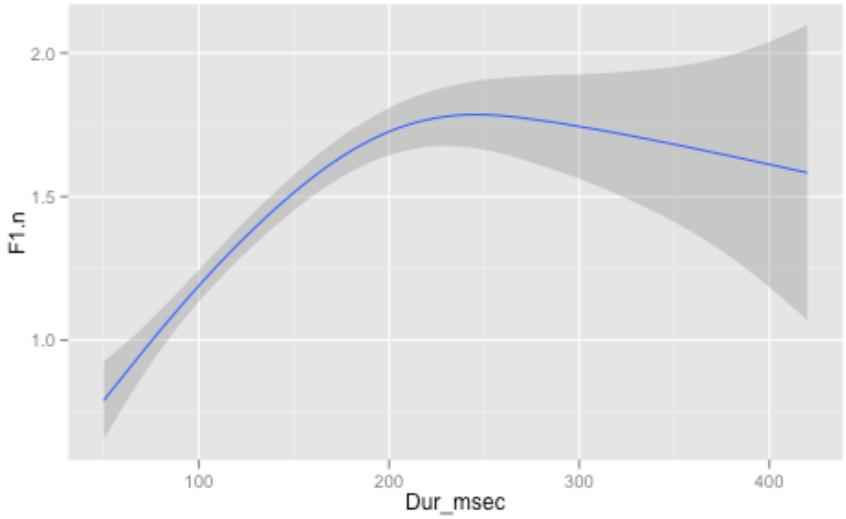
Here's how to fit b-splines with a relatively large basis for the amount of data.

```
library(splines)
ggplot(I_jean, aes(Dur_msec, F1.n)) +
  stat_smooth(method = lm, formula = y ~ bs(x, df = 10))
```



And a generalized additive model fit with a cubic regression spline.

```
library(mgcv)
ggplot(I_jean, aes(Dur_msec, F1.n)) +
  stat_smooth(method = gam, formula = y ~ s(x, bs = "cs"))
```



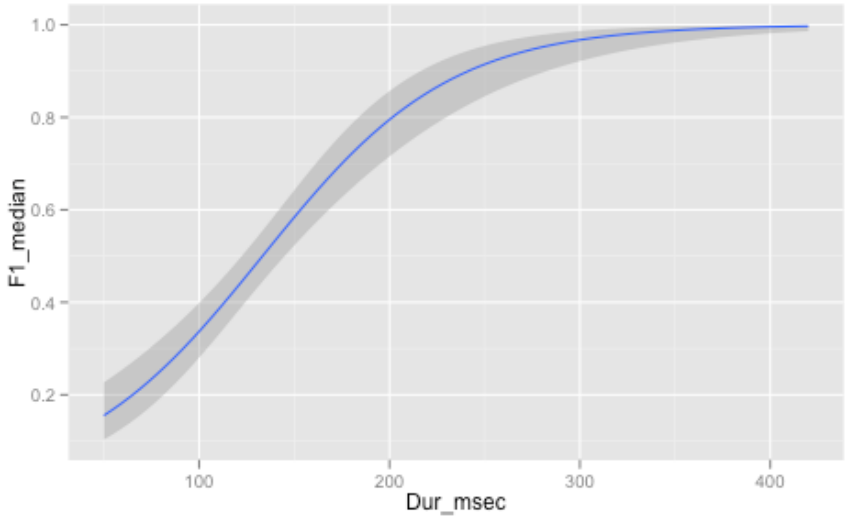
You can also include regression lines for other kinds of response variables. For example, let's create a binomial variable based on whether or not F1 is greater than the median F1.

```
I_jean$F1_median <- (I_jean$F1.n > median(I_jean$F1.n)) *
    1
head(I_jean)
```

```
##    Name Age Sex Word FolSegTrans Dur_msec     F1   F2
## 1 Jean   61   f  I'M           M      130  861.7 1336
## 2 Jean   61   f    I           N      140 1010.4 1349
## 3 Jean   61   f I'LL           L      110  670.1 1293
## 4 Jean   61   f  I'M           M      180  869.8 1307
## 5 Jean   61   f    I           R       80  743.0 1419
## 6 Jean   61   f I'VE           V      120  918.2 1581
##     F1.n    F2.n F1_median
## 1 1.6609 -0.8855         1
## 2 2.6883 -0.8536         1
## 3 0.3370 -0.9873         0
## 4 1.7168 -0.9536         1
## 5 0.8407 -0.6897         0
## 6 2.0512 -0.3068         1
```

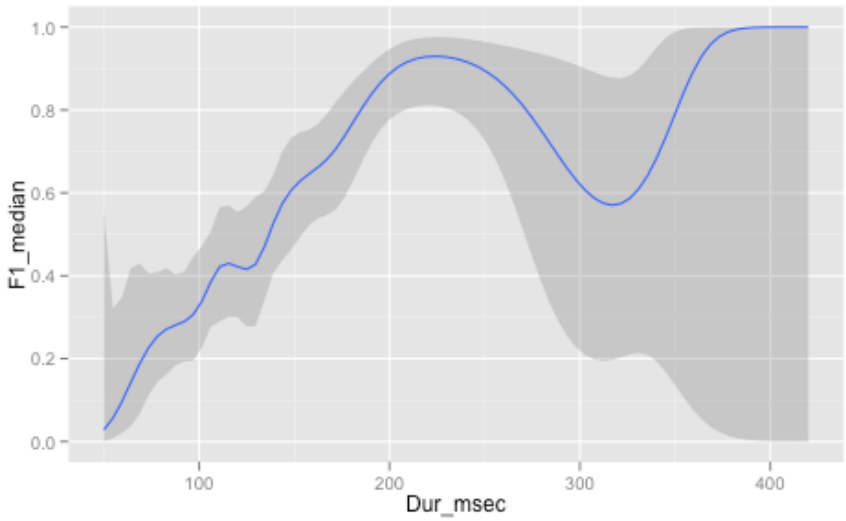
If we map `F1_median` to the y-axis, and then tell `stat_smooth()` to use `glm()` with a binomial family, we've got an instant logistic regression line.

```
ggplot(I_jean, aes(Dur_msec, F1_median))+
  stat_smooth(method = glm, family = binomial)
```

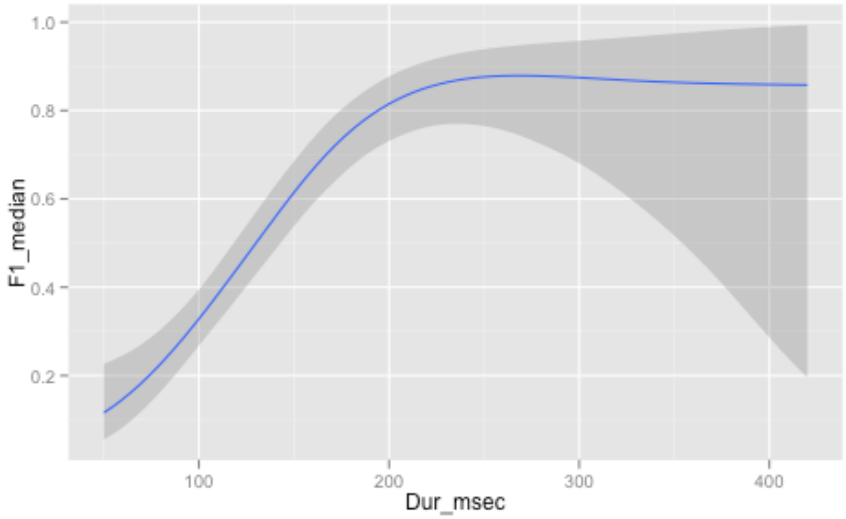


And just like we did for the continuous F1 variable, we can adjust the model formula for more complex regressions.

```
## b-splines
ggplot(I_jean, aes(Dur_msec, F1_median))+
  stat_smooth(method = glm,
              family = binomial,
              formula  = y ~ bs(x, df = 10))
```



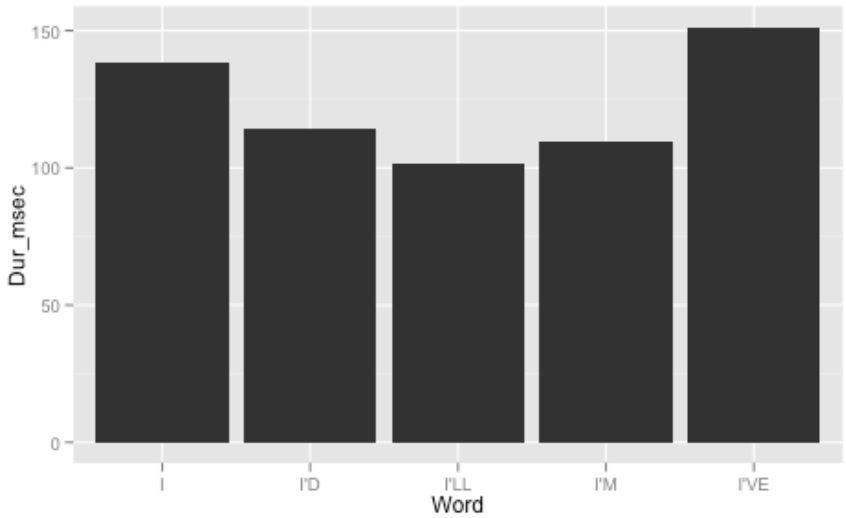

```
## cubic regression splines
ggplot(I_jean, aes(Dur_msec, F1_median))+
  stat_smooth(method = gam,
              family = binomial,
              formula  = y ~ s(x, bs = "cs"))
```



`stat_summary()`

Another very useful statistic is `stat_summary()`, which allows you to plot arbitrary summaries of your data. `stat_summary()` takes all of the data along the y-axis for each value along the x-axis, applies the summary function that you pass to `fun.y`, then plots that value with the specified geom. For example, we may want to plot each lexical item and its mean duration, represented by a bar.

```
ggplot(I_jean, aes(Word, Dur_msec))+
  stat_summary(fun.y = mean, geom = "bar")
```



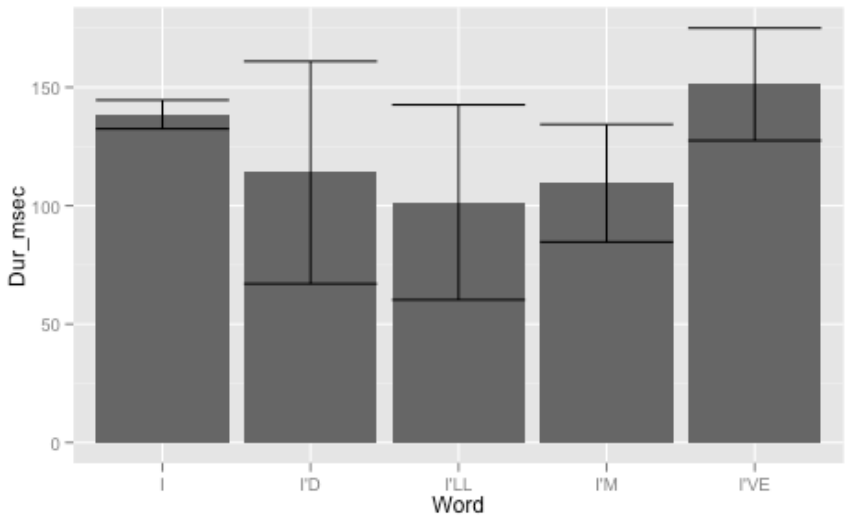
You can experiment with different summary functions and geometries as you desire.

Along with defining `fun.y`, you can also define `fun.data`, which allows you to plot ranges of data summaries with errorbars, or other similar geometries. There are 4 nicely tuned functions in `ggplot2` for doing this.

- `mean_cl_boot()`
  - This will return the sample mean, and 95% bootstrap confidence intervals.
- `mean_cl_normal()`
  - This will return the sample mean, and the 95% percent Gaussian confidence interval based on the t-distribution
- `mean_sdl()`
  - This will return the sample mean and values at 1 sd and -1 sd away. You can make it return points any arbitrary number of sds away by passing that value to `mult`. For example, `mult = 2` will return 2 and -2 sds.
- `median_hilow()`
  - This will return the sample median, and confidence intervals running from the 0.025 quantile to the 0.975 quantile, which covers 95% of the range of the data. You can change what range of the data you want the confidence interval to cover by passing it to `conf.int`. For example `conf.int = 0.5` will return confidence intervals ranging from the 0.25 quantile to the 0.75 quantile.

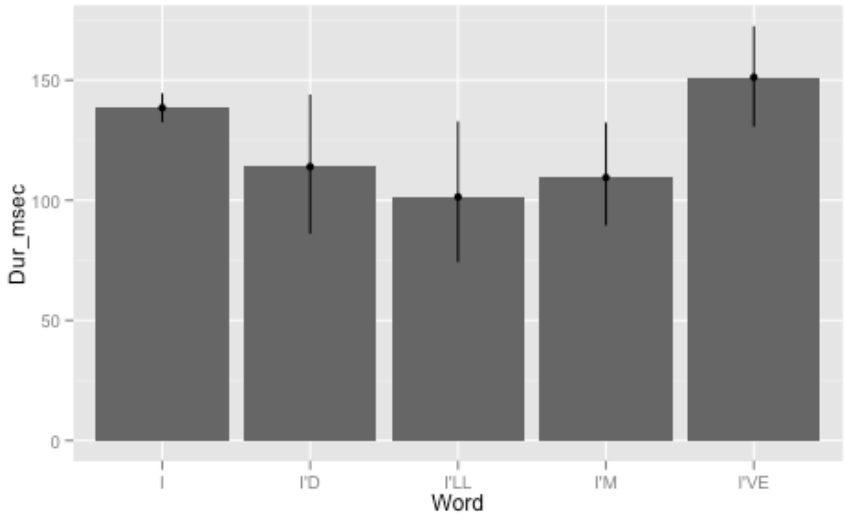
Coming back to the example of plotting the mean durations for each word, all we need to do is add another `stat_summary()` layer. For ordinary error bars, we'll use `geom_errorbar()`.

```
ggplot(I_jean, aes(Word, Dur_msec))+
  stat_summary(fun.y = mean, geom = "bar", fill = "grey40")+
  stat_summary(fun.data = mean_cl_normal, geom = "errorbar")
```



But I think that error bars are pretty ugly in comparison to point ranges, and it's always worth going for bootstrapped estimates since they have fewer assumptions.

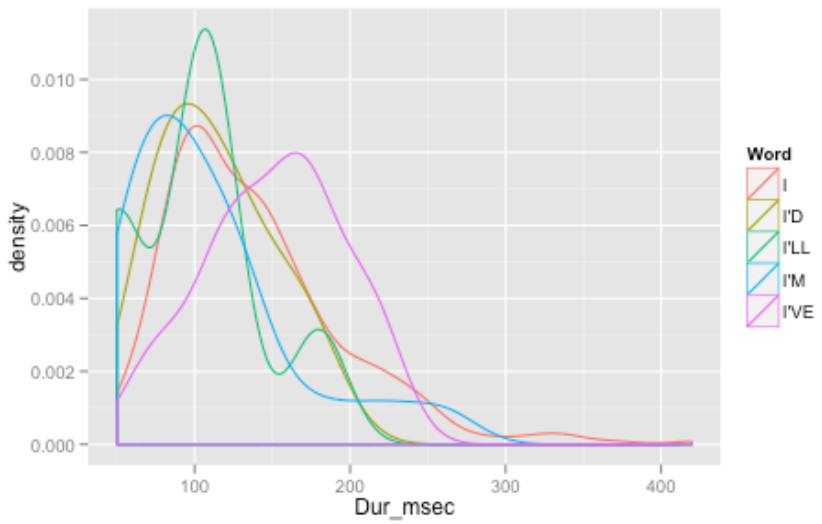
```
ggplot(I_jean, aes(Word, Dur_msec))+
  stat_summary(fun.y = mean, geom = "bar", fill = "grey40")+
  stat_summary(fun.data = mean_cl_boot, geom = "pointrange")
```



Values created by statistics

Statistical layers added to plots actually create new pieces of data, like the y-coordinates of the smoother. Some statistical layers create a few different values, and you can choose which one you want to plot. For example, here is a density plot, where the kernel density estimate is represented by a colored line.

```
ggplot(I_jean, aes(Dur_msec, color = Word))+
  geom_density()
```

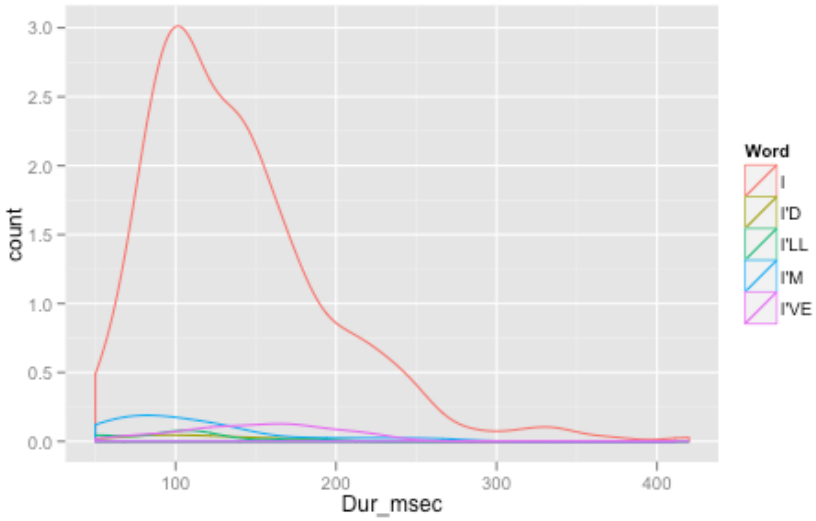


You have to understand the densities represented in this plot as being conditional on selecting a specific word. That is, given that we have decided to think about the lexical item “I've”, what is the probability it will

be found in a specific range of durations?

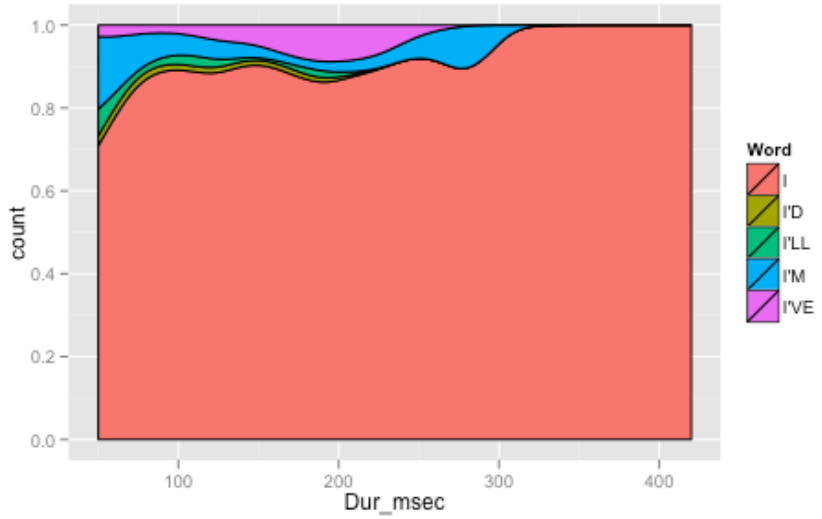
However, it's also possible to plot densities that are not conditional lexical item. That is, given a range of durations, what is the mixture of lexical items we'll find there? We can do this by plotting a value called `..count..` along the y-axis. `..count..` is created by `stat_density()`, which is why it's begins and ends with `...` All values that are created by a statistical layer begin and end with `..`. Here's the result of plotting `..count..` along the y-axis.

```
ggplot(I_jean, aes(Dur_msec, color = Word))+
  geom_density(aes(y = ..count..))
```

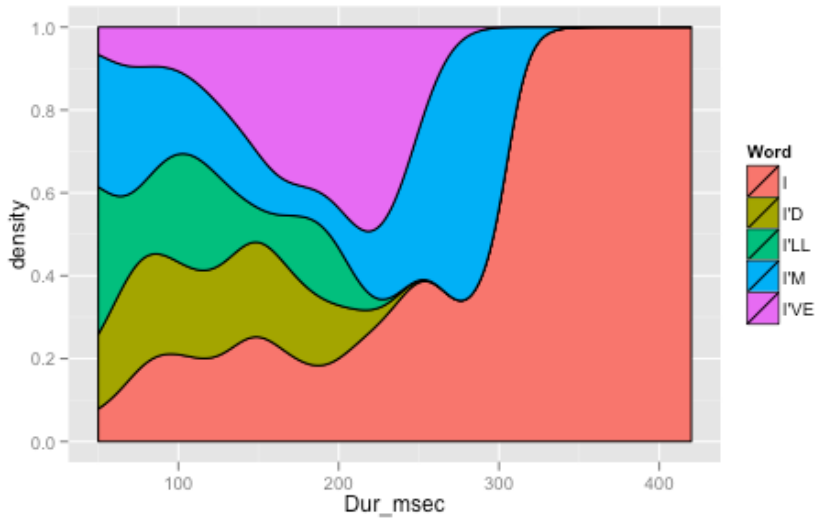


The density distribution for “I” is suddenly huge, and that's because it's so frequent. Note that I used `y = ..count..` for stacking and filling density distributions above, because that's actually the only thing that makes any sense. Compare the following two filled density plots.

```
ggplot(I_jean, aes(Dur_msec, fill = Word))+
  geom_density(aes(y = ..count..), position = "fill")
```



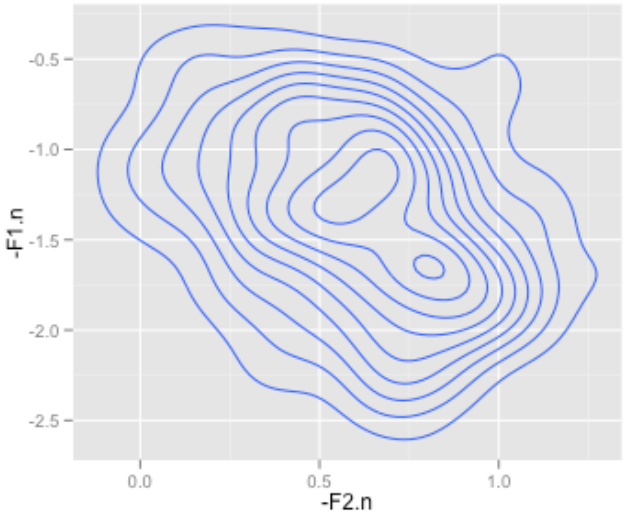

```
ggplot(I_jean, aes(Dur_msec, fill = Word))+
  geom_density(aes(y = ..density..), position = "fill")
```



We *know* that "I" is super frequent, so the plot with `y = ..count..` is the accurate one. What the plot with `y = ..density..` is actually displaying is a little complicated to worry about, and it would almost certainly confuse any readers of your papers.

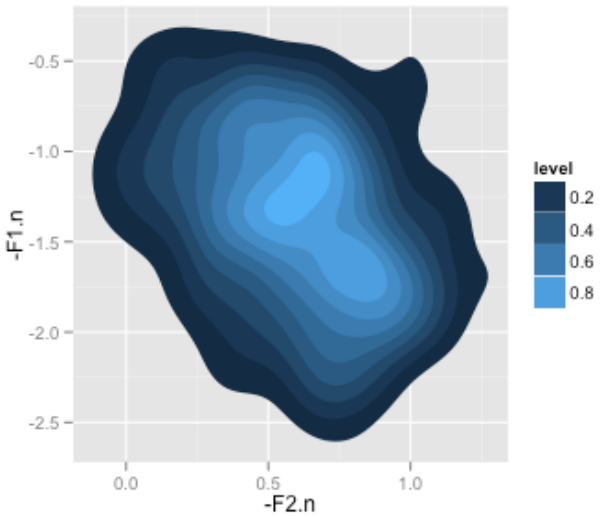
Another cool use of values generated by a statistic is with two dimensional density estimation. Here's an F2 by F1 plot, illustrating the default behavior of `stat_density2d()`.

```
ggplot(I_jean, aes(-F2.n, -F1.n))+
  stat_density2d()
```



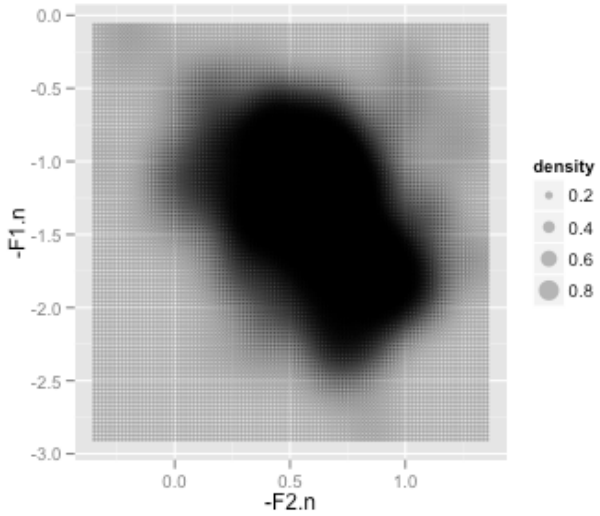
The default behavior of `stat_density2d()` is to bin up the two dimensional density estimates into discrete levels for plotting as topographic contours. The value corresponding to this discretized density estimate is called `..level..`, and we can use it to replace the contour lines with filled in polygons.

```
ggplot(I_jean, aes(-F2.n, -F1.n))+
  stat_density2d(geom = "polygon", aes(fill = ..level..))
```

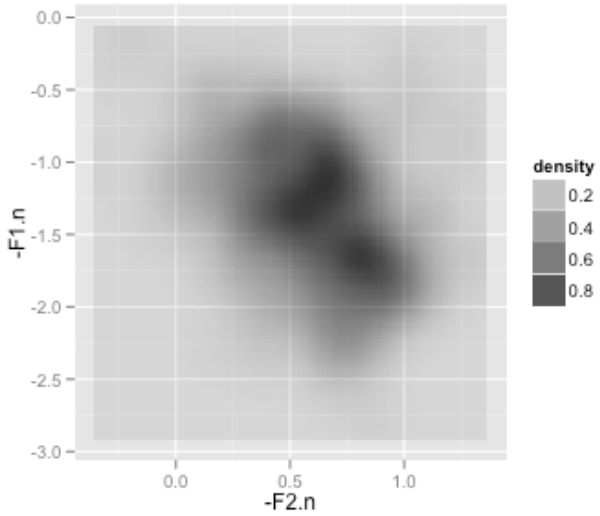


Or, we can turn off the discritization entirely by saying `contour = F`, and then access the density estimate itself, `..density..` and map that to a variety of different aesthetics.

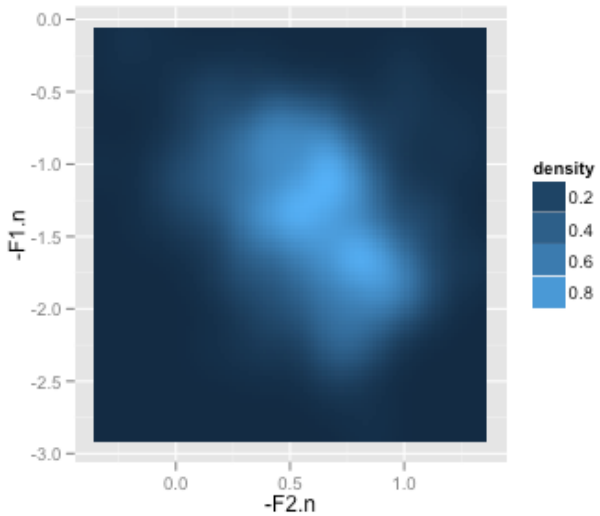
```
ggplot(I_jean, aes(-F2.n, -F1.n))+
  stat_density2d(geom = "point",contour = F,
                 aes(size = ..density..), alpha = 0.3)
```




```
ggplot(I_jean, aes(-F2.n, -F1.n))+
  stat_density2d(geom = "tile", contour = F, aes(alpha = ..density..
```




```
ggplot(I_jean, aes(-F2.n, -F1.n))+
  stat_density2d(geom = "tile", contour = F, aes(fill = ..density..)
```



Scales

Scales provide you more fine grained control over over the presentation of aesthetics. For every aesthetic, there is a corresponding scale you can use.

x and y scales

All scales begin with `scale_` followed by the name of the aesthetic it controls, then followed by its sub-type. Here are all the x-axis scales, for which there are identical y-axis scales.

```
apropos("^scale_x_")
```

```
## [1] "scale_x_continuous" "scale_x_date"
## [3] "scale_x_datetime"   "scale_x_discrete"
## [5] "scale_x_log10"      "scale_x_reverse"
## [7] "scale_x_sqrt"
```

`scale_x_continuous`, `scale_x_discrete`, `scale_x_datetime` and `scale_x_date` are the basic kinds of x and y axes you can construct in `ggplot2`. For the most part, `ggplot2` will figure out which kind of scale to use, and you'll only need to add one of these if you want to modify the default appearance.

`scale_x_log10`, `scale_x_sqrt` and `scale_x_reverse` are basic transformations to a continuous scale. Many more kinds of transformations are possible, and it's even possible to define your own custom transformations, but these are the only ones for which there are special convenience `scale_x_*` functions.

`scale_` arguments

There are a few basic arguments that you can pass to a scale.

- `name`
  - This is the title that will be displayed on the scale, either on the axes for x and y scales, or in the legend for other scales.
- `limits`
  - This defines the range of data to be presented in the plot. For discrete scales, it'll define the order with which to display categorical factors.
- `breaks`
  - These are the labeled points along the scale, either the major axis

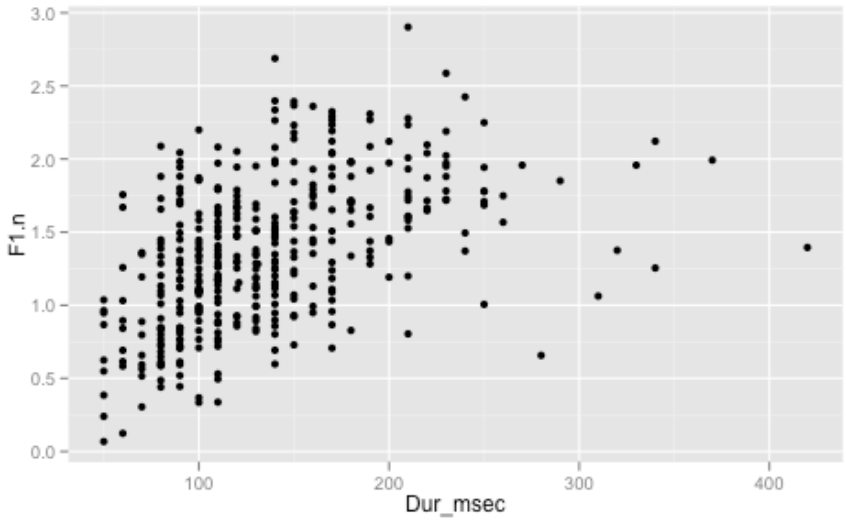
labels, or the labels on the legend.

- `labels`
  - This defines the labels to use at each break point, if you want to override them
- `trans`
  - This defines a numerical transformation you'd like to apply to a scale, and usually only applicable to continuous scales. We'll talk more about transformations below.

`scale_[xy]_continuous`

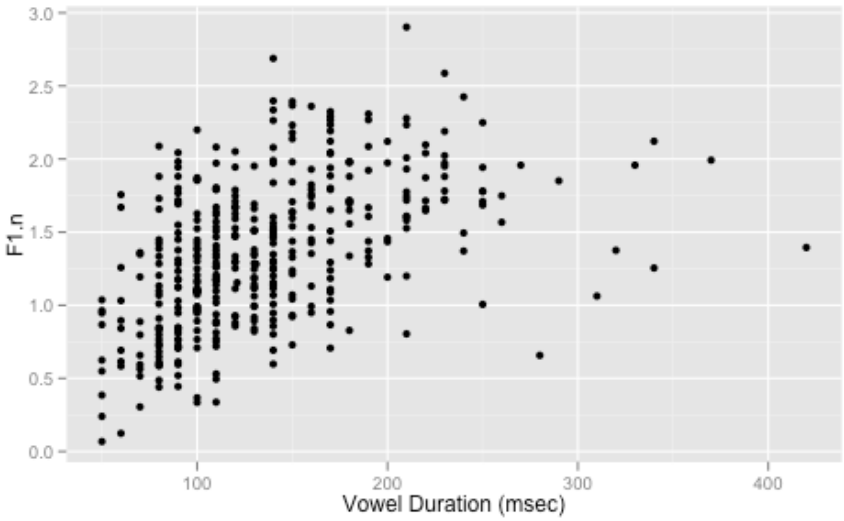
Here's our basic duration by F1 plot again, which has two continuous scales for the x and y axes.

```
ggplot(I_jean, aes(Dur_msec, F1.n))+
  geom_point()
```



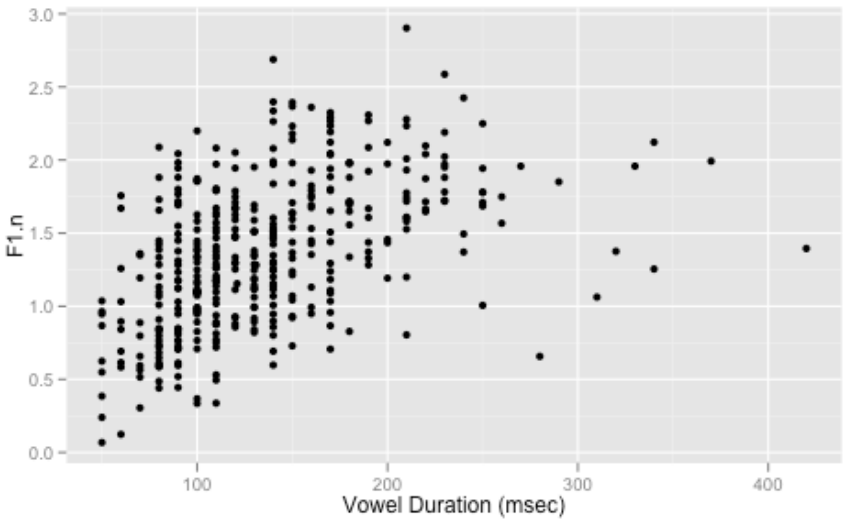
If we want to change the x axis label, we can do so by adding `scale_x_continious()` and passing our desired title to `name`.

```
ggplot(I_jean, aes(Dur_msec, F1.n))+
  geom_point()+
  scale_x_continuous(name = "Vowel Duration (msec)")
```



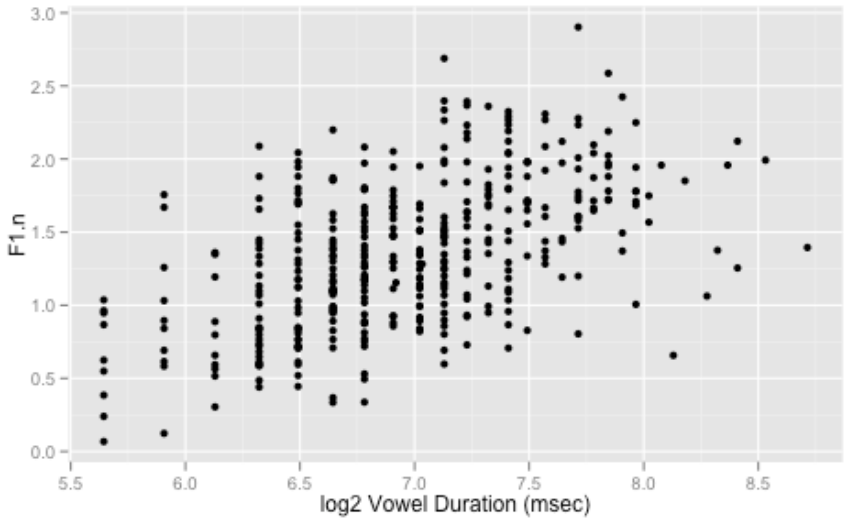
Since changing the axis titles is something you're likely to do very frequently, there are two convenience functions for this purpose with shorter names that save you typing: `xlab()` and `ylab()`.

```
ggplot(I_jean, aes(Dur_msec, F1.n))+
  geom_point()+
  xlab("Vowel Duration (msec)")
```

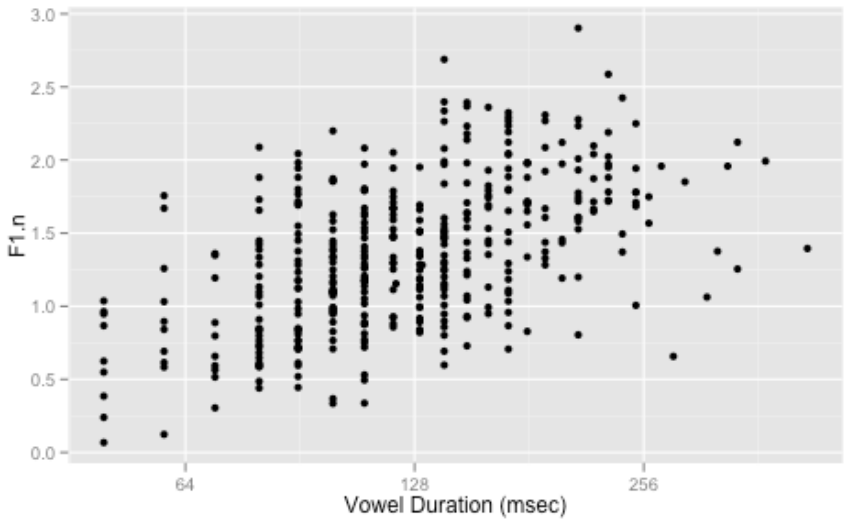


The next most important thing you'll want to do to x and y axis scales is transform them. For example, duration measurements tend to be left-skewed, so a log transformation is advisable. The benefit of transforming the scale over simply plotting `log2(Dur_msec)` is that `ggplot2` will very nicely label the axis according to the original values. Compare the labels of the x-axes of these two plots.

```
ggplot(I_jean, aes(log2(Dur_msec), F1.n))+
  geom_point()+
  scale_x_continuous("log2 Vowel Duration (msec)")
```




```
ggplot(I_jean, aes(Dur_msec, F1.n))+
  geom_point()+
  scale_x_continuous("Vowel Duration (msec)",
                     trans = "log2")
```



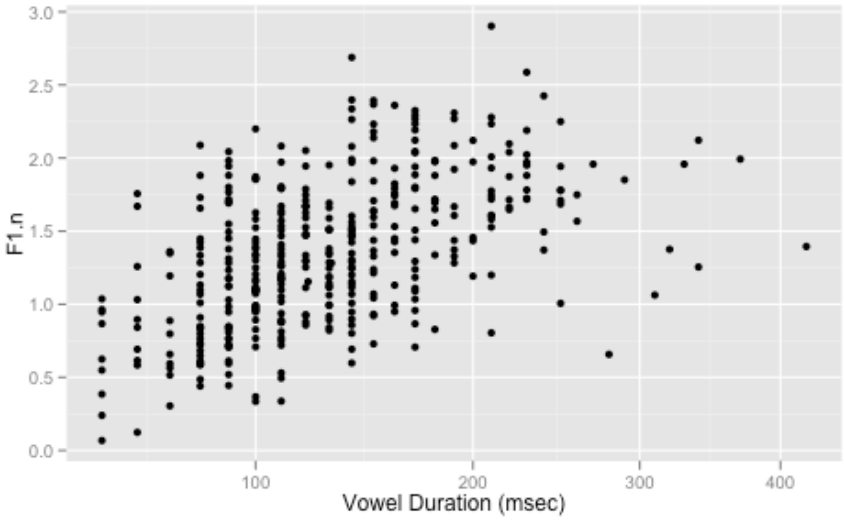
These are all of the pre-defined transformations.

```
apropos("_trans")
```

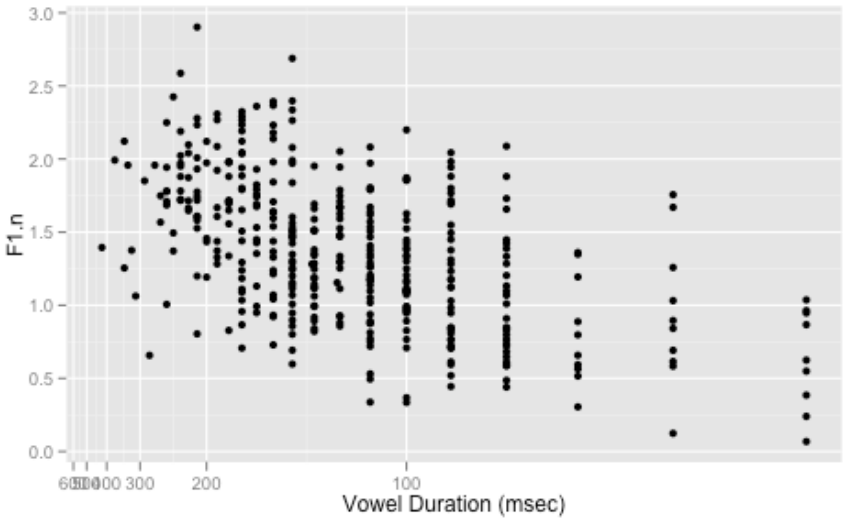
```
##  [1] "asn_trans"         "atanh_trans"
##  [3] "boxcox_trans"      "coord_trans"
##  [5] "date_trans"        "exp_trans"
##  [7] "identity_trans"    "log10_trans"
##  [9] "log1p_trans"       "log2_trans"
## [11] "log_trans"         "logit_trans"
## [13] "probability_trans" "probit_trans"
## [15] "reciprocal_trans"  "reverse_trans"
## [17] "sqrt_trans"        "time_trans"
```

And here are some in action.

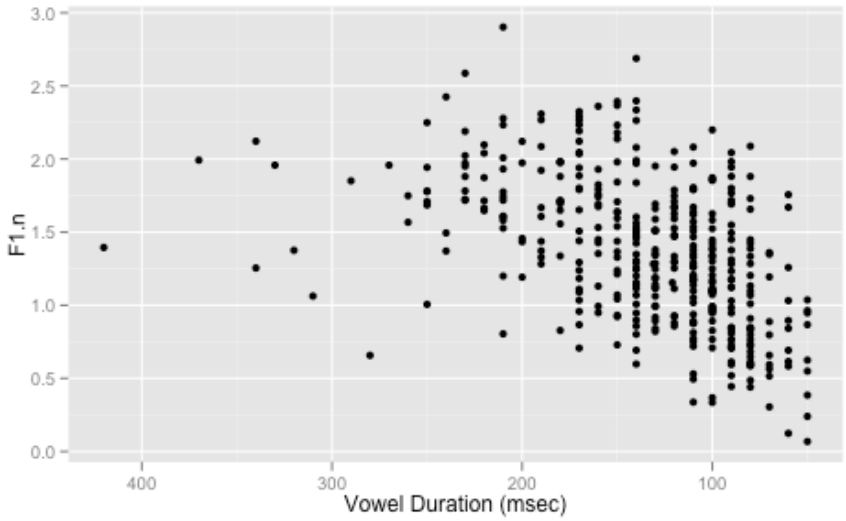
```
ggplot(I_jean, aes(Dur_msec, F1.n))+
  geom_point()+
  scale_x_continuous("Vowel Duration (msec)",
                     trans = "sqrt")
```




```
ggplot(I_jean, aes(Dur_msec, F1.n))+
  geom_point()+
  scale_x_continuous("Vowel Duration (msec)",
                     trans = "reciprocal")
```




```
ggplot(I_jean, aes(Dur_msec, F1.n))+
  geom_point()+
  scale_x_continuous("Vowel Duration (msec)",
                     trans = "reverse")
```



One frustrating thing is that it's not possible to apply two transformations to a scale at once. For example, I frequently want to both log transform an axis *and* reverse it. It's possible, though, to define a new transformation with the `scales` package, but for that I'll refer you to the `ggplot 0.9.0` transition guide.

color and fill scales

Here are all of the color scales, for all of which there is an accompanying fill scale.

```
apropos("^scale_color_")
```

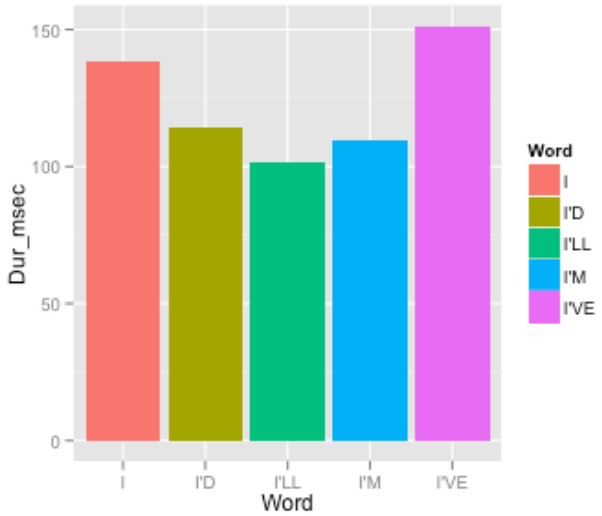
```
##  [1] "scale_color_brewer"     "scale_color_continuous"
##  [3] "scale_color_discrete"   "scale_color_gradient"
##  [5] "scale_color_gradient2"  "scale_color_gradientn"
##  [7] "scale_color_grey"       "scale_color_hue"
##  [9] "scale_color_identity"   "scale_color_manual"
```

There are two very distinct kinds of color scales here: categorical and gradient. The use of one over the other has everything to do with the kind of data which is passed to `color` and `fill`, and they have their own specific customizations.

Categorical color scales

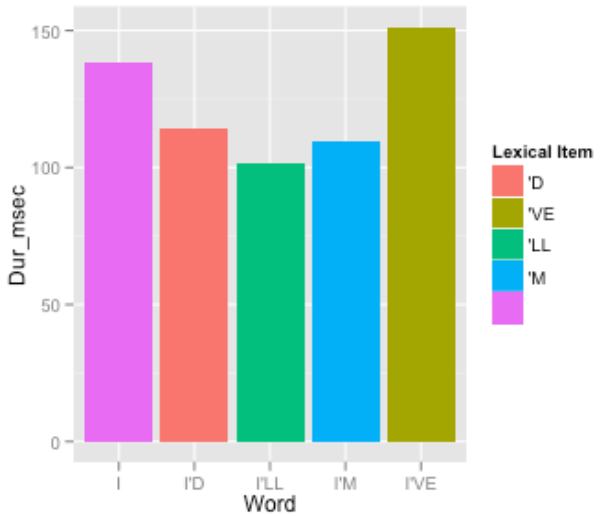
Here is the default categorical color scale, which is called `scale_[fill/color]_hue()`

```
ggplot(I_jean, aes(Word, Dur_msec, fill = Word))+
  stat_summary(fun.y = mean, geom = "bar")+
  scale_fill_hue()
```



Just like the x and y scales, you can change the title, limits, breaks and labels of this scale, which will control how it appears in the legend. Here's an illustrative example, which actually detracts from the default

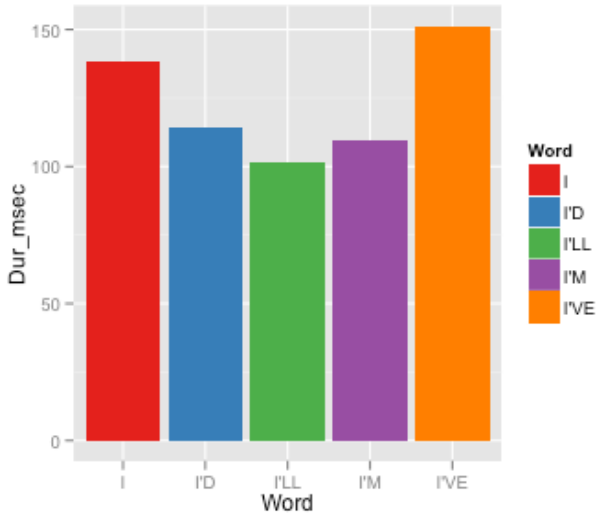
```
ggplot(I_jean, aes(Word, Dur_msec, fill = Word))+
  stat_summary(fun.y = mean, geom = "bar")+
  scale_fill_hue(name = "Lexical Item",
                 limits = c("I'D","I'VE","I'LL","I'M","I"),
                 labels = c("'D","'VE","'LL","'M",""))
```



Many people don't like the default color scheme, but there are other available color palettes, and you can define your own. One really nice set of color palettes comes from the package `RColorBrewer`. You can

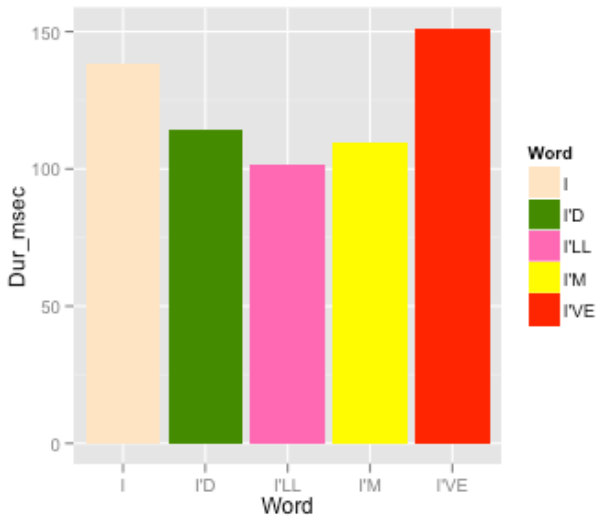
explore the set of available color palettes available in it here. A personal favorite of mine is called `Set1`, which you can apply to the plot with `scale_fill_brewer()`

```
ggplot(I_jean, aes(Word, Dur_msec, fill = Word))+
  stat_summary(fun.y = mean, geom = "bar")+
  scale_fill_brewer(palette = "Set1")
```



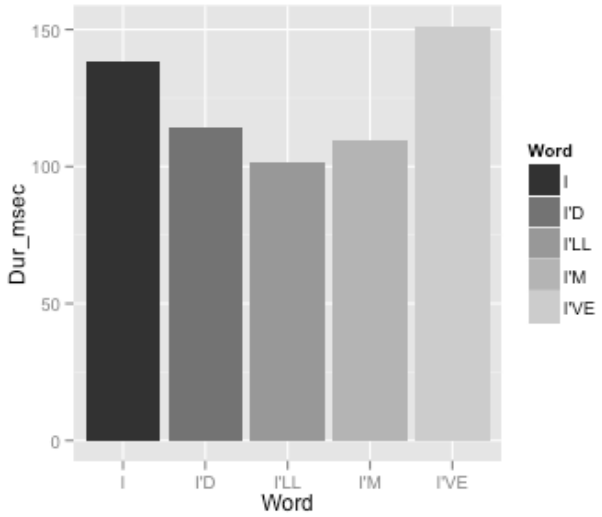
If you are particularly picky, or want to express your questionable aesthetic sense to the world, there is also `scale_fill_manual()`, where you define an arbitrary list of colors to use for the scale.

```
ggplot(I_jean, aes(Word, Dur_msec, fill = Word))+
  stat_summary(fun.y = mean, geom = "bar")+
  scale_fill_manual(values=c("bisque", "chartreuse4",
                             "hotpink","yellow", "red"))
```



And finally, if you're preparing a plot for publication, and you want to be sure that the colors will be distinguishable when printed in black and white, there is `scale_fill_grey()`.

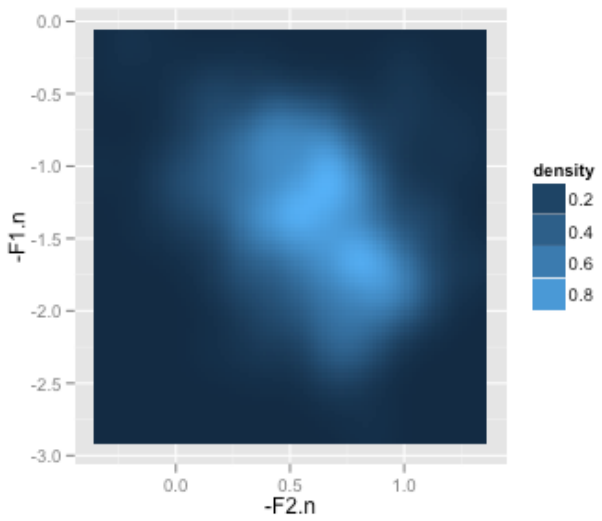
```
ggplot(I_jean, aes(Word, Dur_msec, fill = Word))+
  stat_summary(fun.y = mean, geom = "bar")+
  scale_fill_grey()
```



Gradient color scales

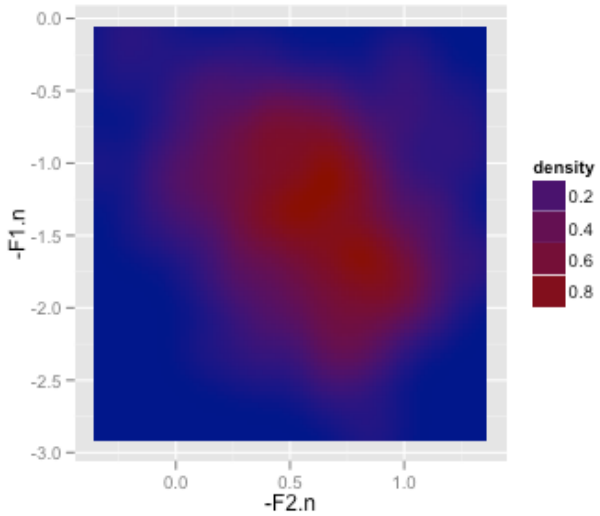
There are also a nice set of customizable gradient color scales. Here's the default gradient color scale, which is called `scale_fill_gradient()`

```
ggplot(I_jean, aes(-F2.n, -F1.n))+
  stat_density2d(geom = "tile",contour = F, aes(fill = ..density..))+
  scale_fill_gradient()
```



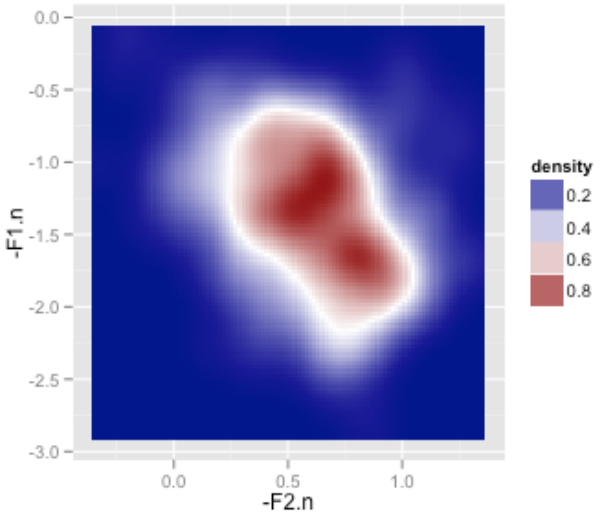
`scale_fill_gradient()` constructs a color continuum from A to B, where the lowest values in the plot will have solid A, and the highest values in the plot will have solid B, and everything else will fall along the gradient. It's possible to override the original two colors that the gradient is built between by passing the colors you prefer to `low` and `high`.

```
ggplot(I_jean, aes(-F2.n, -F1.n))+
  stat_density2d(geom = "tile",contour = F, aes(fill = ..density..))+
  scale_fill_gradient(low="darkblue",high="darkred")
```



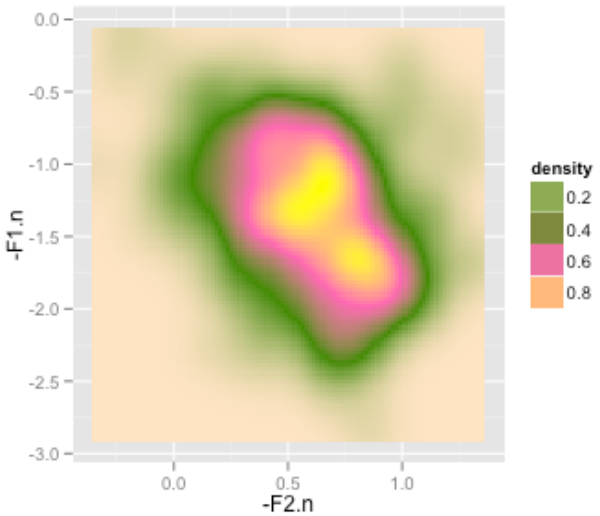
It's also possible to define a gradient that passes from A to B via C with `scale_fill_gradient2()`. Here, you define `low` and `high`, as well as a third color you want the gradient to transition through, `mid`. You also need to define the value the scale should treat as a midpoint.

```
ggplot(I_jean, aes(-F2.n, -F1.n))+
  stat_density2d(geom = "tile",contour = F, aes(fill = ..density..))+
  scale_fill_gradient2(low="darkblue",
                       high="darkred",
                       mid="white",
                       midpoint=0.5)
```



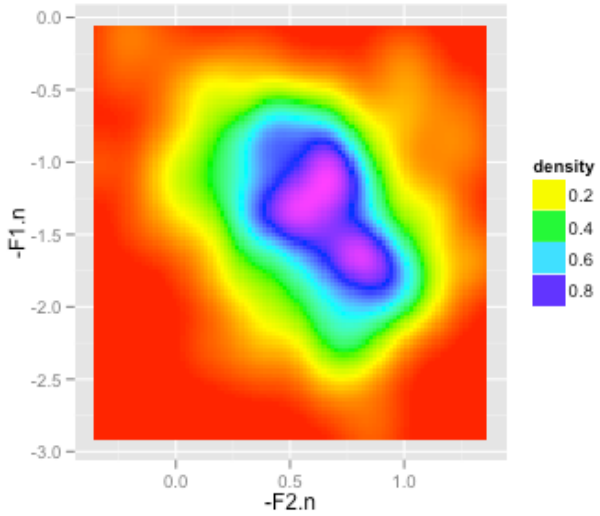
And, finally, you're able to define a color gradient passing through any arbitrary colors with `scale_fill_gradientn()`. Here's a pretty ugly one:

```
ggplot(I_jean, aes(-F2.n, -F1.n))+
  stat_density2d(geom = "tile",contour = F, aes(fill = ..density..))+
  scale_fill_gradientn(colours = c("bisque", "chartreuse4",
                                   "hotpink","yellow"))
```

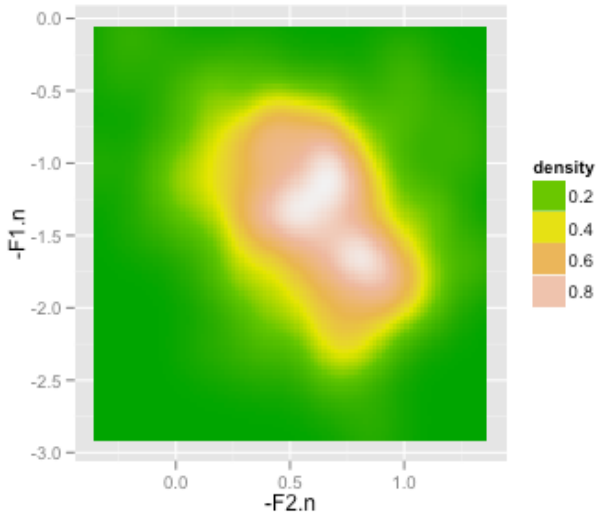


A benefit to having `scale_fill_gradientn()` is that you can utilize some of R's built in color palettes, like `rainbow()`, `terrain.colors()` and `topo.colors()`

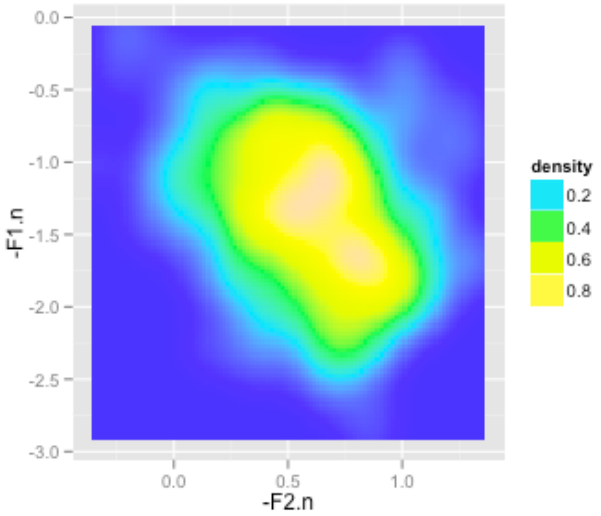
```
ggplot(I_jean, aes(-F2.n, -F1.n))+
  stat_density2d(geom = "tile",contour = F, aes(fill = ..density..))+
  scale_fill_gradientn(colours = rainbow(6))
```




```
ggplot(I_jean, aes(-F2.n, -F1.n))+
  stat_density2d(geom = "tile",contour = F, aes(fill = ..density..))+
  scale_fill_gradientn(colours = terrain.colors(6))
```



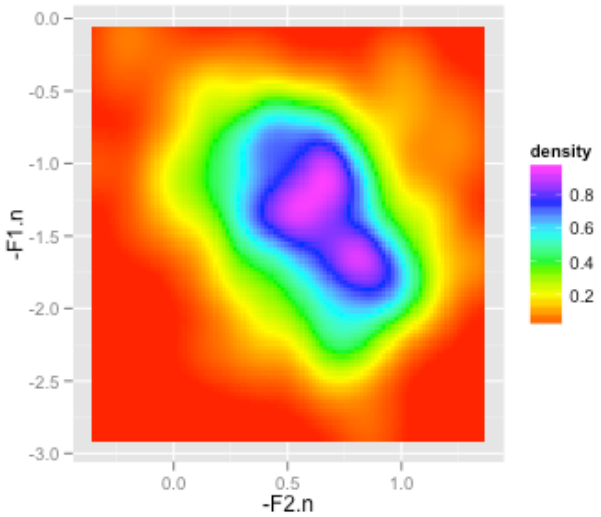

```
ggplot(I_jean, aes(-F2.n, -F1.n))+
  stat_density2d(geom = "tile",contour = F, aes(fill = ..density..))+
  scale_fill_gradientn(colours = topo.colors(6))
```



Guides

A set of new mechanics for handling the presentation of legends was introduced with version 0.9.0, including a new kind of legend for continuous color scales: the colorbar. The plots above have all used the default guide type (legend) which displays the color value for specific breaks, but not the gradient in between. The color bar guides show the entire gradient, with the breaks labeled over top.

```
ggplot(I_jean, aes(-F2.n, -F1.n))+
  stat_density2d(geom = "tile",contour = F, aes(fill = ..density..))+
  scale_fill_gradientn(colours = rainbow(6),
                       guide = "colorbar")
```



shape and linetype

The shape and linetype scales are much more limited. Despite the apparent existence of `scale_shape_continuous` and `scale_linetype_continuous`, you can't actually pass continuous variable to these aesthetics.

```
apropos("^scale_shape_")
```

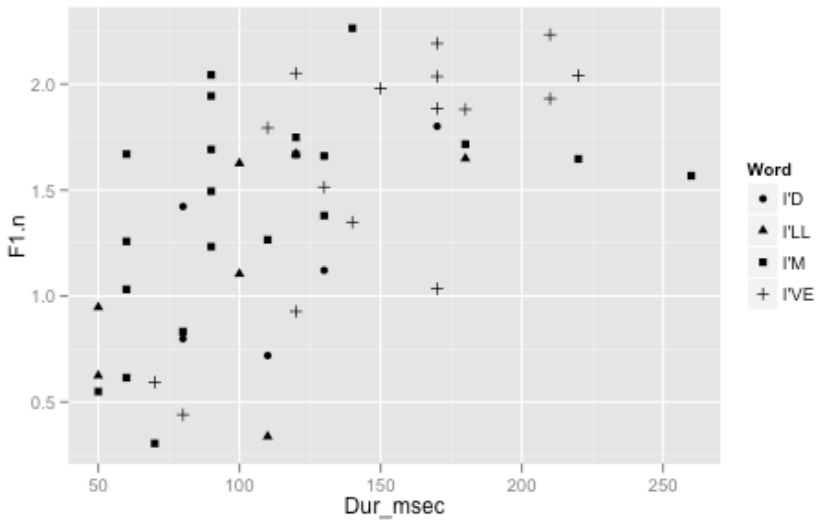
```
## [1] "scale_shape_continuous" "scale_shape_discrete"
## [3] "scale_shape_identity"   "scale_shape_manual"
```

```
apropos("^scale_linetype_")
```

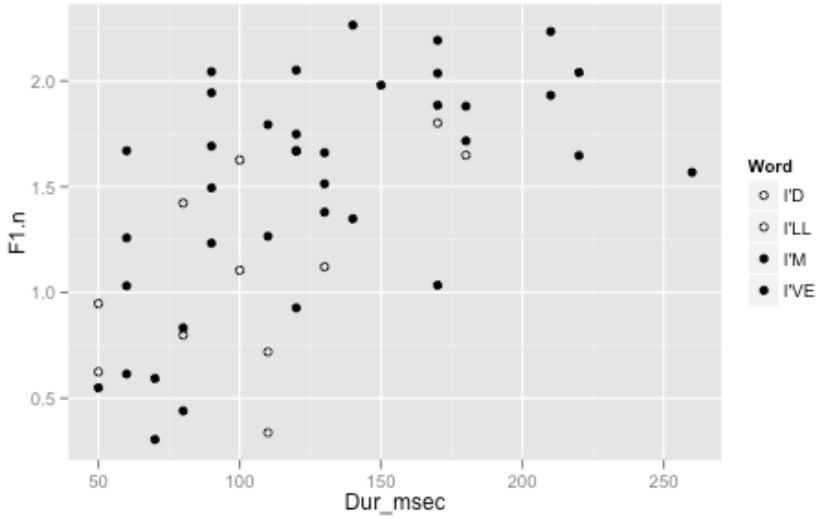
```
## [1] "scale_linetype_continuous" "scale_linetype_discrete"
## [3] "scale_linetype_identity"   "scale_linetype_manual"
```

The only time you'll be likely to use the shape and linetype scales is when you want to manually control the point shape and linetypes to be used. I actually find the default shape scale to not be strongly contrastive, and prefer to contrast point types based on filled vs hollow shapes.

```
ggplot(I_subset, aes(Dur_msec, F1.n, shape = Word))+
  geom_point()
```




```
ggplot(I_subset, aes(Dur_msec, F1.n, shape = Word))+
  geom_point()+
  scale_shape_manual(values=c(1,1, 19, 19))
```



Other scales

```
apropos("^scale_size_")
```

```
## [1] "scale_size_continuous" "scale_size_discrete"
## [3] "scale_size_identity"   "scale_size_manual"
```

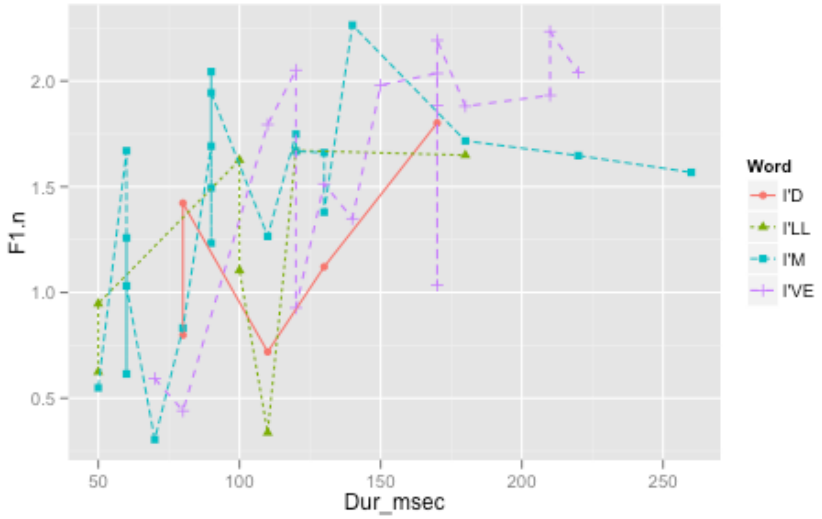
```
apropos("^scale_alpha_")
```

```
## [1] "scale_alpha_continuous" "scale_alpha_discrete"
## [3] "scale_alpha_identity"   "scale_alpha_manual"
```

More on Guides

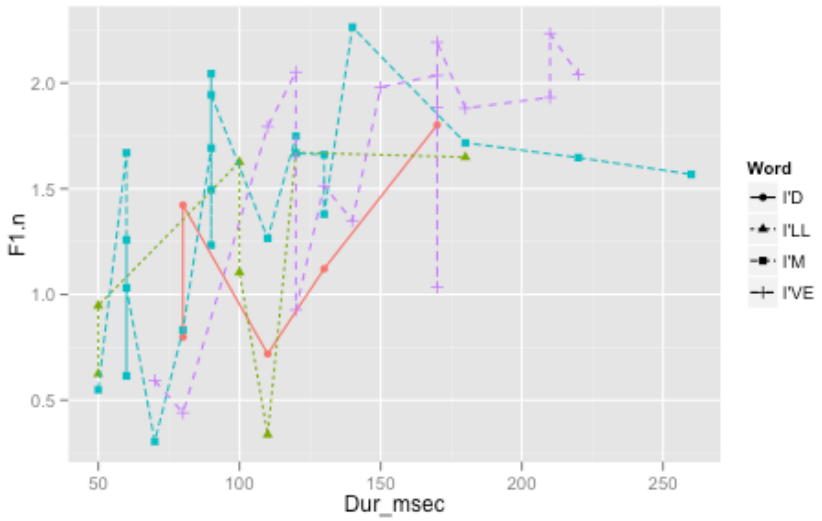
Occasionally, you'll want to exclude some scale, or some geom from being included in the legend. For example, here's a plot where `color`, `shape` and `linetype` are all mapped to `word`.

```
ggplot(I_subset, aes(Dur_msec, F1.n,
                     color = Word,
                     shape = Word,
                     linetype = Word))+
  geom_point()+
  geom_line()
```



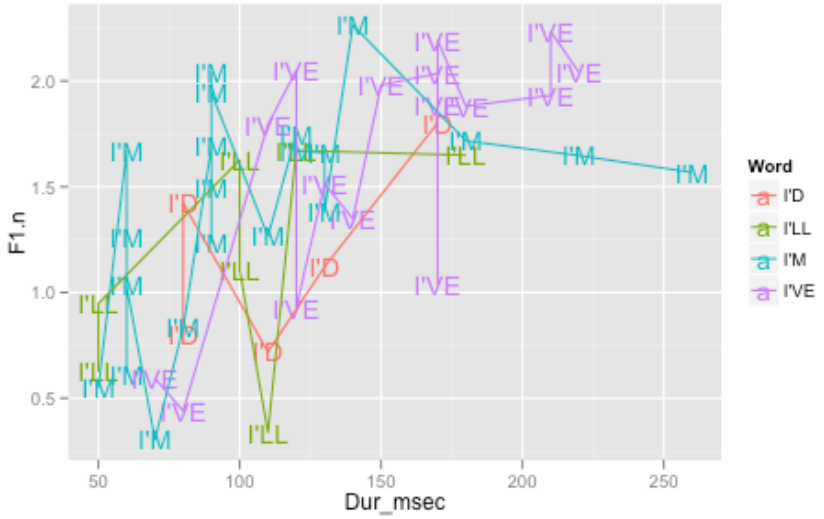
If, for some reason, you only wanted the shape and linetype to show up in the legend, but not color, you should explicitly add `scale_color_hue()` to the plot and pass it `guide = F`.

```
ggplot(I_subset, aes(Dur_msec, F1.n,
                     color = Word,
                     shape = Word,
                     linetype = Word))+
  geom_point()+
  geom_line()+
  scale_color_hue(guide = F)
```



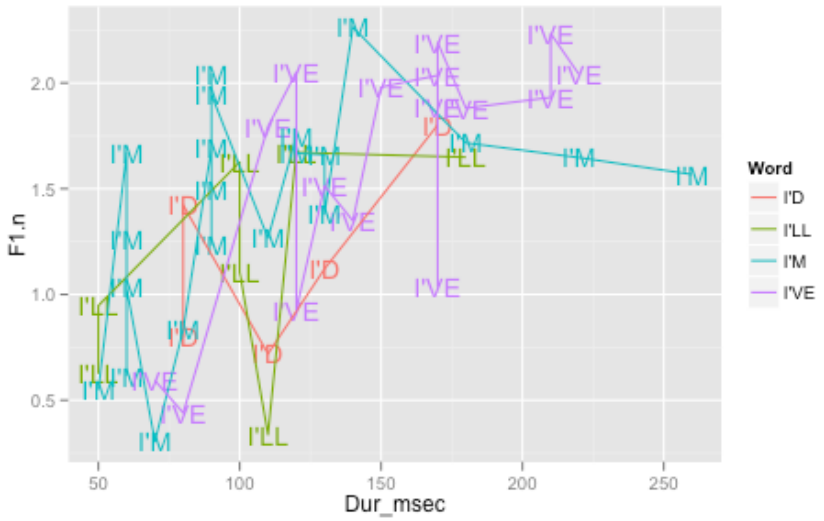
More often, though, you'll want to exclude specific geoms from being added to the legend. For example, I personally don't like how `geom_text()` is represented in legends.

```
ggplot(I_subset, aes(Dur_msec, F1.n, color = Word, label = Word))+
  geom_text()+
  geom_line()
```



To exclude this geom from appearing in the legend, simply pass `show_guide = F` to the geom.

```
ggplot(I_subset, aes(Dur_msec, F1.n, color = Word, label = Word))+
  geom_text(show_guide = F)+
  geom_line()
```



Coordinate systems

I don't have time to go over different coordinate systems in `ggplot2`, but for the most part everything you'd use them for is already captured by scale transformations. The only exception is `coord_polar()`, which was demonstrated at the beginning as a way of creating pie charts. But I think it's just as well because most plots in polar coordinates are inferior to the same plot in Cartesian coordinates,

Faceting

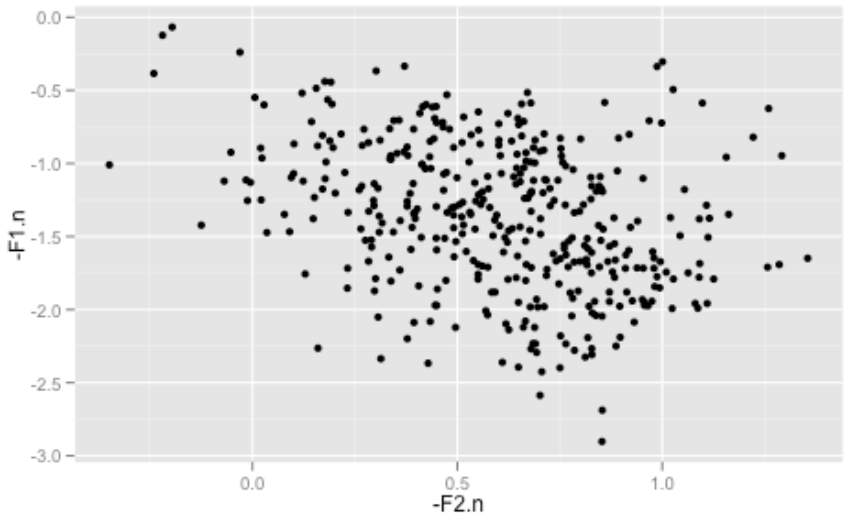
A really powerful graphical technique is the small multiple, and `ggplot2` allows for easy creation of small multiples via faceting. Let's create an additional categorical variable for the entire data set (we did this already for the subset excluding “I”).

```
I_jean$Dur_cat <- I_jean$Dur_msec > mean(I_jean$Dur_msec)
```

facet_wrap

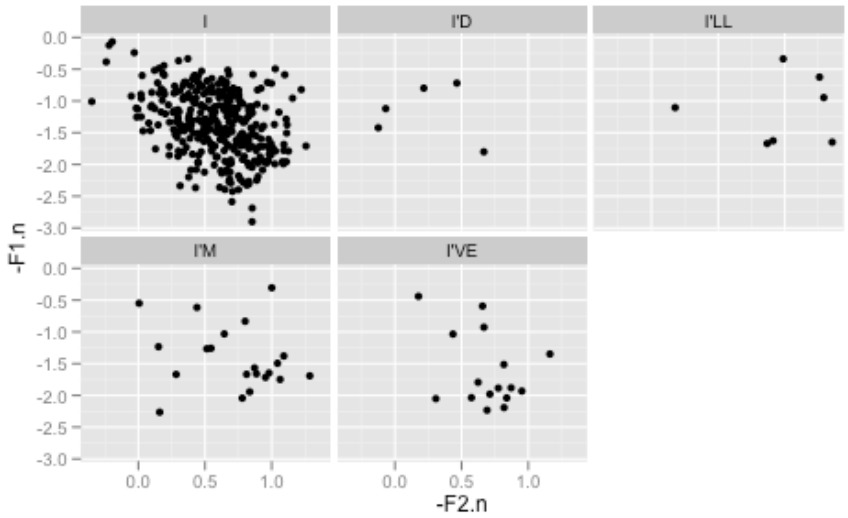
If we wanted to create an F2 x F1 plot for every word, we'd start out by creating a simple F1 x F2 plot:

```
ggplot(I_jean, aes(-F2.n, -F1.n ))+
  geom_point()
```



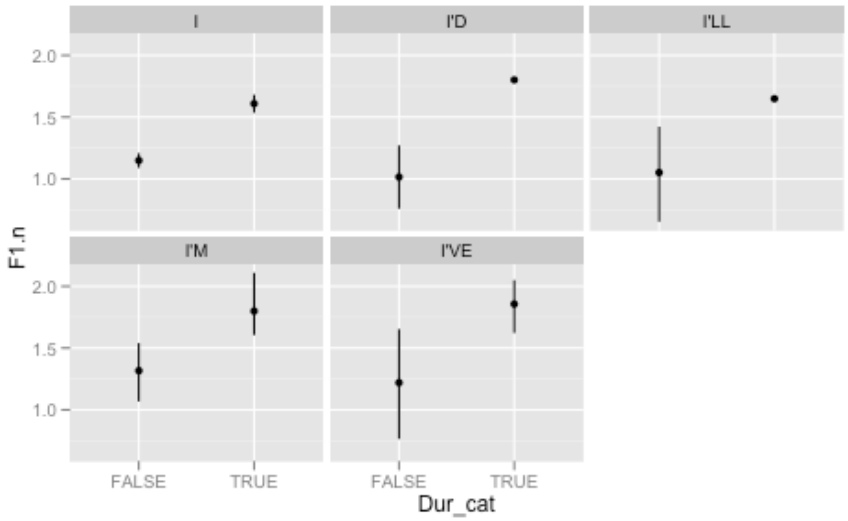
and then faceting by `Word` with `facet_wrap()`.

```
ggplot(I_jean, aes(-F2.n, -F1.n ))+
  geom_point()+
  facet_wrap(~Word)
```



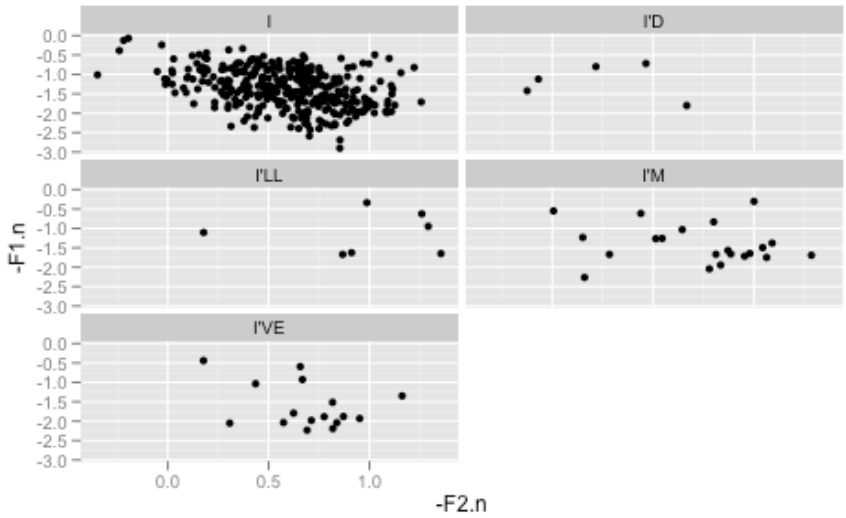
Or, if we wanted summaries of F1 according to `Dur_cat` for each word, we could do so like this.

```
ggplot(I_jean, aes(Dur_cat, F1.n))+
  stat_summary(fun.data = mean_cl_boot)+
  facet_wrap(~Word)
```



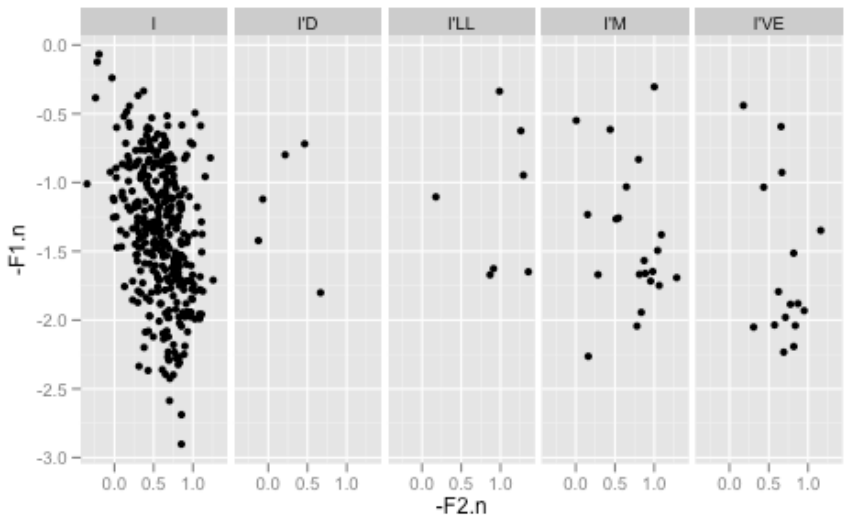
You can exercise some control about the layout of facets with `ncol` and `nrow`. For example. if you really only wanted there to be 2 columns of facets, you could make that happen with by passing `ncol=2` to `facet_wrap()`

```
ggplot(I_jean, aes(-F2.n, -F1.n ))+
  geom_point()+
  facet_wrap(~Word, ncol = 2)
```



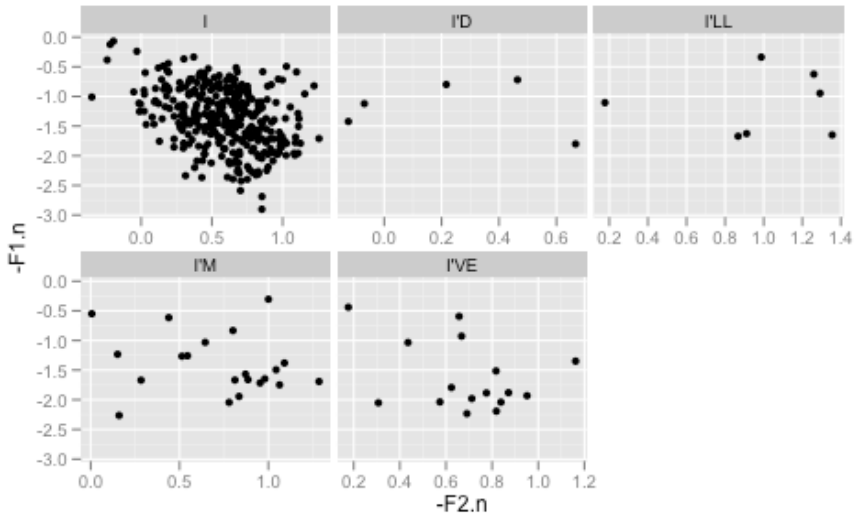
Or, if you wanted all the facets to be lined up in one row, you would pass `nrow = 1`.

```
ggplot(I_jean, aes(-F2.n, -F1.n ))+
  geom_point()+
  facet_wrap(~Word, nrow = 1)
```

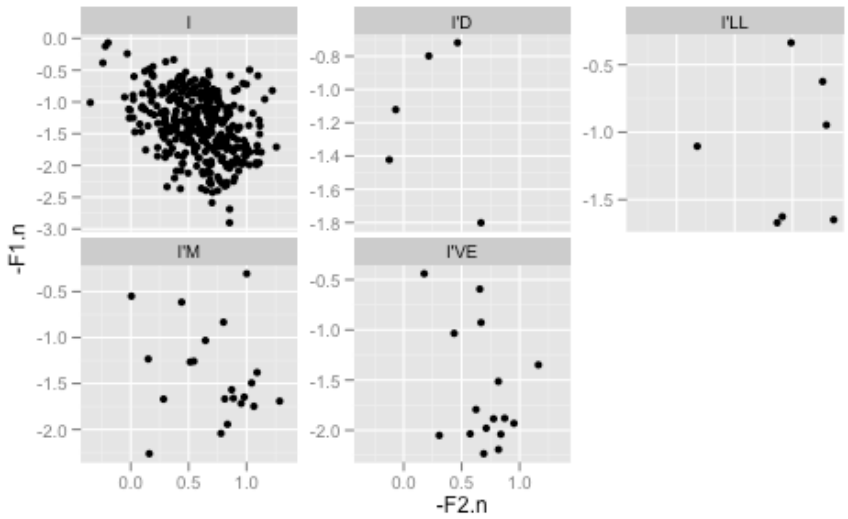


By default, the ranges of the axes in each facet are fixed to be the same across all facets, and that should be changed only in very limited circumstances. You can set the x axis, y axis, or both to be free by passing the following arguments to `scales` inside of `facet_wrap()`.

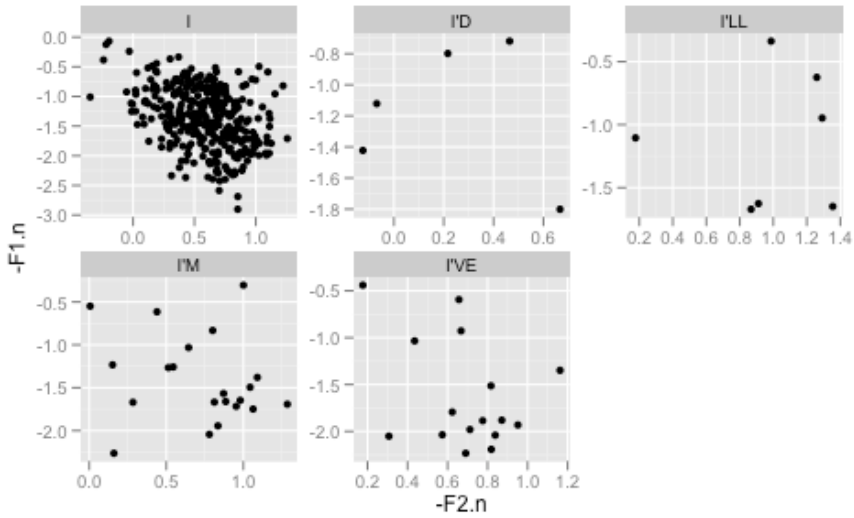
```
## Inadvisable
ggplot(I_jean, aes(-F2.n, -F1.n ))+
  geom_point()+
  facet_wrap(~Word, scales = "free_x")
```




```
## Inadvisable
ggplot(I_jean, aes(-F2.n, -F1.n ))+
  geom_point()+
  facet_wrap(~Word, scales = "free_y")
```



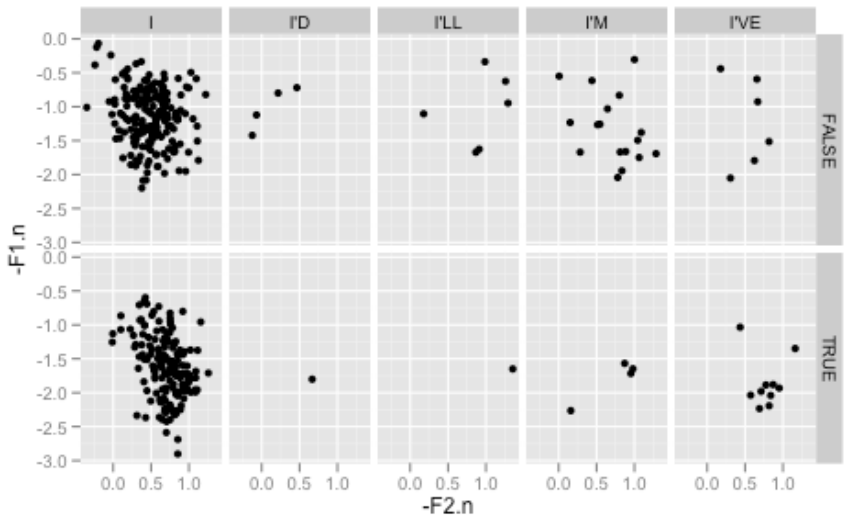

```
## Inadvisable
ggplot(I_jean, aes(-F2.n, -F1.n ))+
  geom_point()+
  facet_wrap(~Word, scales = "free")
```



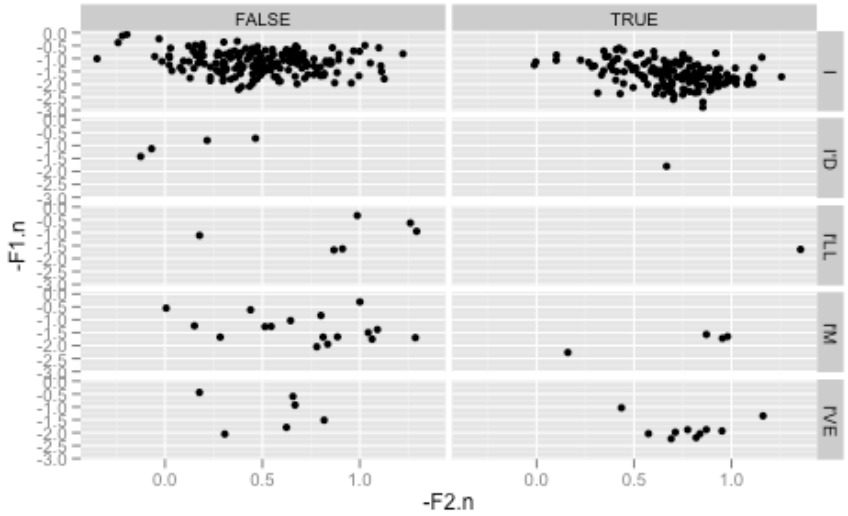
facet_grid

`facet_grid()` is another form of faceting in two dimensions.

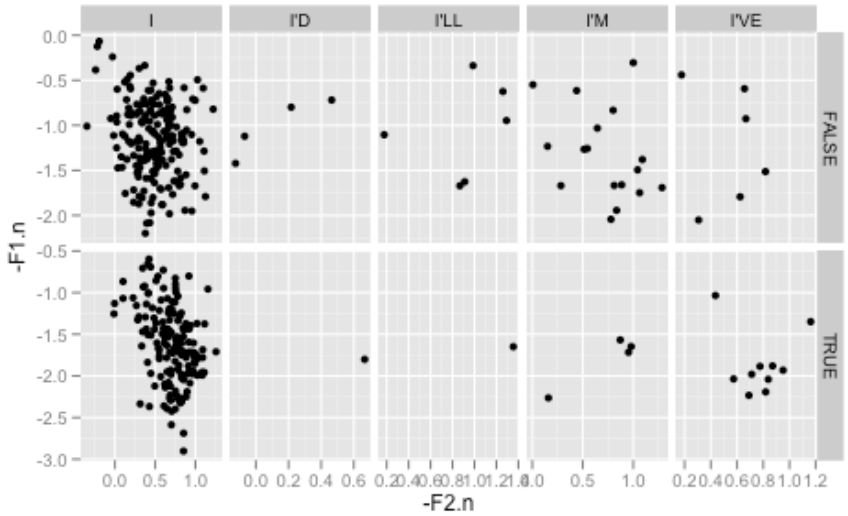
```
ggplot(I_jean, aes(-F2.n, -F1.n ))+
  geom_point()+
  facet_grid(Dur_cat~Word)
```




```
ggplot(I_jean, aes(-F2.n, -F1.n ))+
  geom_point()+
  facet_grid(Word~Dur_cat)
```




```
ggplot(I_jean, aes(-F2.n, -F1.n ))+
  geom_point()+
  facet_grid(Dur_cat~Word, scales = "free")
```

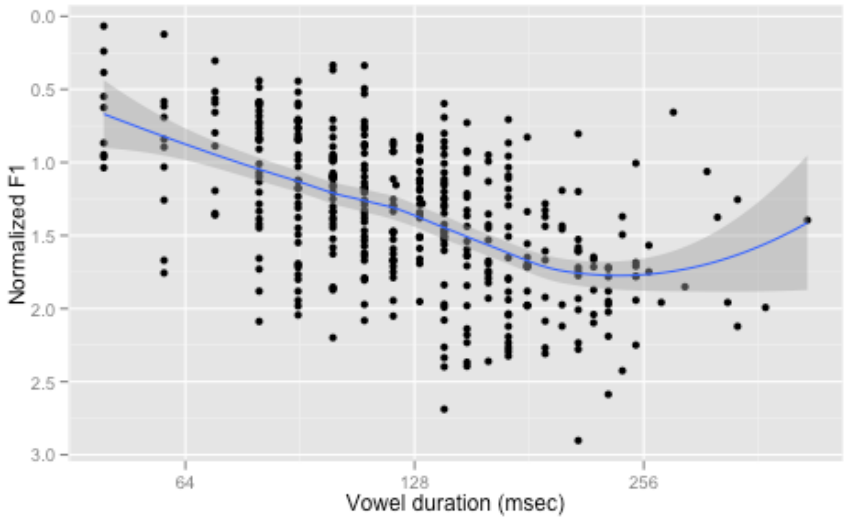


Additional options

Here, we'll go over how to fine tune a plot's final appearance, starting with theming.

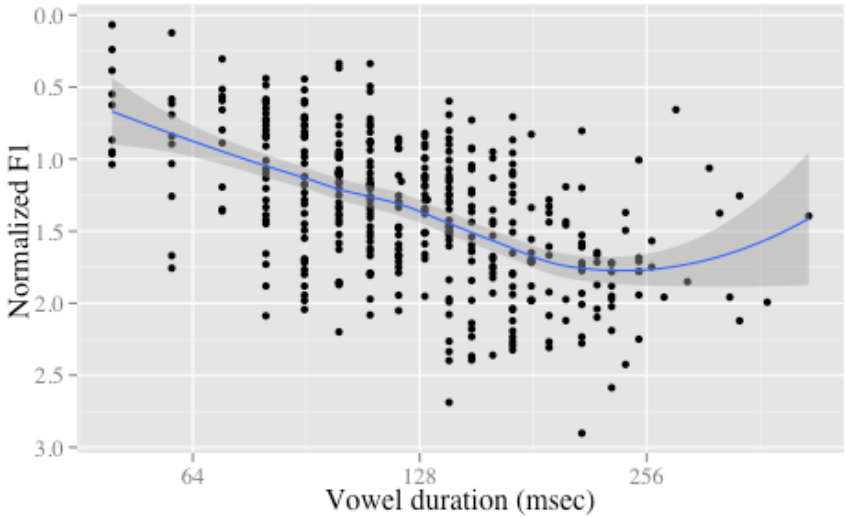
All `ggplot2` plots have a theme. The default theme is `theme_grey()`, and it should look very familiar to you now.

```
ggplot(I_jean, aes(x=Dur_msec, y=F1.n))+
  geom_point()+
  stat_smooth()+
  scale_x_continuous(trans = "log2")+
  scale_y_reverse()+
  ylab("Normalized F1")+
  xlab("Vowel duration (msec)")+
  theme_grey()
```



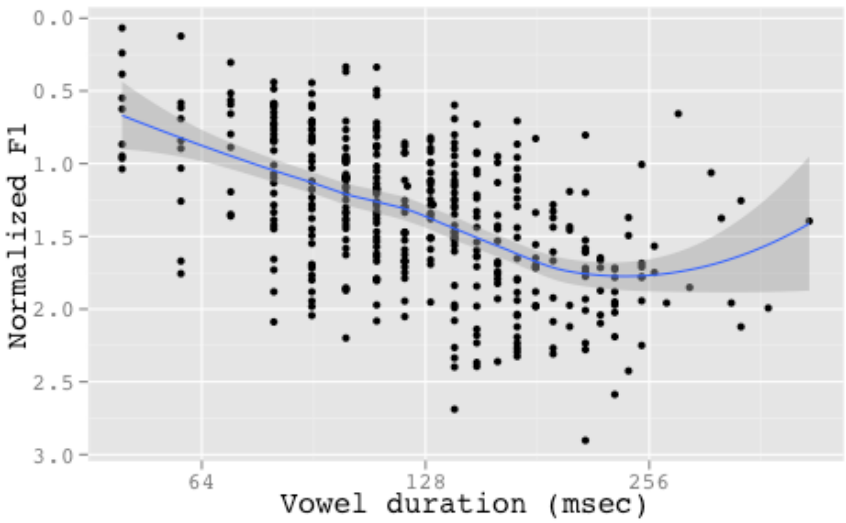
Within the theme, you can adjust the base font size, and the font family. For example, the default base size is 12, and the default font family is `sans`. If we wanted to make all the text a little bigger and serif, we would do so like this.

```
ggplot(I_jean, aes(x=Dur_msec, y=F1.n))+
  geom_point()+
  stat_smooth()+
  scale_x_continuous(trans = "log2")+
  scale_y_reverse()+
  ylab("Normalized F1")+
  xlab("Vowel duration (msec)")+
  theme_grey(base_size = 16, base_family = "serif")
```



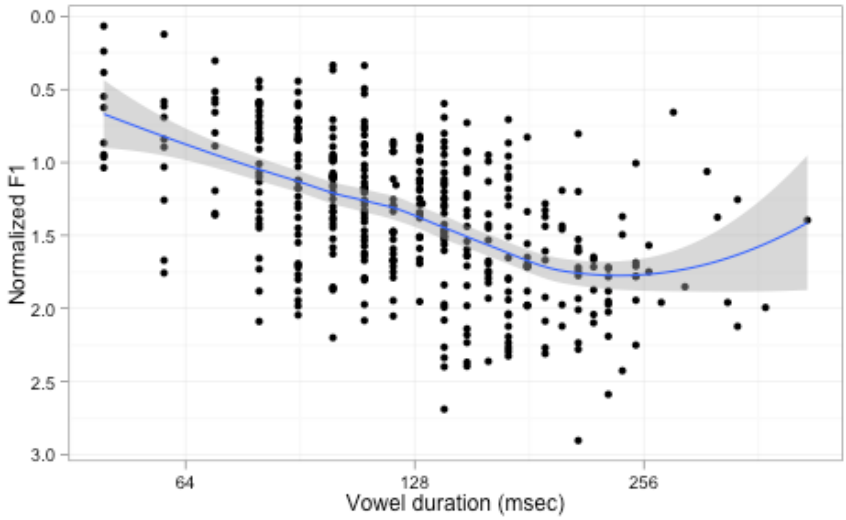
There is also a monospaced font family.

```
ggplot(I_jean, aes(x=Dur_msec, y=F1.n))+
  geom_point()+
  stat_smooth()+
  scale_x_continuous(trans = "log2")+
  scale_y_reverse()+
  ylab("Normalized F1")+
  xlab("Vowel duration (msec)")+
  theme_grey(base_size = 16, base_family = "mono")
```



If you want to drop the grey theme all together for a more traditional theme, use `theme_bw()`.

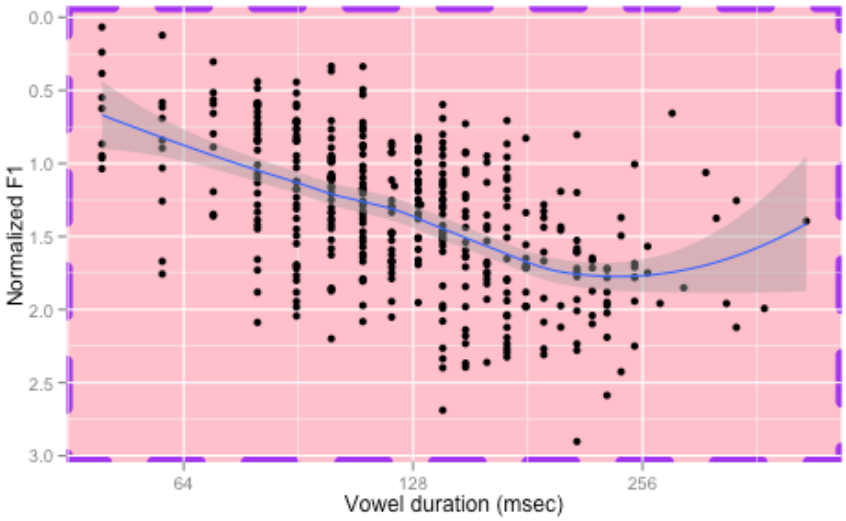
```
ggplot(I_jean, aes(x=Dur_msec, y=F1.n))+
  geom_point()+
  stat_smooth()+
  scale_x_continuous(trans = "log2")+
  scale_y_reverse()+
  ylab("Normalized F1")+
  xlab("Vowel duration (msec)")+
  theme_bw()
```



You can adjust the base font size and font family the same way with `theme_bw()`.

Outside of these simple approaches to theming, `ggplot2` plots are highly customizable, but it's not necessarily for the faint of heart. You can change almost anything by adding an `opts()` layer. For example

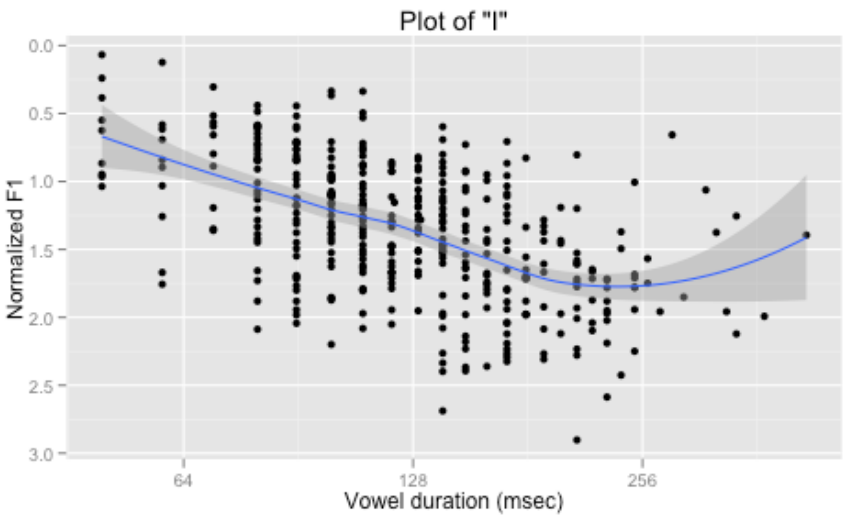
```
ggplot(I_jean, aes(x=Dur_msec, y=F1.n))+
  geom_point()+
  stat_smooth()+
  scale_x_continuous(trans = "log2")+
  scale_y_reverse()+
  ylab("Normalized F1")+
  xlab("Vowel duration (msec)")+
  opts(panel.background = theme_rect(colour = 'purple',
                                     fill = 'pink',
                                     size = 3,
                                     linetype='dashed'))
```



For a full list of all the customizable pieces of plots, and examples of their usage, I'll refer you to the opts() list wiki page.

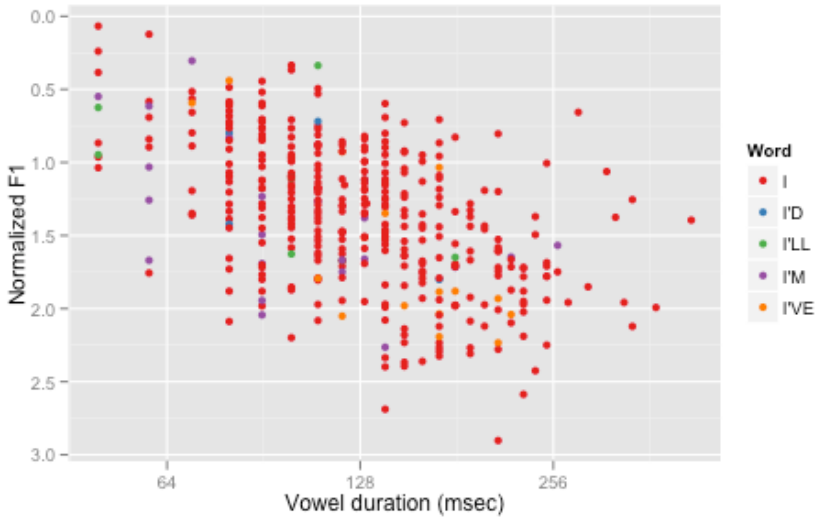
Some frequent options you'll use that we will cover, though, involve adding a title, and adjusting the legend position. To add a title, you simply pass a character string to `title` inside of `opts()`.

```
ggplot(I_jean, aes(x=Dur_msec, y=F1.n))+
  geom_point()+
  stat_smooth()+
  scale_x_continuous(trans = "log2")+
  scale_y_reverse()+
  ylab("Normalized F1")+
  xlab("Vowel duration (msec)")+
  opts(title = 'Plot of "I"')
```



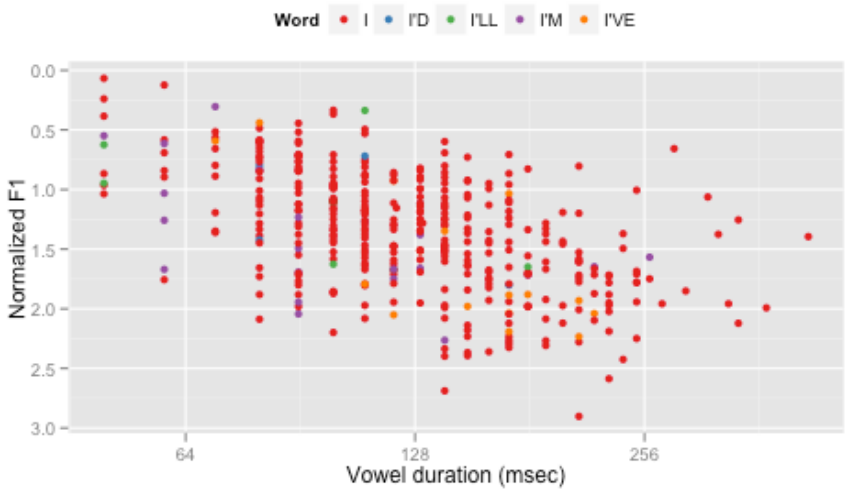
For the legend position, there are a 4 pre-specified locations: `right`, `left`, `top`, `bottom`. `right` is the default location.

```
ggplot(I_jean, aes(x=Dur_msec, y=F1.n, color = Word))+
  geom_point()+
  scale_x_continuous(trans = "log2")+
  scale_y_reverse()+
  scale_color_brewer(palette = "Set1")+
  ylab("Normalized F1")+
  xlab("Vowel duration (msec)")+
  opts(legend.position = "right")
```



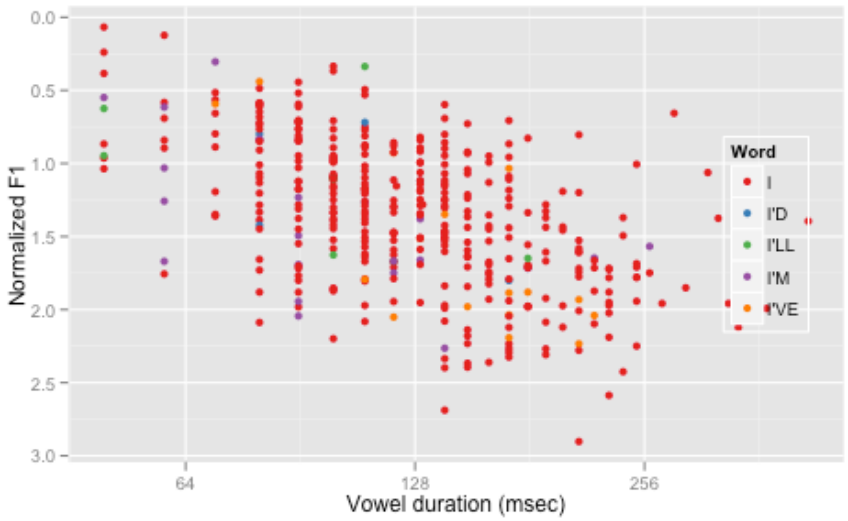
I think `top` is another reasonable location, more so than `left` or `bottom`.

```
ggplot(I_jean, aes(x=Dur_msec, y=F1.n, color = Word))+
  geom_point()+
  scale_x_continuous(trans = "log2")+
  scale_y_reverse()+
  scale_color_brewer(palette = "Set1")+
  ylab("Normalized F1")+
  xlab("Vowel duration (msec)")+
  opts(legend.position = "top")
```



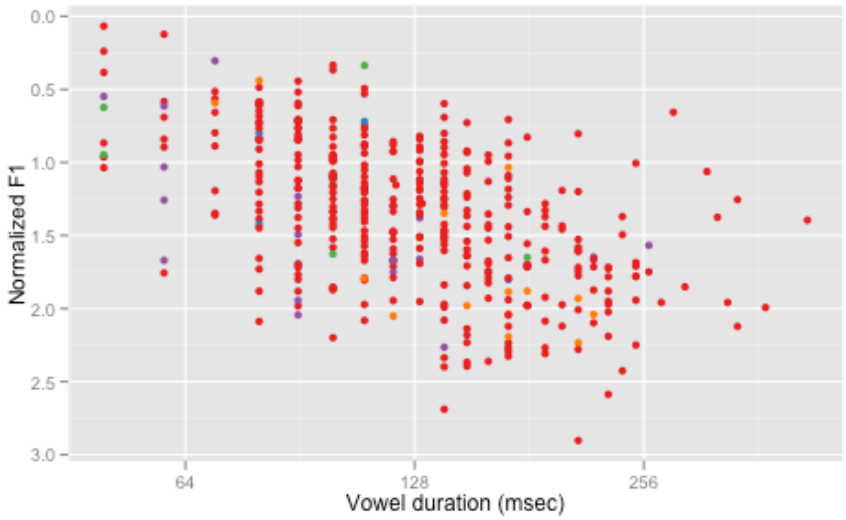
You can also place the legend into the plotting region by passing `legend.position` a vector of two numbers between 0 and 1. The represent how far proportionally along the x and y axes, respectively, the legend should go.

```
ggplot(I_jean, aes(x=Dur_msec, y=F1.n, color = Word))+
  geom_point()+
  scale_x_continuous(trans = "log2")+
  scale_y_reverse()+
  scale_color_brewer(palette = "Set1")+
  ylab("Normalized F1")+
  xlab("Vowel duration (msec)")+
  opts(legend.position = c(0.9,0.5))
```



And, if you want to suppress the legend all together, you can pass `none` to `legend.position`.

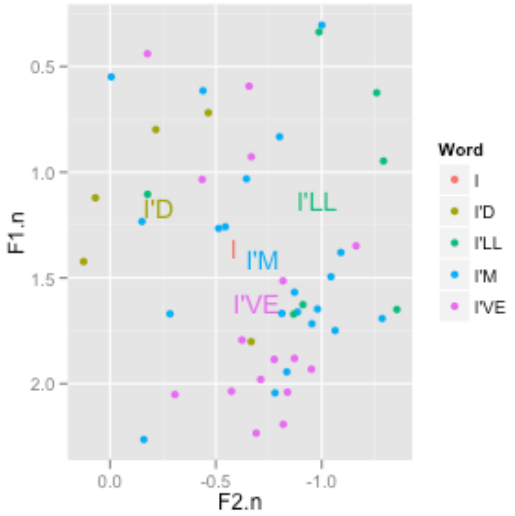
```
ggplot(I_jean, aes(x=Dur_msec, y=F1.n, color = Word))+
  geom_point()+
  scale_x_continuous(trans = "log2")+
  scale_y_reverse()+
  scale_color_brewer(palette = "Set1")+
  ylab("Normalized F1")+
  xlab("Vowel duration (msec)")+
  opts(legend.position = "none")
```



Building plots

```
library(plyr)
word_means <- ddply(I_jean, .(Word), numcolwise(mean))

ggplot(I_subset, aes(F2.n, F1.n, color = Word)) +
  geom_point()+
  geom_text(data = word_means,
            aes(label = Word),
            show_guide = F)+
  scale_y_reverse()+
  scale_x_reverse()+
  coord_fixed()
```



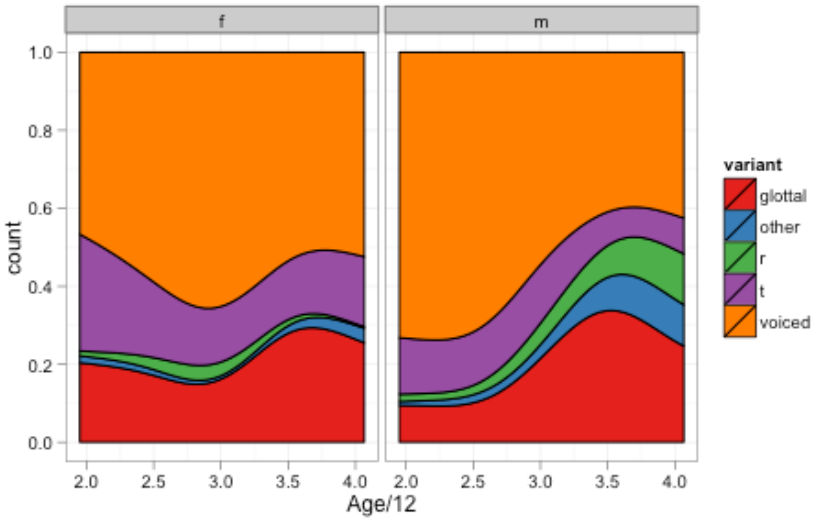
This is a plot of the data published in Foulkes, Docherty & Watt (2005), Appendix B: “Variant Usage by Mothers in CDS, Word-Final Intersonorant Context”.

```
fdw <- read.delim("http://bit.ly/fdw_2005")
head(fdw)
```

```
##    Child Sex  Age Cohort  N variable value
## 1  Alice   f 47.8    4;0  6        t     3
## 2  Alice   f 47.8    4;0  6        r     0
## 3  Alice   f 47.8    4;0  6  glottal     2
## 4  Alice   f 47.8    4;0  6   voiced     1
## 5  Alice   f 47.8    4;0  6    other     0
## 6 Alison   f 31.0    2;6 63        t     5
```

```
fdw <- ddply(fdw, .(Child), transform, prob = value/sum(value))
```

```
ggplot(fdw, aes(Age/12, fill = variable)) +
  geom_density(aes(weight = prob, y = ..count..), position = "fill")+
  facet_wrap(~Sex)+
  scale_fill_brewer(name = "variant", palette = "Set1")+
  theme_bw()
```


f
m
1.0
0.8
0.6
0.4
0.2
0.0
count
2.0
2.5
3.0
3.5
4.0
Age/12
variant
glottal
other
r
t
voiced