Data Visualization, Conditional Means, Smoothing, TimeSeries, and ggplot, oh my!

Josh Clinton

9/21/2021

Motivating questions:

- How do we work with dates as a variable?
- How do we work with data over time (time-series)?
- What is the principle of smoothing?
- How can I become famous by analyzing political polling and making cool figures? (That report uncertainty!)

New Skills:

- `ggplot()`
- user-built functions

Principles of Data Visualization

For better or worse, humans are visual creatures. Our brain is wired to try to find patterns in what we observe and we often attempt to understand the world around us by telling stories about how the world works. The tools of data science can go hand-in-hand with these tendencies, but our work must also account for these tendencies when trying to provide an accurate characterization of a relationship.

Visualization is a uniquely powerful way of communicating insights to others because it provides you, the data science, with the ability to summarize and characterize the relationship of interest. But with this power comes great responsibility as you can use those abilities to mislead rather than inform depending on the choices that you make. Because visuals are often more readily accessible than tables and statistical results, visualization can often have a powerful impact on how data are interpreted and understood.

There are many excellent books on principles of visualization – see, for example, the work by Edward Tufte – but for the purposes of this class we can identify a few general principles should guide our visualizations.

First, data always exists in a context and visualizations should put the results in a proper context. *Make sure your visualizations focus attention on the comparisons of interest.*If you are interested in the relationship between two variables then do show that relationship using the data you have! But don't show extraneous information that can district from the point you are trying to make with the visualization. For example, we often see political results being reported using a map. But the geographic location of a state is not always of relevance or interest. For example, if you are interested in how the support for a candidate in the polls compares to the vote that the candidate receives in an election, do you really need information about where the state is located geographically? No. Moreover, using a map to try to display that information – like we see all the time! – likely subtracts from our ability to focus on the relationship of primary interest because we are naturally attracted to the shape and arrangement of states rather that the values of the variables of interest. Moreover, because we process data in relation to what we are shown, we may tend to evaluate the

data looking at geographically proximate states even though that is irrelevant for determining how accurate the polls were.

Second, don't try to do too much with your visualization. Having too much information is as bad as having too little.If the reader is confused by the amount of information being displayed – or if you are using hard-to-distinguish colors, points, and fonts then you are failing at communicating the relationships of interest. Simplify and clarify!

Third, if you are tying to assess whether an effect is large or small, make sure to provide enough context for people to understand the variation and helps people understand the importance of what you are showing. Visualizations should help make the point abut the relations and also be informative about the distribution of the data and/or the nature of a relationship. Scale and axes matter for conveying this information.

Fourth, as much as possible, your visualizations should be stand-alone. It is optimistic to assume that your readers will do their proper due diligence and read the accompanying text. Instead, assume that a reader is only able to observe your figure. Does it make sense? Or, at a minimum, does it not mislead your readers?

Fifth, don't use a visualization if a visualization is uninformative. If there is no variation, then there is no need for a visualization.

To make it more concrete, here are some "preachy" rules:

- Always title your graph and use informative axes labels – no variable names! If you show the graph to a non-expert they should be able to say what is being plotted in accessible and understandable language.
- Always scale the x and y axes so that the variation is meaningful and reflects the variation of the actual data. Do not minimize the scale to make small differences seem larger and do not maximize the scale to make big differences appear small!
- Always be cognizant of how your visualization is going to be consumed. Do not use colors if it is intended to be printed out and read by people who have black and white printers. Be aware that some people (like me!) are red-green colorblind and cannot distinguish between some colors.
- Each graph should make at least 1 point, and probably not more than 3. If the visualization does not make at least 1 point then the graph has no point and it is unneccesary. If you are trying to make more than 3 points in a single visualization the visualization is likely too complicated and you may end up confusing your readers.
- 3D-visualizations are almost always horrific - avoid. Unless you have a 3D printer, the visualizations are consumed on a 2D medium. Trying to include an extra dimension almost always results in a decrease in comprehension as you are forcing the reader to make interpolations in their brain to project the 3rd-dimension. Don't risk this error.
- Pie charts are bad - absolutely avoid. The goal of a visualization is to make differences clear, but it is harder for the brain to calculate the relative area of a circle than it is to compare the height of a rectangle. Barplots (or histograms) are always superior to pie charts - especially as the number of "slices" increases.

Visualization as Description

The question we are goign to work through is a descriptive one – how did the average national support for Biden and Trump vary over time using the public polls that were performed and released? How should we measure "average support" over time when there are not polls being released every day? How can we quantify our uncertainty about the support for Biden and Trump under the assumption that a random sample of voters are being interviewed in each poll?

We are not going to focus on the *why* – which is essential for understanding – but we are going to start with the initial task of describing the empirical regularity. Put differently, what does the world look like?

We often start with description because knowing what the world looks like helps us with the next step of understanding why. It is often hard to hypothesize and determine why something occurs the way it does without first getting a sense of how the object of interest varies (over time, over units, over both).

More concretely, what we are going to focus on is the question of how the national popular vote in the 2020 presidential election between Biden and Trump changed over time according to the publically released polls. For this lesson we are going us polling data that I collected as Chair of the American Association of Public Opinion Task Force on the Performance of Pre-election polls. The data consists of every public poll that was reported and released on the national presidential vote between Biden and Trump. We are going to start with polls pertaining to the national popular vote in this chapter, but we will pivot to state level polls in later lectures.

Let's begin by loading the data and taking a look at the variables that are included using `glimpse`. Recall that the first thing we want to do in any data science project is to get a sense of what the data is that we are working with.

```
load(file="data/Pres2020.PV.Rdata")
glimpse(Pres2020.PV)
```

```
## Rows: 528
## Columns: 16
## $ poll.id     <dbl> 1942, 1941, 1940, 1939, 1938, 1937, 1936, 1935, 1934, 1933~
## $ Geography   <chr> "NAT", "NAT", "NAT", "NAT", "NAT", "NAT", "NAT", "NAT", "N~
## $ Poll        <chr> "Economist/YouGov", "Research Co.", "Ipsos", "Swayable", "~
## $ StartDate   <chr> "10/31/2020", "10/31/2020", "10/29/2020", "11/1/2020", "11~
## $ EndDate     <chr> "11/2/2020", "11/2/2020", "11/2/2020", "11/1/2020", "11/1/~
## $ DaysinField <dbl> 3, 3, 5, 1, 1, 3, 3, 3, 4, 4, 2, 5, 5, 14, 2, 3, 3, 3, 3, ~
## $ MoE         <dbl> NA, 3.10, 3.70, 1.70, 3.20, NA, 1.00, NA, 2.20, 2.26, 2.50~
## $ Mode        <chr> "Online", "Online", "Online", "Online", NA, "Online", "Onl~
## $ SampleSize  <dbl> 1363, 974, 914, 5174, 1008, 1360, 799401, 8765, 3505, 1880~
## $ Biden       <dbl> 53, 53, 52, 52, 48, 53, 52, 53, 52, 52, 48, 50, 49, 54, 48~
## $ Trump       <dbl> 43, 44, 45, 46, 42, 43, 46, 41, 41, 42, 47, 39, 46, 43, 39~
## $ DemCertVote <dbl> 51, 51, 51, 51, 51, 51, 51, 51, 51, 51, 51, 51, 51, 51, 51~
## $ RepCertVote <dbl> 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47, 47~
## $ Winner      <chr> "Dem", "Dem", "Dem", "Dem", "Dem", "Dem", "Dem", "Dem", "D~
## $ Funded      <chr> "Economist", "Research Co.", "Reuters", "Swayable", "John ~
## $ Conducted   <chr> "YouGov", "Research Co.", "Ipsos", "Swayable", "John Zogby~
```

So we have a lot of data here – data on `{r} nrow(Pres2020.PV)` polls – including when the poll started polling (`StartDate`), when each poll ended polling (`EndDate`) how the poll interviewed respondents (`Mode`), how many respondents were interviewed (`SampleSize`), the poll result for Biden (`Biden`) and Trump (`Trump`), and who funded and conducted the poll. Some of the variables do not vary because we are looking at polls for the national popular vote (e.g., `Geography`, `DemCertVote`, `RepCertVote`) but we shall see their relevance when we consider state-level polling below.

One variable that we may be interested in is the difference between Biden and Trump in each poll. To calculate this, we can mutate the tibble and create a new variable called `margin`.

```
Pres2020.PV <-
  Pres2020.PV %>%
  mutate(margin = Biden - Trump)
```

When doing election work, dates are very important. Polls have historically been more accurate the closer they are to Election Day so we want to know when Election Day is. R has a series of functions that are related to dates, including the function called `as.date` that can change a string in a particular format into a date object. The code below takes the US-standard month/date/year format and converts it to a date to define Election Day 2016 and Election Day 2020.

```
election.day <- as.Date("11/3/2020", "%m/%d/%Y")
election.day16 <- as.Date("11/8/2016", "%m/%d/%Y")
```

So what we have now done is to create two new objects correponding to Election Day 2020 (`election.day`) and Election Day 2016 (`election.day16`). Why in the world would we do this? Well now we can use the built-in calendar features to do calculations using the dates. So if we wanted to see how many days were between the 2016 and 2020 election we can use the date objects we just created to determine this. In particular:

```
election.day - election.day16
```

```
## Time difference of 1456 days
```

So let's use this as.Date function to change the string variables in our data to date objects. This will allow us to calculate how far each poll's ending date is from Election day – as well as how many days each poll was in the field trying to contact respondents (i.e., the differenece between the End date nad the start date.)

```
Pres2020.PV$EndDate <- as.Date(Pres2020.PV$EndDate, "%m/%d/%Y")
Pres2020.PV$StartDate <- as.Date(Pres2020.PV$StartDate, "%m/%d/%Y")
class(Pres2020.PV$EndDate)
```

```
## [1] "Date"
```

Now that we have the date of the 2020 election defined as a date object we can use mutate to create a variable that denotes how many days each poll was from Election Day. To do so we can use:

```
Pres2020.PV <- Pres2020.PV %>%
  mutate(DaysToED = election.day - EndDate)
```

Plotting the distribution of a Single Variable: Bar graphs in `ggplot`

So now let's look at the nature of 2020 pre-election polling. In particular, what does the distribution of public opinion polls look like over time? Because we want to describe the support for Biden and Trump during the 2020 election we have to start by getting a sense of how much data we actually have. This will help us determine what we can do. If, for example, we only have a few polls – or if there are long periods of time without polls – then we may want to think about the implications of that for how we analyze and visualize the patterns.

To motivate the desire for visualization, consider the alternative representation of the data using the tools we have learned earlier. To get the number of polls being concluded on each day we simply need to `count` the number of polls for each unique value of `DaysToED` using:

```
Pres2020.PV %>%
  count(DaysToED)
```

```
## # A tibble: 179 x 2
##    DaysToED     n
##    <drtn>   <int>
##  1  1 days      5
##  2  2 days      9
##  3  3 days      6
##  4  5 days      6
##  5  6 days      8
##  6  7 days      7
##  7  8 days      9
##  8  9 days      9
##  9 10 days      3
```

```
## 10 11 days      1
## # ... with 169 more rows
```

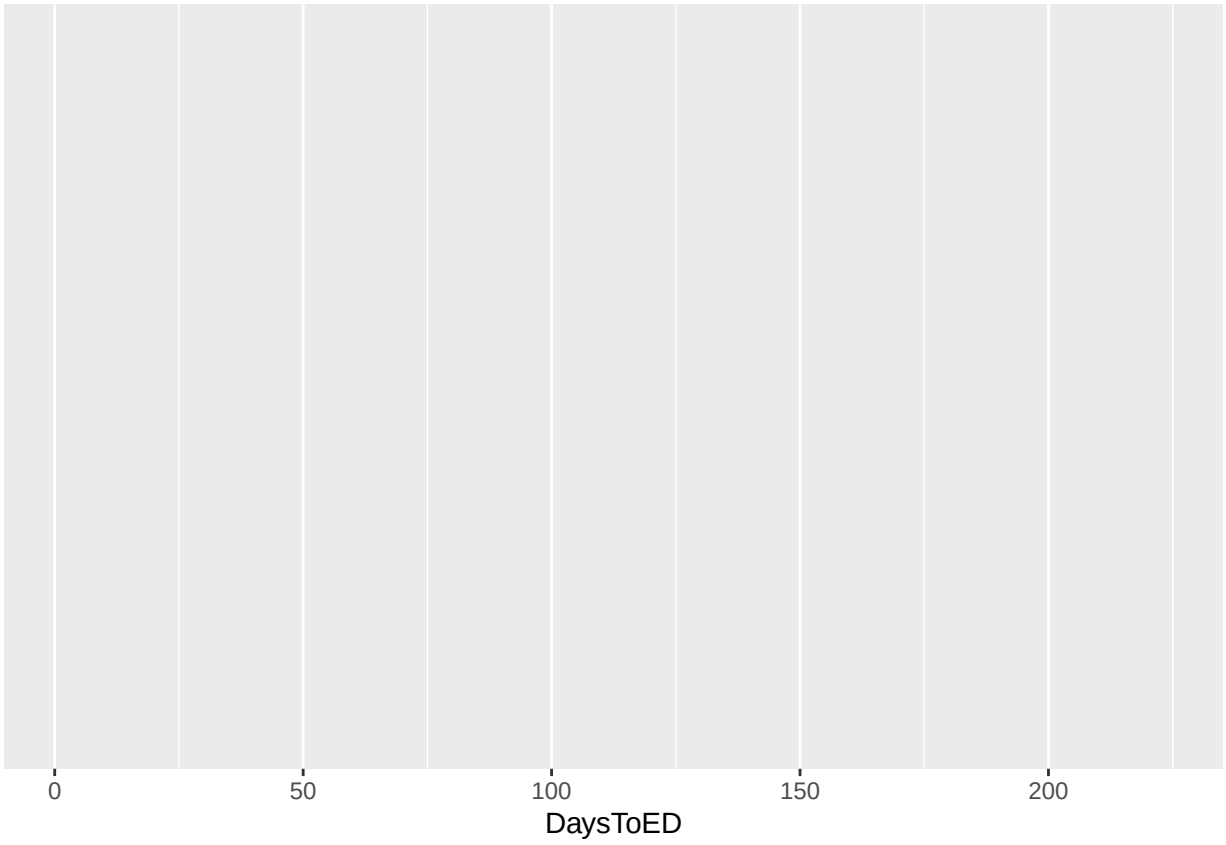
As is immediately clear, it is very difficult to be able to detect a pattern from this – in part because the table of counts is nearly as long as the original tibble (as it has `{r} nrow(Pres2020.PV %>% count(DaysToED)` rows!)

To visualize the patterns we are going to use `ggplot`. As with nearly everything in R, `ggplot` creates graphs as objects that can then be manipulated and extended. We will see this shortly, but let's start with the basic syntax.

Because we can have multiple tibbles in memory, when we call `ggplot` to produce a plot we need to tell it what tibble we are using and also what variables we are going to analyze. To identify the variables, we need to use the `aes`thetic function to list the variable name in the tibble being plotted. You can specify both an x variable (to be plotted along the x-axis which is also the horizontal part of the graph) and also a y variable to define the y-axis (vertical part of the graph). Here we only want to plot the distribution of polls being done for each possible day prior to Election Day. Calling this function will not actually plot any data. At this point, the command will just create the plotting canvas to which we can add additional elements.

```
Pres2020.PV %>%
    ggplot(aes(x = DaysToED))
```

```
## Don't know how to automatically pick scale for object of type difftime. Defaulting to continuous.
```



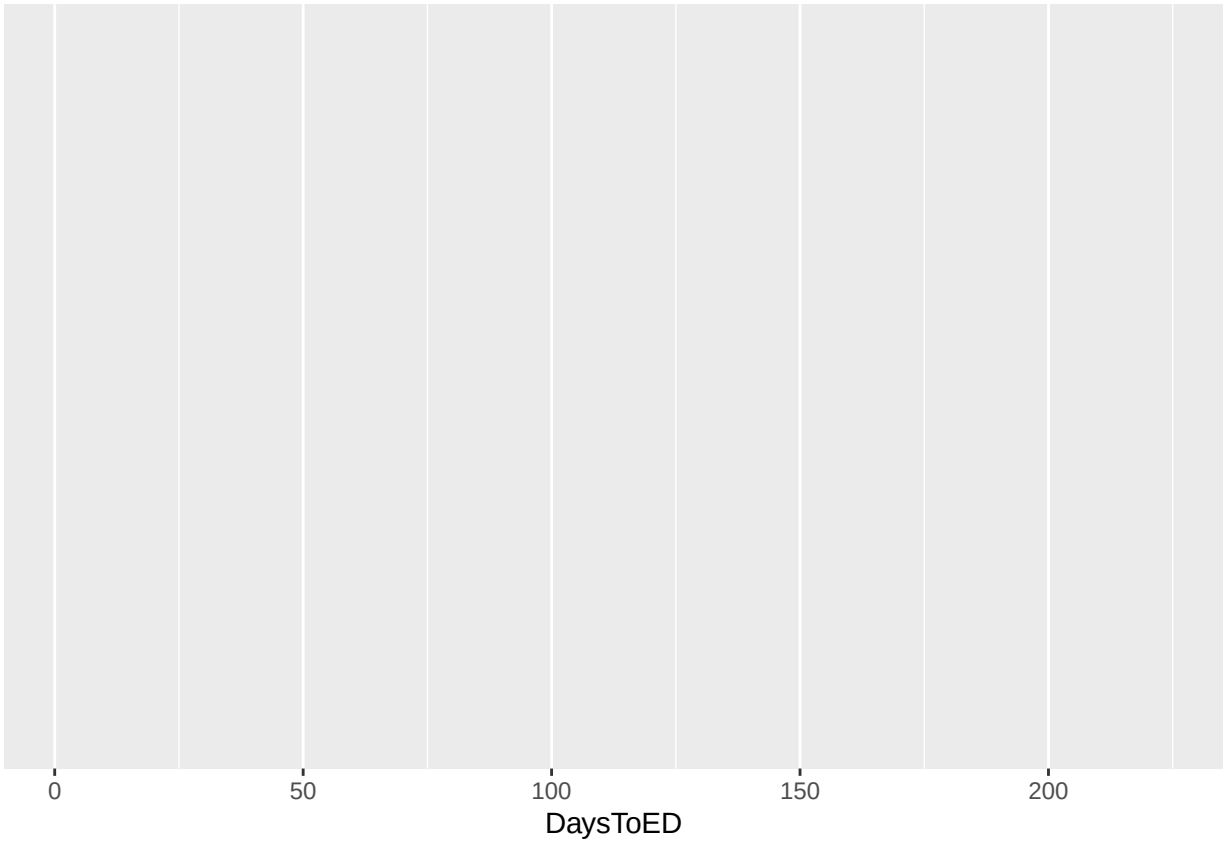
Now we can also define this as an object. So we can define the object `p` as our plot. Notice that when we write the code in this way that nothing will be plotted until we call the object `p` we have created to the screen.

```
p <- Pres2020.PV %>%
    ggplot(aes(x = DaysToED))
```

So if want to see this again, we need to call p:

```
p
```

```
## Don't know how to automatically pick scale for object of type difftime. Defaulting to continuous.
```



So why would we want to do that? Well, this allows us to build a plot sequentially by adding elements without having to retype previously defined commands. This is more than just a way to save time (and typing), but it also helps protect ourselves against our future self. The more our code relies on multiple identical code snippets the more likely we are to make an error because if ever we have to change the code we have to make sure to change it wherever it appears. If we define it once and create an object based on that snippet then we only need to manipulate that little bit if we change our mind.

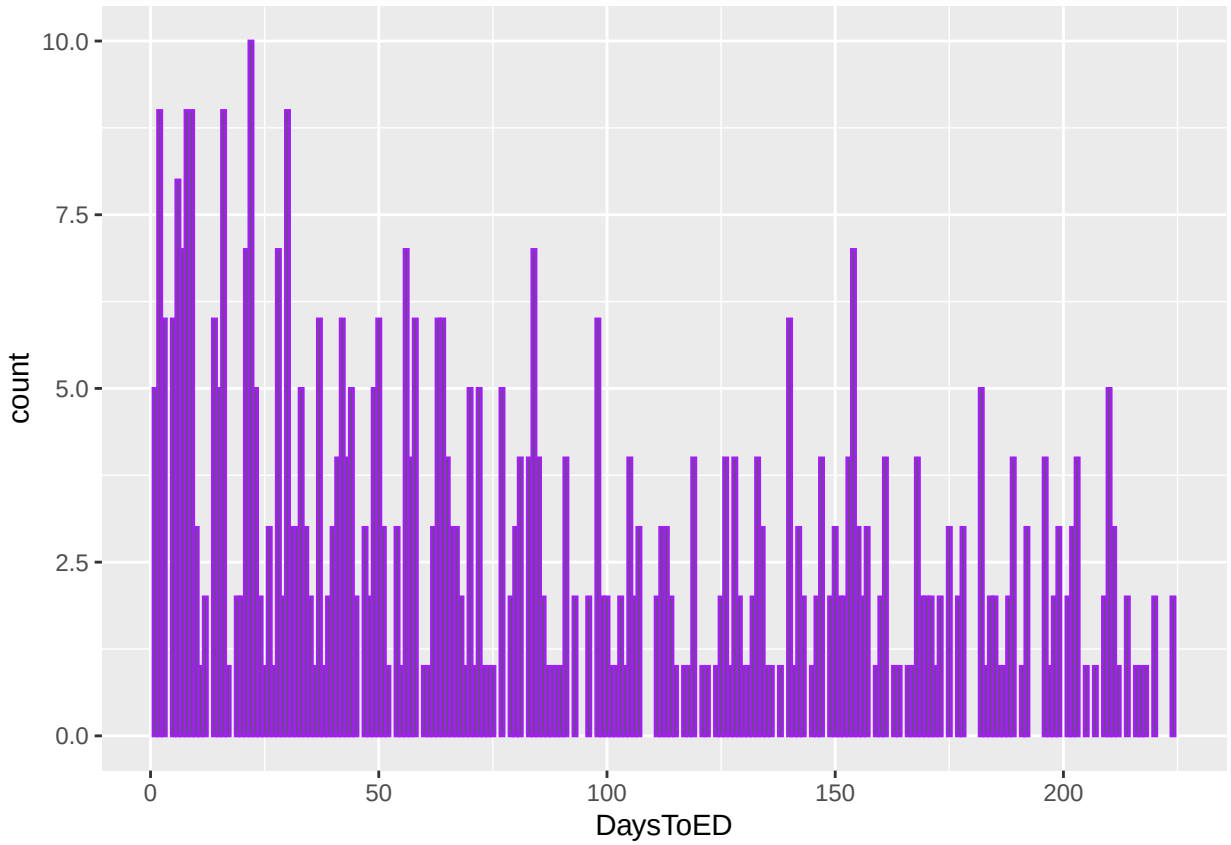
So let's use this to start to fill in the canvas we created in p. `ggplot` has a bunch of graphical elements that can be added to a graphical canvass. Here we want to graphically depict how many polls were concluded at each point in time.

The type of visualization that we want to use will depend on the type of data that we are working with. Here we are working with a count of the number of polls that are ending on each day of the campaign. Since this is going to produce a count, we want to use a graph for categorical data. In this case, we want to use a bar-graph where the height of the bar being plotted designates the number of polls concluding on each day.

So let's create a new ggplot that includes bars indicating the number of polls – using the color purple because we can. And to print what we did to the screen, let's call the new ggplot object `p2` to the screen.

```
p2 <- p +
  geom_bar(color="PURPLE")

p2
```

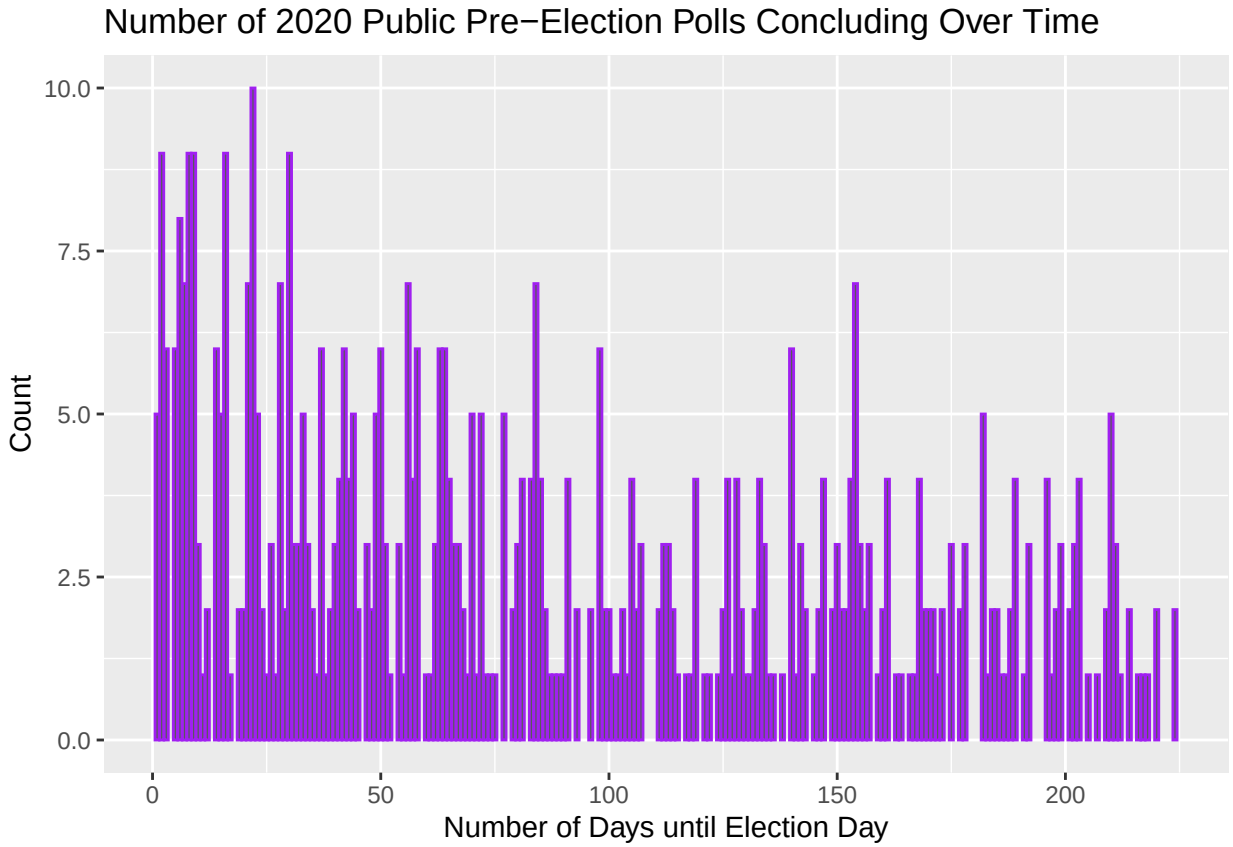
```
## Don't know how to automatically pick scale for object of type difftime. Defaulting to continuous.
```



Nice! But the graph is incomplete. The goal of data science is to make every figure self-contained. That is, it should be possible to interpret your graph by looking at it. This is important to minimize the potential for misinterpretation. To do this we want to create graphs that are labelled using the `labs` command – including a title that describes what is being plotted and also labels on the x and y axis to make it clear what each axis means.

```
p3 <- p2 +   labs(title = "Number of 2020 Public Pre-Election Polls Concluding Over Time") +
  labs(x = "Number of Days until Election Day") +
  labs(y = "Count")

p3
```

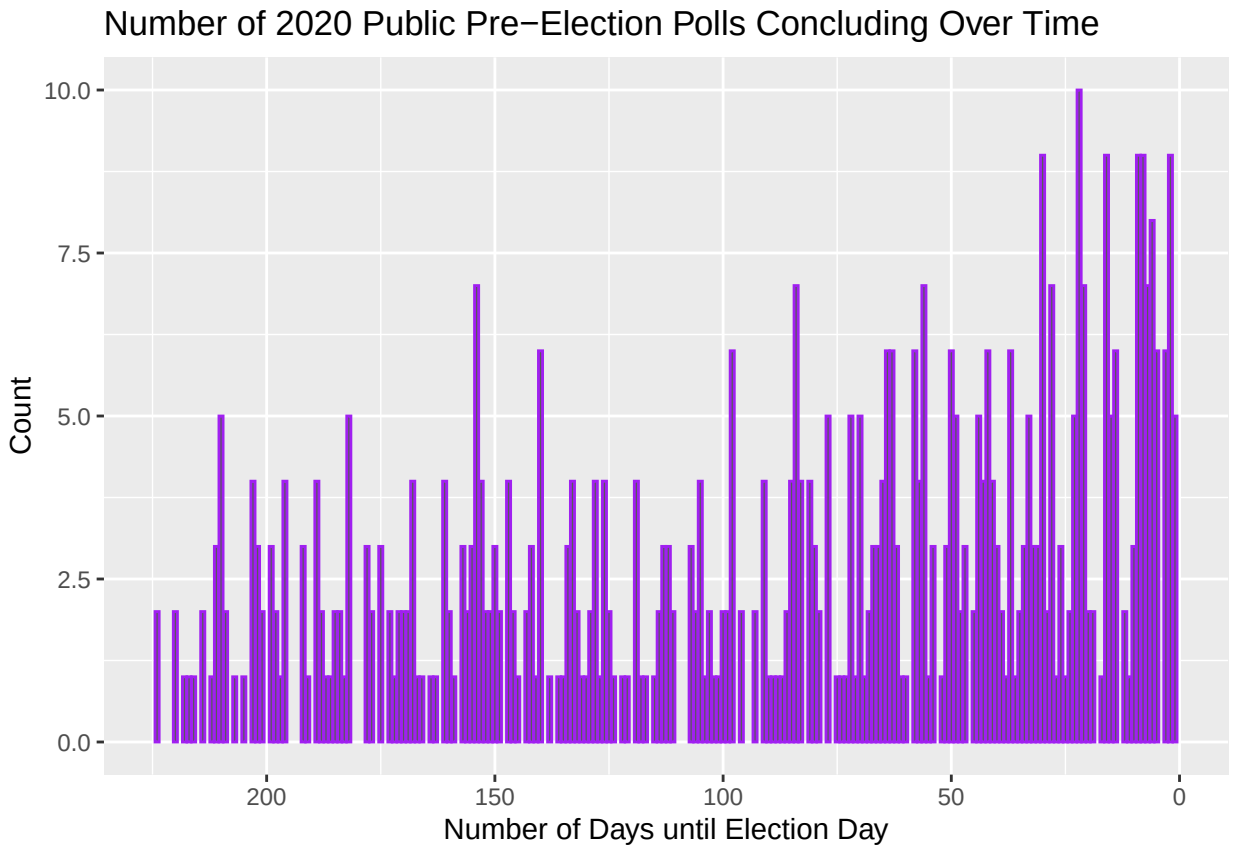
```
## Don't know how to automatically pick scale for object of type difftime. Defaulting to continuous.
```



Looking good! One final thing is that when we look at the graph we realize that the graph is a bit weird in that R naturally orders the x-axis from low to high values. But the data we are plotting is the number of days between the final day of data collection in the poll and Election Day. So lower values indicate dates that are later in the year. You can make an argument either way, but to preserve the meaning of time so that later dates appear to the left, we may want to "flip" the x-axis so that polls done in early spring are on the left and polls done in early November appear on the right.

Thankfully `ggplot` has commands to flip the x-axis (`scale_x_reverse()`) and also the y-axis (`scale_y_reverse)` that we can use to do precisely this. All we need to do is to call this function to let R know to flip the scale. We do NOT need to do anything with the actual variables because all we are doing is plotting the values of the variable in a differnet order. Here we are just going to call it to the screen.

```
p3 + scale_x_reverse()
```



So how do we save this as a file? First we need to choose a file format. We typically use a PDF, but we can also use JPEG or PNG and the code is similar. We begin by calling a function

```
pdf(file="MyGraph.pdf")
p3 + scale_x_reverse()
dev.off()
```

Plotting the Relationship between A Continuous and a Categorical Group Variable: The Boxplot and Violin Plot

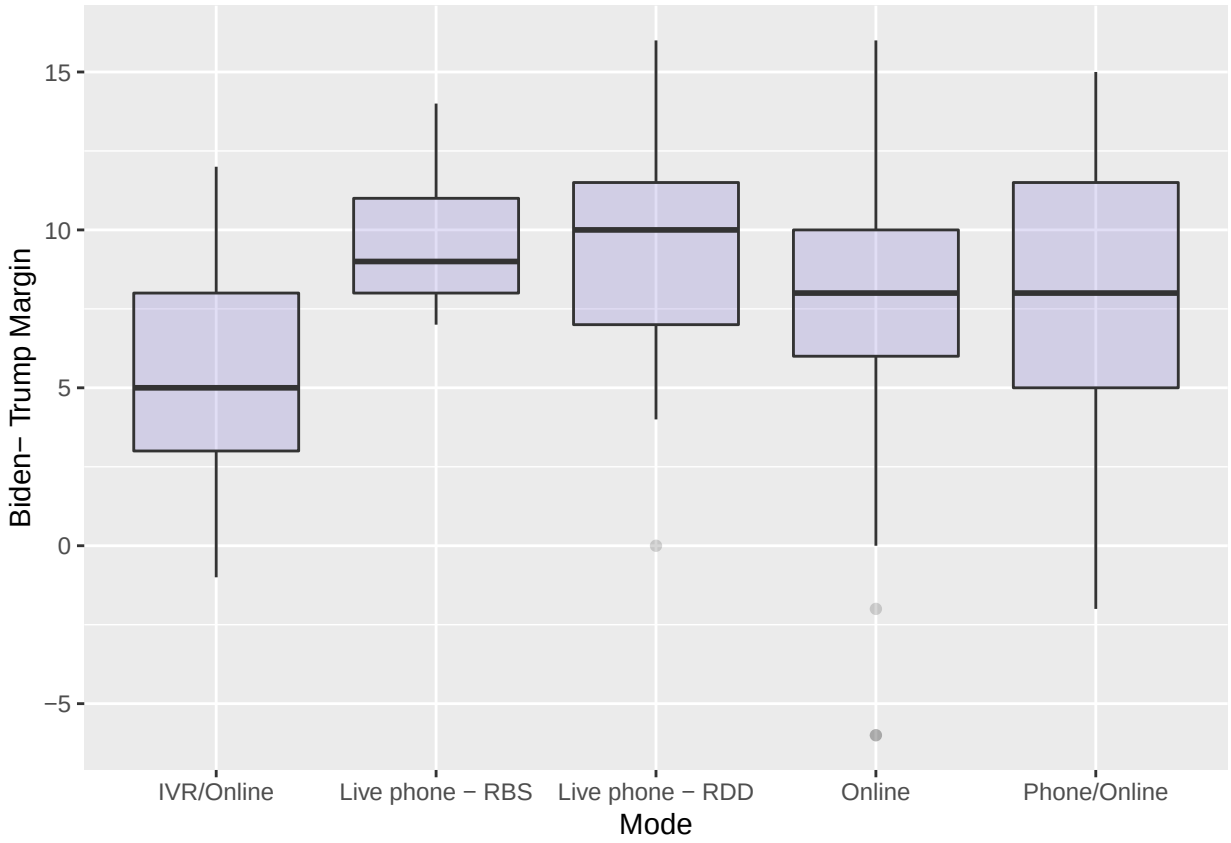
Suppose that we wanted to see how the margin varied depending on the type of poll being done. That is, do the results of some types of polls favor Biden more than others? One concern with polling is that different voters may be accessible using different methods of interviewing.The voters you are likely to reach on a landline telephone may be very different from the ones that you reach online or via text messaging.

When we are comparing how the Biden-Trump margins vary by method of polling a `boxplot` is often a useful tool for comparing how the margin compares across modes. To do so we need a continuous variable – here `margin` and a factor-variable that describes group membership – here `Mode` – that we want to use to compare the `margin`.

So in the code snippet we filter out survey modes with a small number of polls, and then we add labels and a `geom_boxplot` object that is filled with the color blue and which is slightly transparent (`alpha=.2`).

```
Pres2020.PV %>%
  filter(Mode != "IVR" & Mode != "Online/Text" & Mode != "Phone - unknown" & Mode != "NA") %>%
  ggplot(aes(x = as.factor(Mode), y = margin)) +
    xlab("Mode") +
```

```
    ylab("Biden- Trump Margin") +
    geom_boxplot(fill = "slateblue", alpha = 0.2)
```



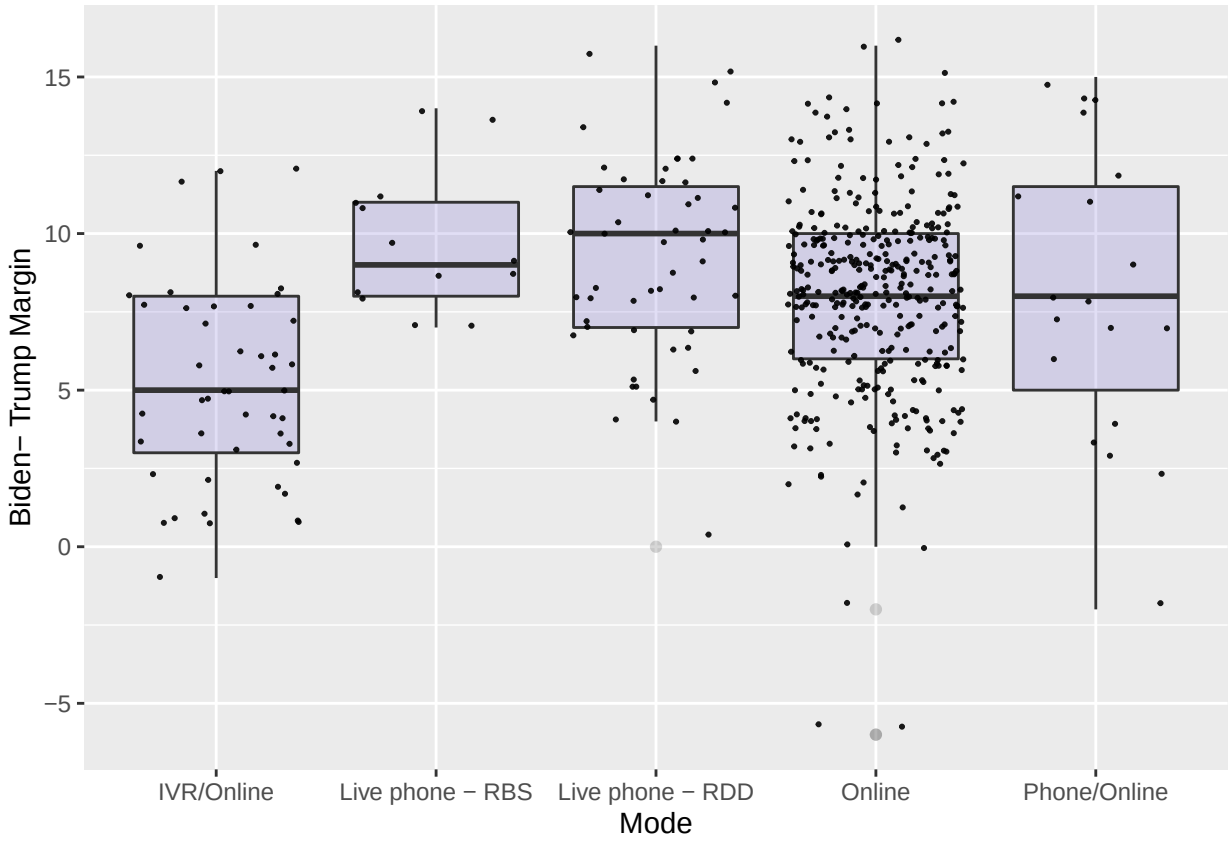
The boxplot shows several useful pieces of information. The horizontal lines in the middle of the box is the median value – here the median margin in each mode. The height of the box denotes the range of the data spanning the 25th and 75th percentiles (sometimes called the "interquartile range"). That is, if we ordered the margin within each type of poll, the bottom of the box is the location of the 25th percentile of the lowest margin and the top of the box indicates the location of the 75th percentile of margins within each type of poll. The vertical lines associated with each box indicate the range of values that span 1.5 × the difference in the 75th and 25th percentiles, and the dots that are plotted indicate polls that are "outliers" – i.e., very different from the rest of the data. Thus, there is at least one online poll that had Trump winning the national popular vote by more than 5 percentage points.

One issue is that there is no way to determine how much data is associated with each of the modes. How many polls are associated with each method of interviewing respondents? This can matter because we may trust the estimates from some types of polls more than others — e.g., if there are 1000 polls conducted Online and only 3 polls using live phone interviewers for respondents reached via random digit dialing we may think that the variation in the former is more precisely estimated than the variation of the latter.

Relatedly, it is also hard to to determine the distribution of the data from a boxplot. For example, the boxplot for a variable that took on two values with equal frequency would be hard to distinguish from a boxplot for a variable whose values were uniformly spaced. In other words, the boxplot for `Phone/Online` could reflect the possibility that there are an equal number of polls with a margin of 11 and 5 and no other margin, or it could reflect that there are equal number of polls at 5,6,7,8,9,10,11. These are very different worlds – the former is a world in which the polls produce only two values – but a boxplot does not display any information about the actual variation in the data beyong the percentiles being plotted.

One way to include this information is to include the actual datapoints in the boxplot.

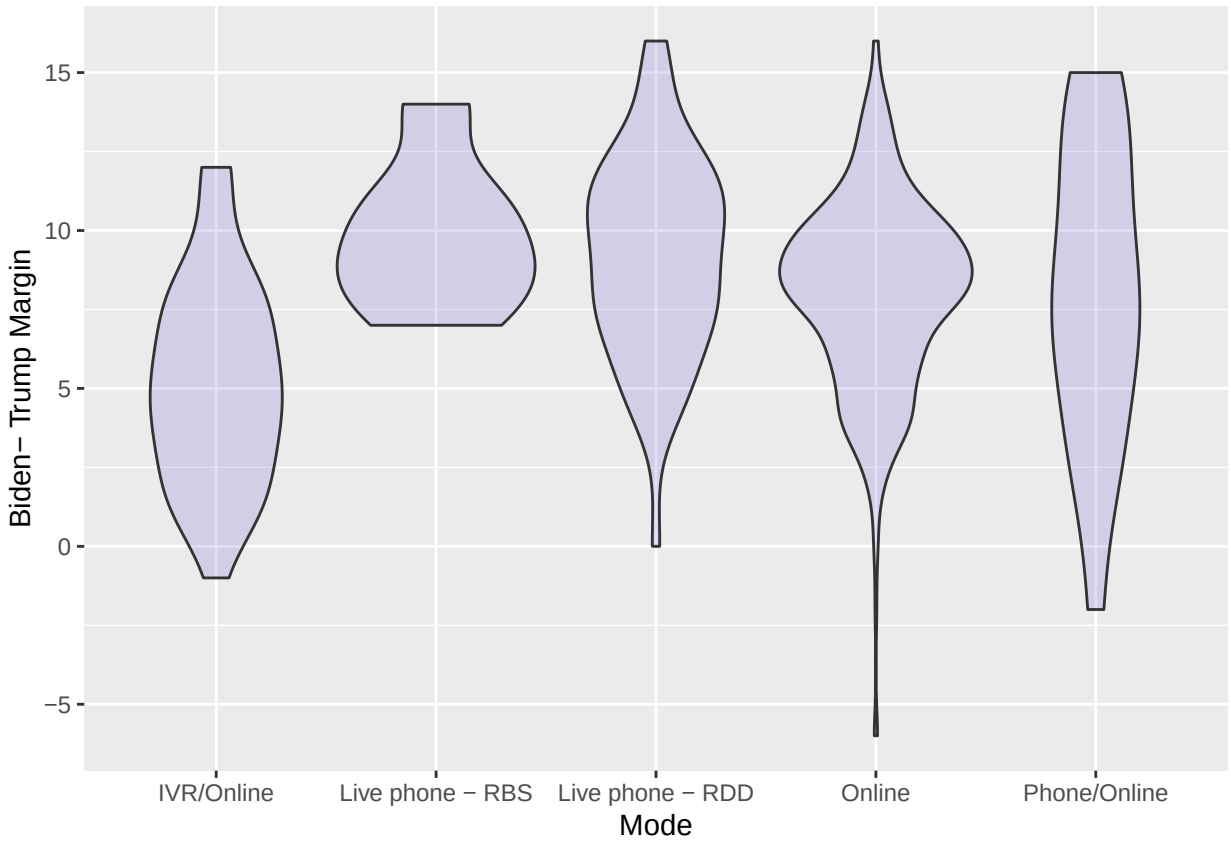
```
Pres2020.PV %>%
  filter(Mode != "IVR" & Mode != "Online/Text" & Mode != "Phone - unknown" & Mode != "NA") %>%
  ggplot(aes(x=as.factor(Mode), y=margin)) +
    xlab("Mode") +
    ylab("Biden- Trump Margin") +
    geom_boxplot(fill="slateblue", alpha=0.2) +
    geom_jitter(color="black", size=0.4, alpha=0.9)
```



What we have done is to use `geom_jitter` to add the points. What is "jitter"? Basically all it does is add a bit of random noise to each value being plotted to make sure that the points are not all directly on top of one another. Here what it does is to separate the points within each mode along the x-axis even though those differences are meaningless – i.e., within "IVR/Online" polls there is no meaning to points that are to the left or right of one another. Here is where you have to use your judgement – is adding the "jitter" and introducing error into the raw data (and potential confusion because there is no meaning for the variation along the x-axis within group) offset by the ability "see" every data point.

Another way to compare the distribution of a continuous variable is by using a "violin" plot is useful. Whereas a boxplot creates a box that obscures the distribution of data within each category, a "violin" plot reveals the distribution of of the continuous variable (here `margin`) by each group so that the thicker the violin is the more data is located near that point and the thinner the violin the less the data. To produce a violin plot we simply need to change `geom_boxplot` to `geom_violin`.

```
Pres2020.PV %>%
  filter(Mode != "IVR" & Mode != "Online/Text" & Mode != "Phone - unknown" & Mode != "NA") %>%
  ggplot(aes(x=as.factor(Mode), y=margin)) +
    xlab("Mode") +
    ylab("Biden- Trump Margin") +
    geom_violin(fill="slateblue",alpha=0.2)
```



So the "violin" associated with "Online" polls reveals that there are a few polls that have Trump in the lead – hence the extended "tail." The pear-shaped violin for "Live phone - RBS" reveals that most of the polls had a marin between 7.5 and 10, and relative few had a margin more than 12. The long-skinny "Phone/Online" reveals that there are roughly an equally number of polls with margins at every value between 5 and 15.

Plotting the Relationship Between Two Continuous Variables: Scatterplots in `ggplot`

While interesting for telling how many polls we have to analyze that were being done on each day what we really want to do is to use those polls to characterize how the race between Biden and Trump was changing over the course of the election. To do so we need to create a new variable that consists of all of the days between the date of the first poll in our dataset and Election Day. This new variable – all_dates – is therefore a sequence that runs from the earliest poll (which we can determine using the fact that R can work with dates and determine the earliest Ending Date by identifying the minimum End date using the min function) to election day. Since both are date objects we want to tell R that the sequence is going to vary by days (rather than numbers) so we can preserve the date characteristics of the variable.

```
all_dates <- seq(min(Pres2020.PV$EndDate), election.day, by = "days")
```

Now why did we create this variable separately from the tibble of polls? Note that when we create this variable it enters into our Environment independently of our polling tibble. We did so because it is in a differnet shape than our polling dataset. Our tibble of polls is organized so that each observation is a different poll that was done. So we have 528 unique polls to analyze. The columns of the tibble refer to different characteristics associated with each poll – including the support that each indicates for Biden and Trump. But all_dates is a variable that is 225 observations long consisting of a sequence that goes from the earliest poll to Election Day. The units in these objects are completely different and it makes no sense to try to somehow combine them in a single object. Thankfully in R we can leave them separate and work with them

as needed.

To characterize what happened in the national popular vote over the course of the 2020 presidential election according to the publicly-released polls we are going to focus on how the difference in support for Biden and Trump varied. To so so we are going to use the `margin` variable we created earlier. By focusing on the margin we are going to characterize the extent to which Biden was ahead or behind of Trump at each point in the campaign.

Let's start by creating our canvass by defining a plotting object. Since we don't need the plot p anymore (we have already created it and saved it to a file), we are going to overwrite it. We are also going to create a canvass that is well-labelled to start using the `labs()` commands.

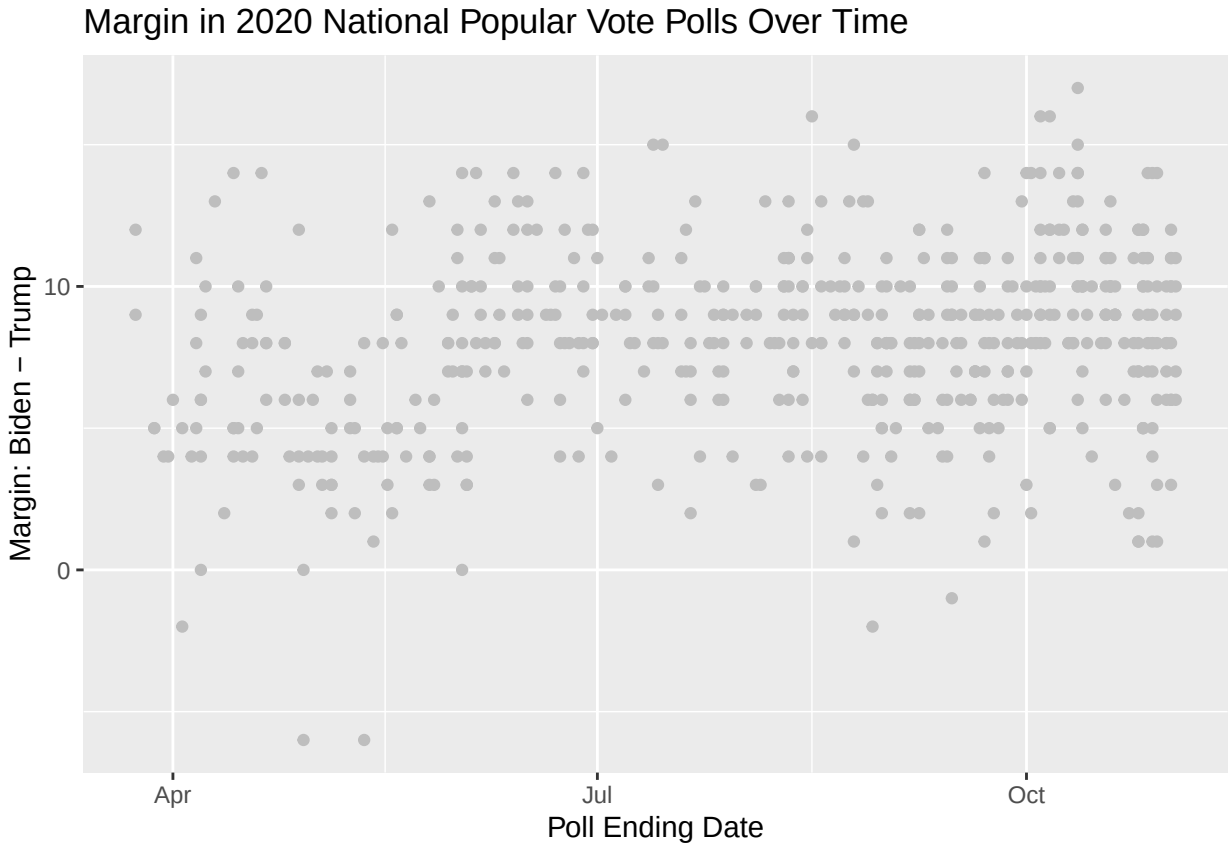
Unlike before, now we have defined both an x and a y variable in the aesthetic function. This is how we tell R what variables we want plotted on the x-axis and y-axis. Since we want to see how the margin between Trump and Biden changed over time, and we want to read time as going from "left to right" we are plotting by EndDate along the x-axis – so days that are later in the year are plotted on the right hand side of the plot. Higher values of the y-axis will have poll results more favorable to Biden and lower values on the y-axis will have poll results more favorable to Trump.

Of course we could totally flip this and have time along the y-axis and poll results along the x-axis. While somewhat a matter of personal choice, insofar as we tend to think of time as moving from "left to right" and poll results goign "up and down" it makes sense to have the plots match human intuitions about the concepts being plotted. If you try to get too cute or too disruptive than your visualization can distract from understanding by being too challenging to process. Your goal as a data scientist is to convey information efficiently and accurately.

```
margin.plot <- Pres2020.PV %>%
  ggplot(aes(x = EndDate, y = margin)) +
  labs(title="Margin in 2020 National Popular Vote Polls Over Time") +
  labs(y = "Margin: Biden - Trump") +
  labs(x = "Poll Ending Date")
```

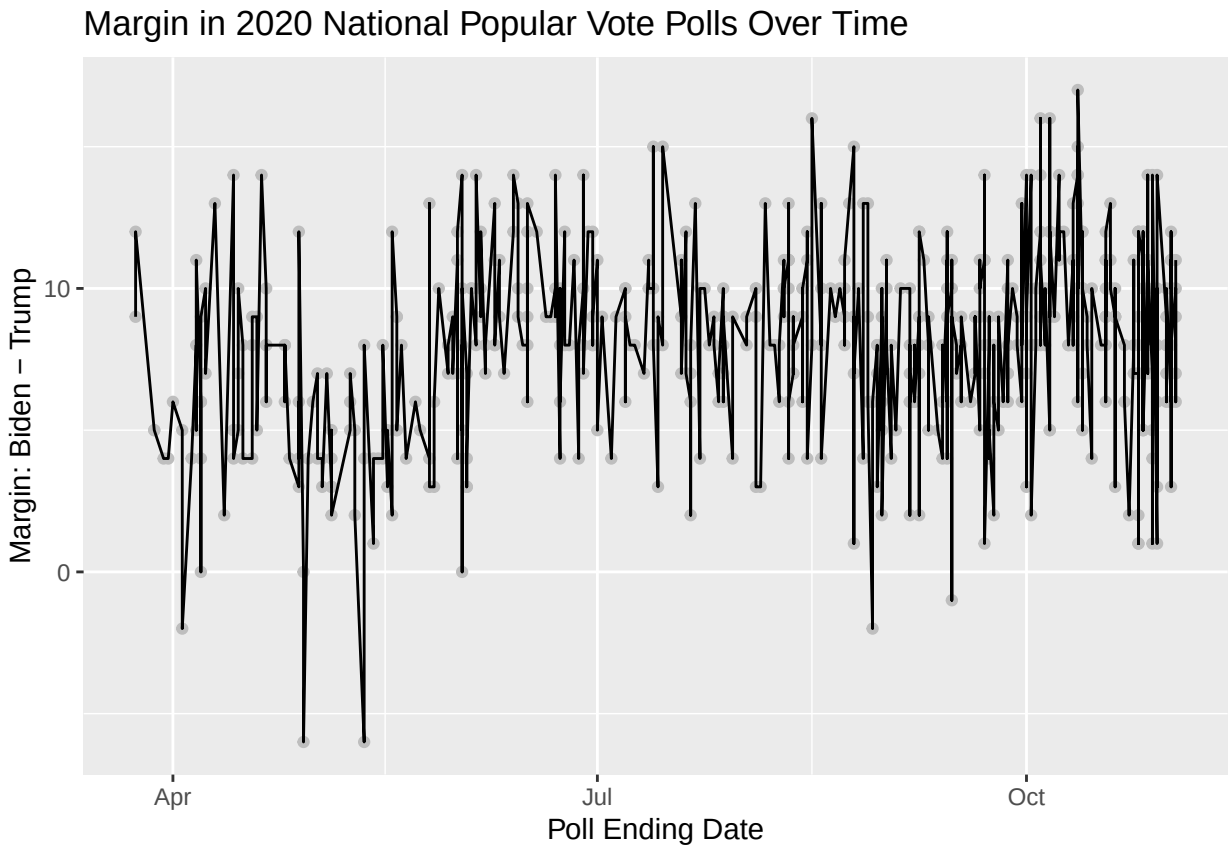
So now we want to add data to our canvass. To add the data points we use geom_point to include a points at locations (x,y) where x is the value of the x-variable (here `EndDate`) and y is the value of the y-variable (here `margin`). While you do not need to specify a color – the default is `BLACK` – let's choose `GREY` to illustrate that it can be done.

```
margin.plot  + geom_point(color = "GREY")
```



Either in addition to, or instead of, we can also include a line connecting all of the points. Note that this will create a line that connects every point being plotted from the "lowest" to the "highest". This can be useful when summarizing a trend over time if the trend is relatively stable, but it can also create more confusion than it is worth. Let's see what happens if we include it here...

```
margin.plot  + geom_point(color = "GREY") + geom_line()
```

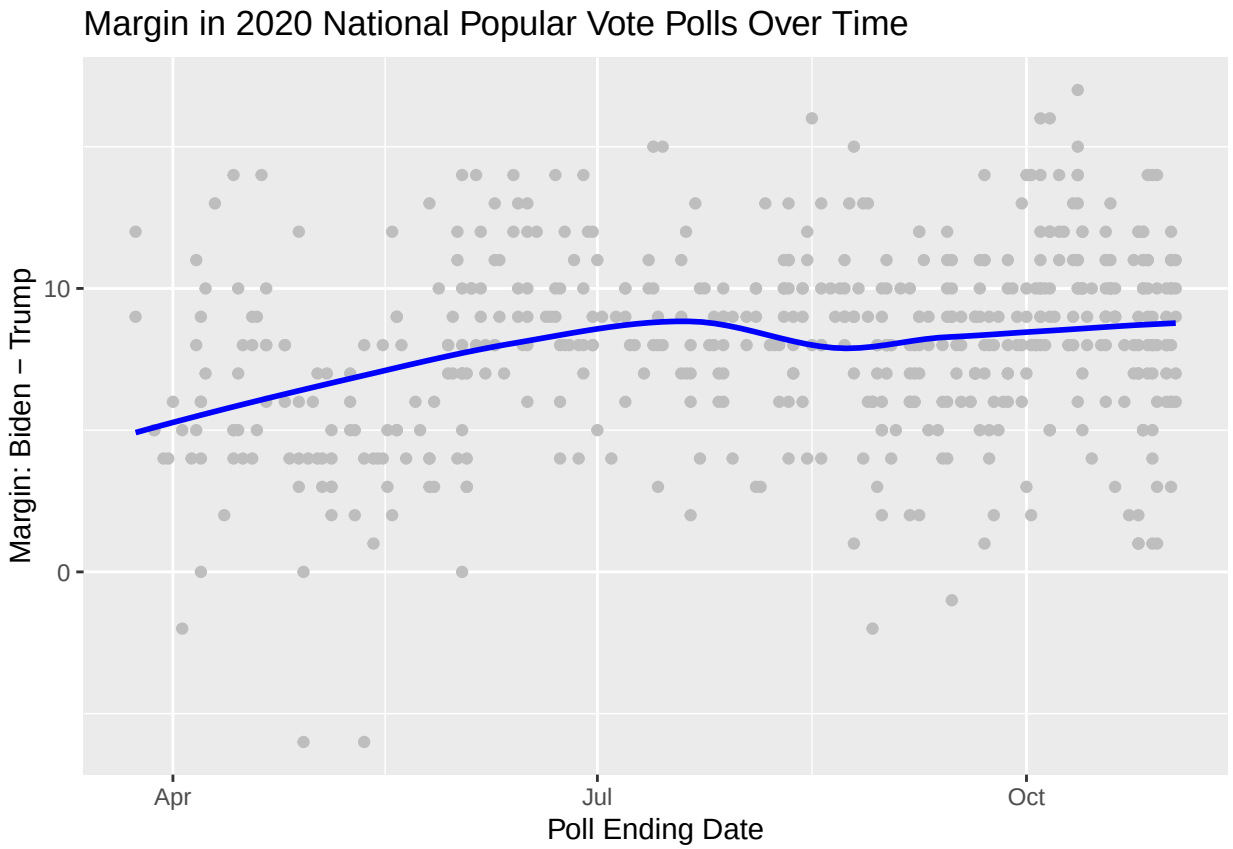


Ugly. In fact, adding that element makes the graph harder to interpret. Whereas your brain naturally interprets and interpolates from the scatterplot of the prior plot, when we add lines connecting all of the points we end up with a complete mess. The lines are jumping up and down – connecting every point – and this creates a jumble of confusion. The trend is far from smooth and the time series can jump around in response to a single poll result.

Now one thing we can do is to include a smoothed line that summarizes the average value across the x-axis. We will do this ourselves in just a bit, but R has a built in capacity to do so in ways to help summarize variation over time in a smooth way. To do so it focuses on the average margin using several nearby days.

```
margin.plot  + geom_point(color = "GREY")  + geom_smooth(color = "BLUE", se=F)
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



Plotting Multiple Time Series on the same plot: Scatterplots with multiple aesthetics in `ggplot`

Plotting the margin over time is interesting, but it is also limited in what it conveys. While it tells you how much Biden is leading Trump by over the course the the campaign – there was never a day where the polls released on that day suggested Trump would win the national popular vote! – we may also be interested in the support for Biden and Trump. Based on the margin it is hard to tell precisely how popular Biden is. While there are only two major party candidates running and you might think that you could use the fact that with two candidates and the difference between two candidates you can solve mathematically for the level of support that each candidate must have (e..g., if the margin is 10 points in a two candidate race this implies that the leading candidate must be winning 55 to 45 given that the percentages must sum to 100). Unfortunately, in polling the percentages in two-candidate races do not always sum to 100 because some indicate that they will vote for a third party candidate – see 2016! – and some pollsters allow respondents to choose response options such as "Don't Know". As a result, we cannot use the margin alone to determine what the polls imply about the level of support for each candidate. (Now if we wanted to focus on the level of support among those respondents who select either Trump or Biden then we could use the margin to determine what the implied level of support must be, but most polls report (and use) the average support rather than the two-candidate support.)

If we want to plot the time-series of Biden estimates and Trump estimates on the same plot we can use the power of `ggplot` to add the series separately. As always, let's start by defining and labeling our canvas.

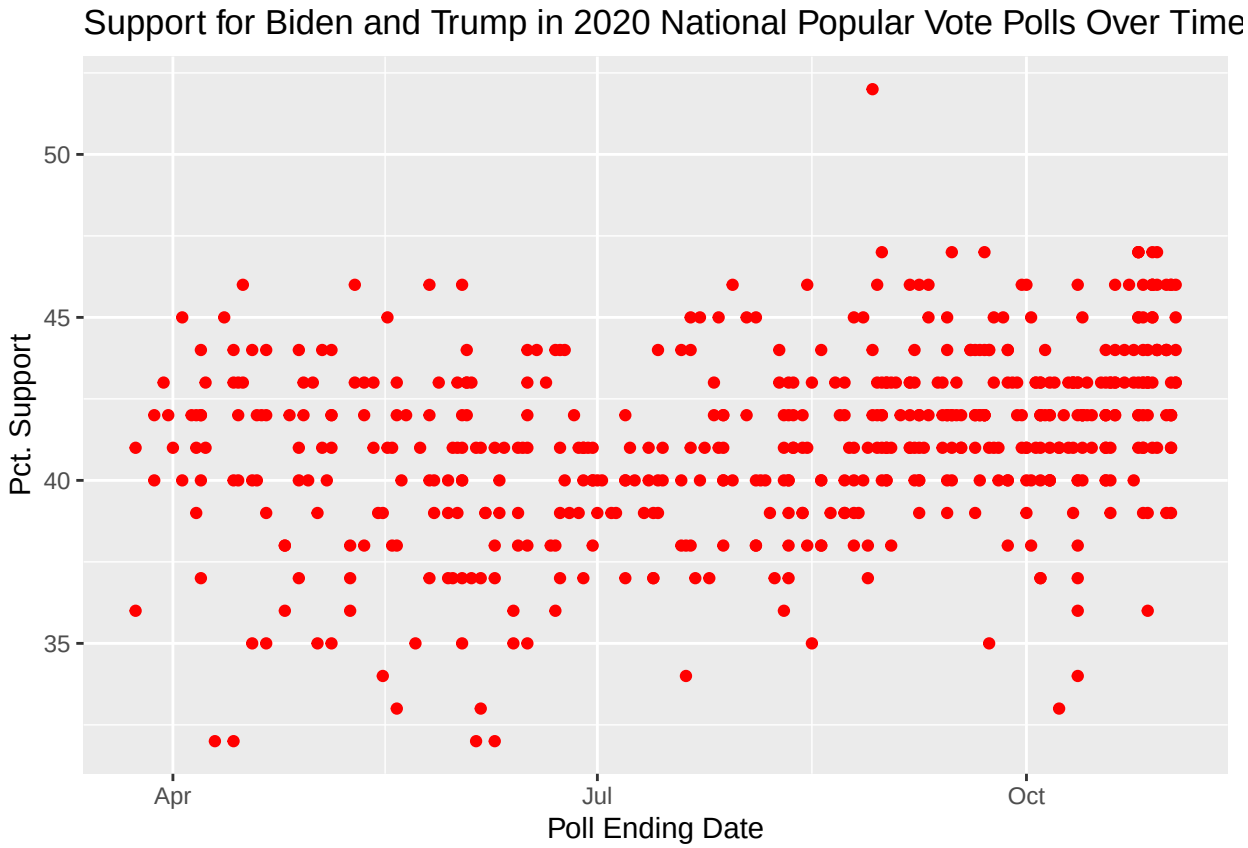
So the `ggplot` function identifies the data frame that we are going to use and then we include labels for the x and y axes.

```
BidenTrumpplot <- Pres2020.PV %>%
  ggplot()  +
  labs(title="Support for Biden and Trump in 2020 National Popular Vote Polls Over Time") +
```

```
  labs(y = "Pct. Support") +
  labs(x = "Poll Ending Date")
```

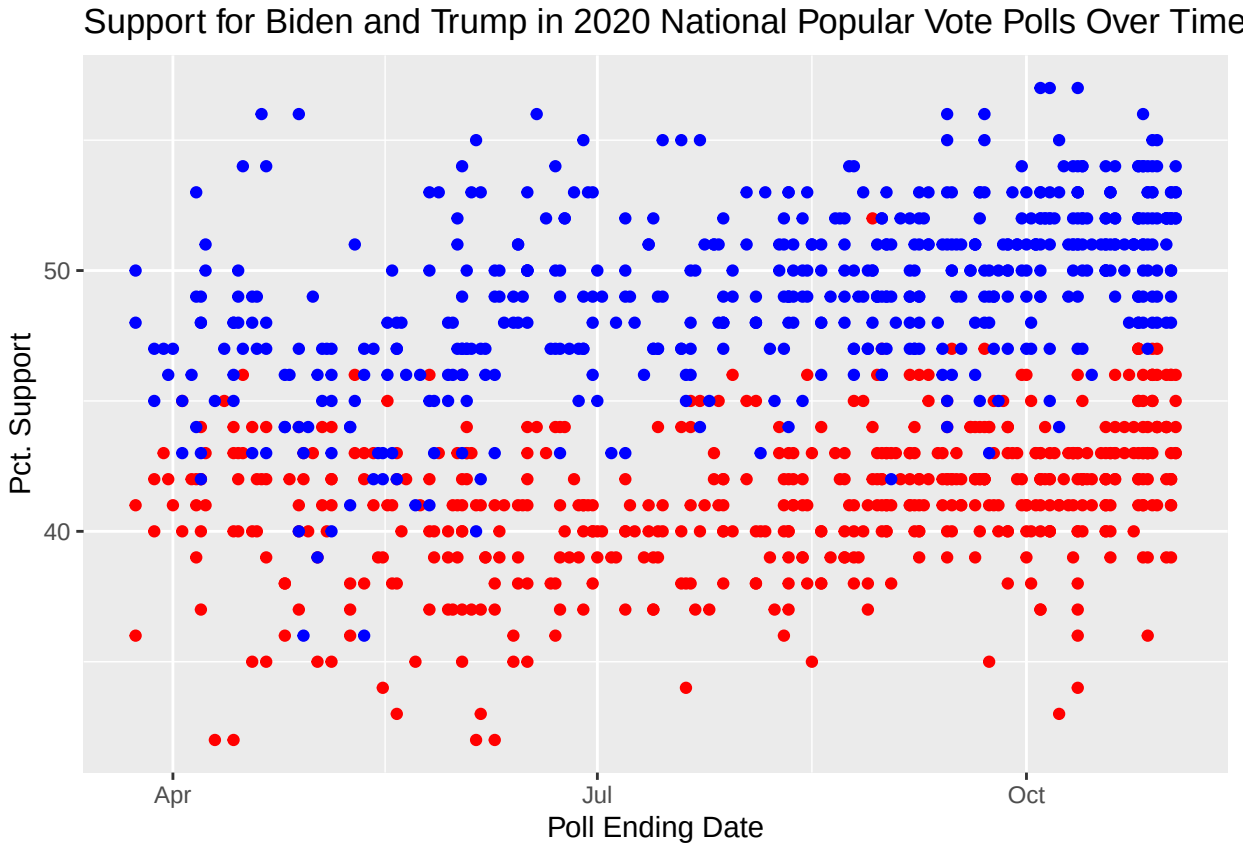
the `BidenTrumpplot` object is a blank canvass with labels. Now we want to add some points. Let's start by adding support for Trump. To do so we want to call `geom_point` and the aesthetic we are going to use is to plot the `EndDate` along the x-axis and the support for Trump according to the poll (`Trump`) along the y-axis. For clarity, we want these points plotted in red. Let's update our `BidenTrumpplot` with that addition and then print the plot object to the screen.

```
BidenTrumpplot <- BidenTrumpplot +
  geom_point(aes(x = EndDate, y = Trump), color = "red")
BidenTrumpplot
```



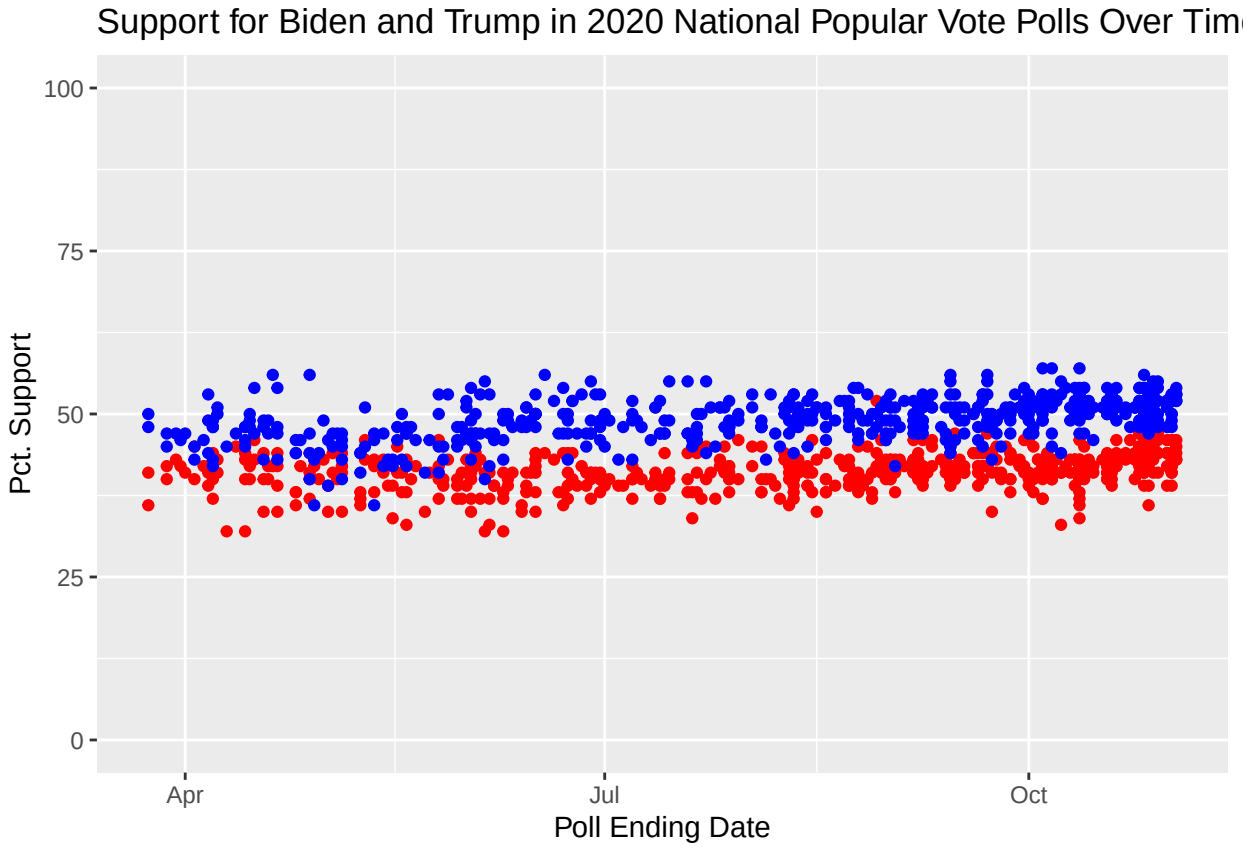
Now let's add the support for Biden in blue.

```
BidenTrumpplot <- BidenTrumpplot +
  geom_point(aes(x = EndDate, y = Biden), color = "blue")
BidenTrumpplot
```



Note that the y-axis scale of the plots has changed. R is choosing values to fit the scale automatically, but the scale it uses changes once we add the Biden percentages because they are so much higher. This is something to keep in mind – if you want readers to compare plots it is useful to define the axes yourself to be the same. It is very hard to compare graphs when the scale varies! We use the commands `xlim` or `ylim` to define the range. So if we wanted to have the y-axis range from 0 to 100 we would use:

```
BidenTrumpplot + ylim(0,100)
```

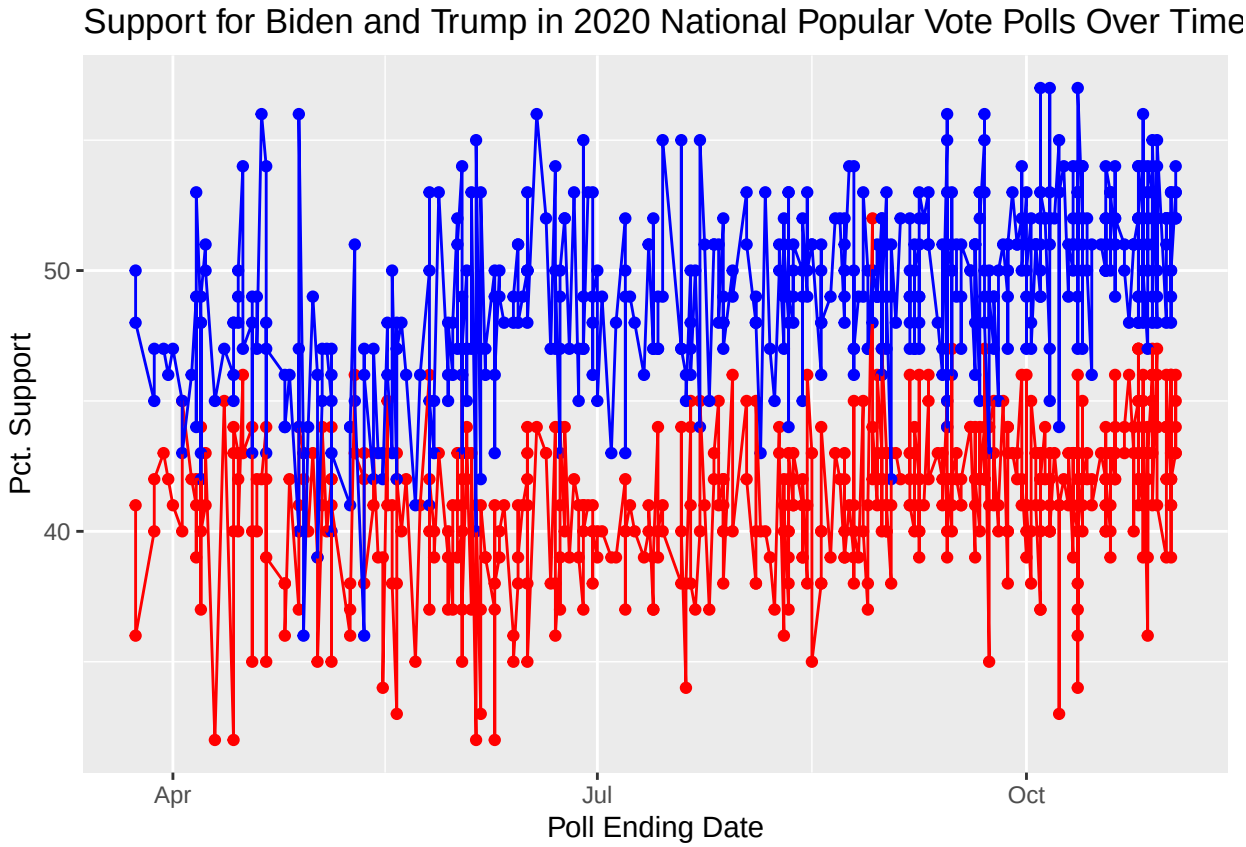


Notice the effect that this has on your reaction to the graph. In the original graph focused narrowly on the range containing the data the plot suggests that Biden is much higher than Trump across all time. When the range goes from 0 to 100 the amount of white space in the plot emphasizes the relatively closeness of Biden and Trump support – they are both near 50 rather than being closer to extreme values.

This is an important point. The data being plotted is exactly the same, but how the human mind tends to interpret the pattern being plotted can be affected by aspects such as the scale that is used. You always need to think about your reader and what the truthful representation of the data is. It is easy to use data to mislead - it is much harder to use it to inform!

Just as we can add separate point using different aesthetics when using geom_point, so too can we plot different lines using geom_line. The code to do so is exactly what you would think given the code used to produce the separate points:

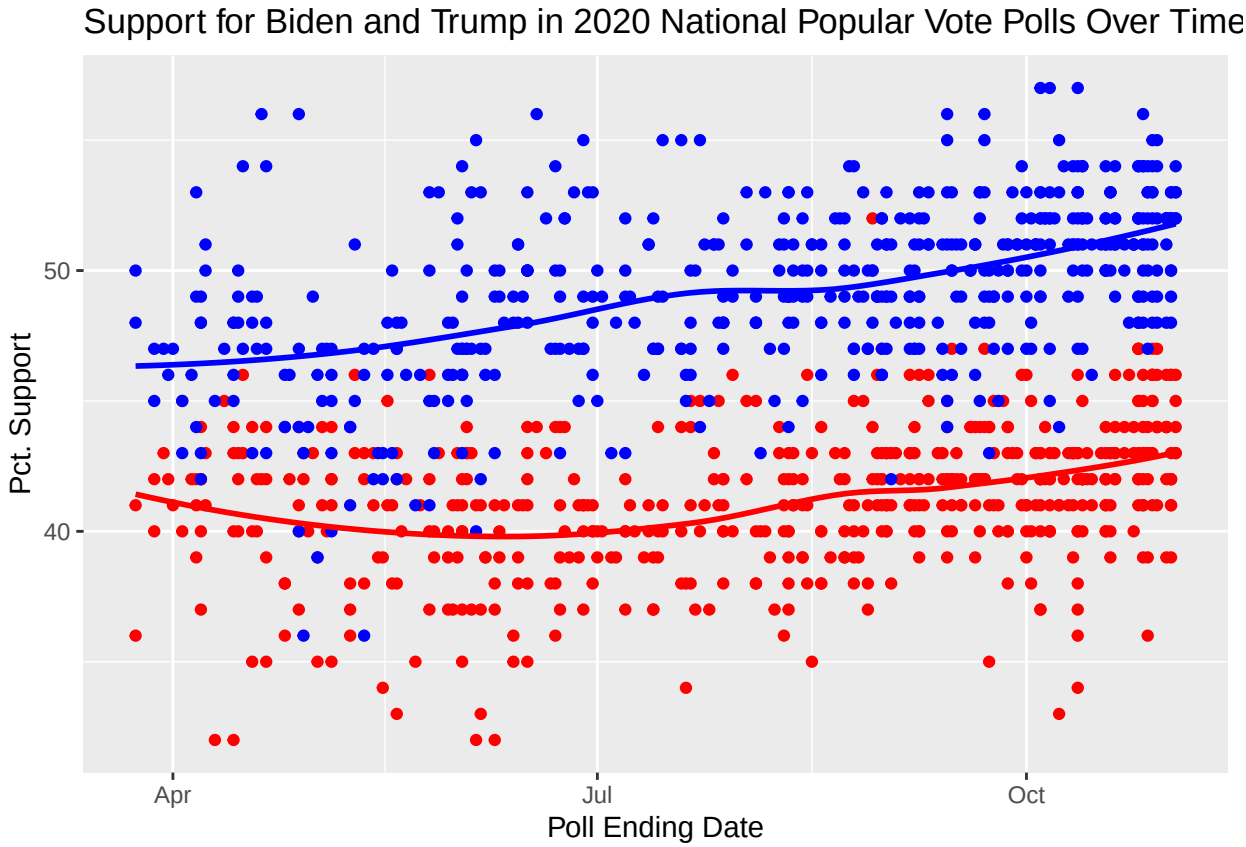
```
BidenTrumpplot +
  geom_line(aes(x = EndDate, y = Trump), color = "red") +
  geom_line(aes(x = EndDate, y = Biden), color = "blue")
```



While you can certainly do this, it is hard to argue that adding the lines improved the plot. There is just too much going on now...

```
Pres2020.PV %>%
  ggplot() +
  geom_point(aes(x = EndDate, y = Trump), color = "red") +
  geom_point(aes(x = EndDate, y = Biden), color = "blue")  +
  geom_smooth(aes(x = EndDate, y = Trump), color = "red",se=F) +
  geom_smooth(aes(x = EndDate, y = Biden), color = "blue",se=F) +
  labs(title="Support for Biden and Trump in 2020 National Popular Vote Polls Over Time") +
  labs(y = "Pct. Support") +
  labs(x = "Poll Ending Date")
```

```
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
## `geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



But it can make sense to try to identify a pattern over time. Is there a pattern that can be extracted from the data? One way to do this is to focus on plotting the average support for Biden and Trump rather than every poll result. The results were were just plotting were the results for each and every poll – so if there were multiple polls released on the same day there would be multiple points.

But that isn't really what we care about. If we have multiple polls released on the same day it may make sense to try to combine them and take the average result. So long as all of the polls are random samples of the electorate, we are better off if we have multiple polls released on a single day. As the Law of Large Numbers revealed, the mean of the multiple polls will get closer to the value as the number of polls increases.

Moreover, because it is unlikely that there is a poll released every day, we are going to take the average of all polls released either on the selected day or the day before. In other words, we are goign to "smooth" the data by taking the average of all poll results released over a 2 day window.

So to do this we need to do several steps. And we want to start by breaking things down to figure out what has to be done. We also want to do the calculation in a generalizable way so that we can tweak it to try different things. To begin we need to define how many days we want to use to select the polls that we are going to average. Rather than hard-coding this into the code, it is better to define this as a value that can can be referenced in the code. This make sense because when we change this number we only need to change it once in the code. We are going to start with looking at the average over two days so we will define Bandwidth to be 2.

Then what we want to do is to start at the earliest date and consider the dates one at a time up to Election Day. For each date, we want to identify all of the polls that occur on that day (and the 2 days prior) and then calculate the mean support for Biden and Trump for those polls. We want to save those means and then move to the next day.

Because we are taking the mean using the polls for several days and doing this for every day between the start of polling and Election Day this means that every poll will enter into our calculations three times – on

the day that it ends, on the day after it ends, and the second day after it ends. Becasue of this, we call the resulting time series a "smoothed" time series because the mean being calculated on each day shares the polls from the prior 2 days which limits how much the mean can jump around.

So let's put these words into action. We begin by defining the Bandwidth – how many days prior to a selected day we want to include when taking the polling average – and also a holding variable that will contain the means we calculate for each day. Here we define PC_avg to be a vector that is as long as the sequences of dates we are going to consider. We make it a "list" object because the output from our code will be in list format.

```
Bandwidth <- 2
PV_avg <- vector(length(all_dates), mode = "list") # holding variable
```

Now let's get into the actual code. Because we want to do the calculation for every day we are goign to need to use a loop that loops over the dates we created in `all_dates`. This means that we will start with the earliest poll and end the time series at the polls done closest to Election Day.

To keep things clear, we are goign to define the date object to be the date associated with the ith element in `all_dates`. (Technically we could skip this and just reference `all_dates[i]` whenever we use `date` in the code below, but we break it out for readability.) Note that date is going to be rewritten as we loop over all_dates. That is totally fine.

So now that we have selected the date we are going to use to select our polls via the loop, we now need to filter the tibble to select only the set of polls that were done in the time span of interest. To do this we subtract the ending date of every poll in the tibble from the date we are working with in the loop and we convert that to an integer (rather than a difference in days) for ease of analysis. To select all polls that were done within the 2 days prior to the selected date we want the concluding date of the poll to be on or before the selected date (i.e., `as.integer(EndDate - date) <= 0`) AND we want the concluding date of the poll to be within 2 days of the selected date. Since polls done prior to the date will have a negative value after subtracting off date, this means that the difference should greater than -2 (i.e. `as.integer(EndDate - date) > - Bandwidth`)).

Using this filtered data, we then can use the `summarize()` function to calculate the average support for Biden (which we define to be `Biden`) and the average support for Trump (`Trump`). What this means is that the object called `week_data` consists of a list of two values – `Biden` (the average support for Biden) and `Trump`. We then add the `date` value as another variable in `weekdata` using the code `week_data$date <- date` and we save the resulting object as the ith value in the `PV_avg` list. Because we are writing a list to the `PV_avg` list object we created, we index the ith element of that list using double-brackets `[[]]`. Note that we would use single brackets `[]` were we writing to a vector variable.

```
for (i in seq_along(all_dates)) {
  date <- all_dates[i]

    PV_avg[[i]] <- Pres2020.PV %>%
        filter(as.integer(EndDate - date) <= 0 &
           as.integer(EndDate - date) > - Bandwidth) %>%
    summarize(Biden = mean(Biden),
              Trump = mean(Trump)) %>%
      mutate(date = date)

}
```

The result of this is object `PV_avg`. If we take a look at what kind of object PV_avg is using the class command we can see that it is a list object. Lists are widely used in R, but they can be tricky when you first encounter them because they are more than a simple tibble and/or matrix of values.

```
class(PV_avg)
```

```
## [1] "list"
```

For example, if we take a look at the dimensions of `PV_age` we see that it reports that there are no dimensions. This is because even though it looks like a matrix to us, to R it is a list. You can see that by considering the first "observation" of `PV_avg` which returns a tibble of 1 row and 3 columns. (So it is a matrix, but just not one that we can access using traditional tools where we can select particular rows and columns! Trying `PV_avg[1,]` or `PV_avg[,1]` to select the first row and column, for example, will fail because of "incorrect number of dimensions").

```
dim(PV_avg)
```

```
## NULL
```

```
PV_avg[1]
```

```
## [[1]]
## # A tibble: 1 x 3
##   Biden Trump date
##   <dbl> <dbl> <date>
## 1    49  38.5 2020-03-24
```

```
PV_avg[[1]]
```

```
## # A tibble: 1 x 3
##   Biden Trump date
##   <dbl> <dbl> <date>
## 1    49  38.5 2020-03-24
```

To create a tibble matrix that we can work with normally we can use the `bind_rows` command to transform the list into a tibble. (There is also a `bind_cols` that binds columns into a tibble.) After we bind_rows, you can see that the class() of the object has changed to be a tibble (produced by a table), and that it has dimensions that we can now access using the normal indexing functions of R.

```
pop_vote_avg <- bind_rows(PV_avg)
class(pop_vote_avg)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
dim(pop_vote_avg)
```

```
## [1] 225   3
```

To confirm the contents of this object – something you should always do to make sure the code did what you think it did! – we can look at the first few rows of the resulting tibble using the head command.

```
head(pop_vote_avg)
```

```
## # A tibble: 6 x 3
##   Biden Trump date
##   <dbl> <dbl> <date>
## 1    49  38.5 2020-03-24
## 2    49  38.5 2020-03-25
## 3   NaN NaN   2020-03-26
## 4   NaN NaN   2020-03-27
## 5    46  41   2020-03-28
## 6    46  41   2020-03-29
```

Note that observations 3 and 4 are missing. Why? Recall that we are taking the average of polls in a three day window. If there are no polls being done in that period then there is nothing to average and we will get missing data as a result. What we can tell is that there was a poll on 3-24, but then another national

presidential polls was not done until 3-28. So when we look at the polling average for 3-26 and 3-27 there are no polls to be averaged. We will come back to the implications of this when choosing our smoothing bandwidth.

For future reference, note that we could also have transformed the object by binding based on columns. Rather than stacking the elements of the list `PV_avg` row-by-row, however, this will stack them column-by-column. The result is a very different looking tibble!

```
pop_vote_avg2 <- bind_cols(PV_avg)
```

```
## New names:
## * Biden -> Biden...1
## * Trump -> Trump...2
## * date -> date...3
## * Biden -> Biden...4
## * Trump -> Trump...5
## * ...
```

```
class(pop_vote_avg2)
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

```
dim(pop_vote_avg2)
```

```
## [1]   1 675
```

What have we done? Essentially we took the various list objects and we defined a new column for each entry. Whereas bind_rows adds each new list element to the "bottom" of the tibble being created, bind_cols will add each new list element (here containing 3 elements) to the "left" of the tibble. As a result, we end up with 665 variables and a single row. Moreover, because each element of the list had the same three objects – Biden, Trump, date – when creating new columns the `bind_cols` command has to rename them so that now we have Biden1, Biden2, . . . , BidenXXX.

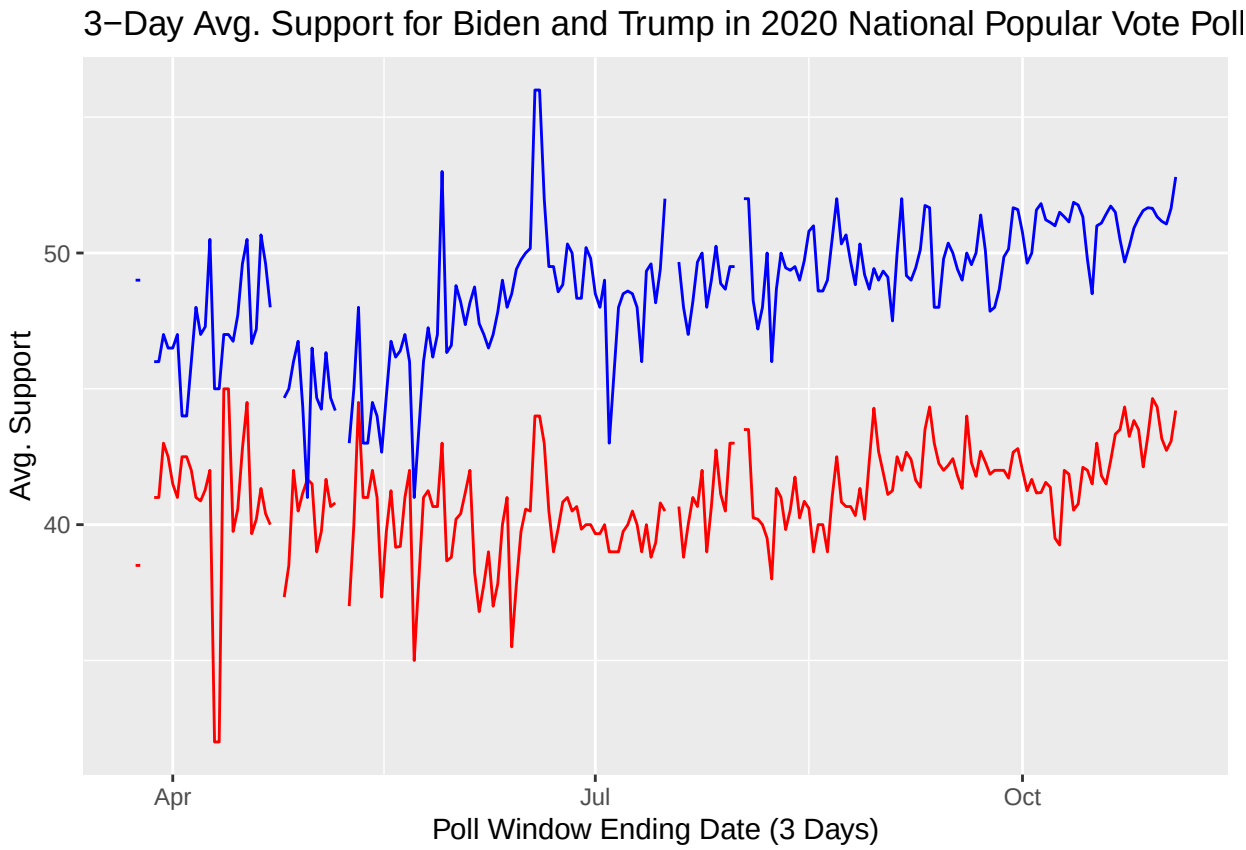
This is clearly not what we want, so let's use the tibble we created using `bind_rows` to analyze the patterns.

Before we were plotting the individual polls and there could be multiple polls each day. Now we have "smoothed" our data and each observation in the pop_vote_avg tibble is an average of the polls that were conducted in the three days ending on the specified date. Because each polling average is based on three days, it will share data with the polling averages being plotted before and after polling results – with the averages being plotted before and after.

So let's plot the smoothed lines over time to see what we have managed to produce. When calling `ggplot` we want to tell R that we are going to be referencing the tibble pop_vote_avg and we want to use the `geom_line()` commands to separately plot the smoothed polling average for Trump (in red) and Biden (in blue). (For now we are not going to include the scatterplot of individual polls from the `Pres2020.PV` tibble, but we will do so shortly!)

Because we are plotting a smoothed average, we want to plot a line rather than points. After all, the whole point of smoothing is to try to summarize and characterize the pattern in the underlying distribution of data points. So we will use a line to do so to help emphasize the trend over time – which is the question we are trying to answer: how does the support for Biden and Trump change over time?

```
pop_vote_avg %>% ggplot() +
  geom_line(aes(x = date, y = Trump), color = "red") +
  geom_line(aes(x = date, y = Biden), color = "blue")  +
  labs(title="3-Day Avg. Support for Biden and Trump in 2020 National Popular Vote Polls Over Time") +
  labs(y = "Avg. Support") +
  labs(x = "Poll Window Ending Date (3 Days)")
```

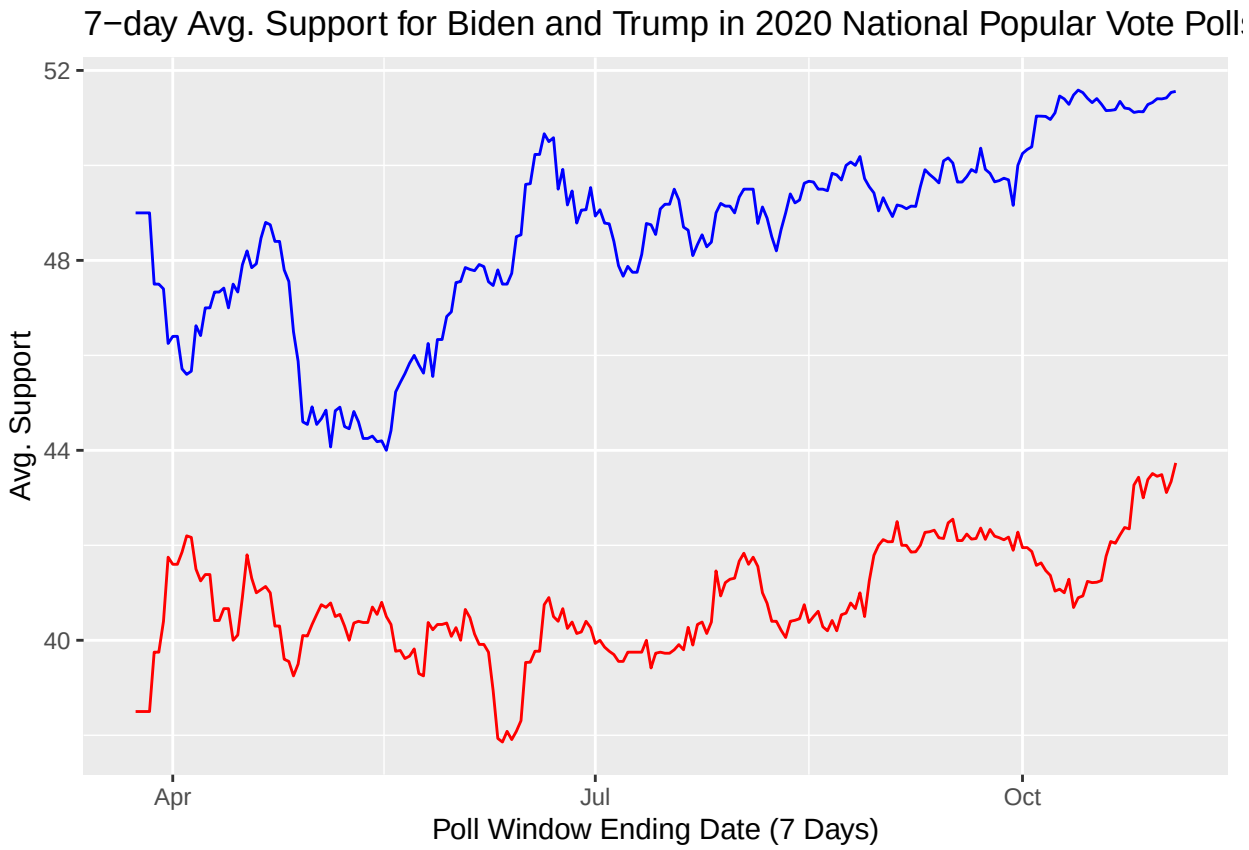


The plot reveals several interesting patterns. First, there are obvious gaps in the polling averages? Why? As previously mentioned, there are some dates for which there no polls conducted in the range defined by the bandwidth we chose. As a result, we get missing data. Second, although the line is smoother than the line that was plotted through every data point because we are taking an average over both polls and days, the trend is not very smooth. There are still jumps and the trend is not particularly smooth - e.g., the spike in June. Third, the support for both Trump and Biden is increasing over time as Election Day approaches. This seems weird as there are only two candidates being asked about, but it makes more sense when we recall that some respondents choose answers other than Biden or Trump. In particular, the percentage of respondents who indicate that they "don't know" decreases as Election Day approaches and as those people decide it can increase the percentage who support each candidate.

To try to give a better characterization, let's choose a wider bandwidth. This will make the line smoother becasue we will be including more polls and more days in each daily average. The decision of the proper bandwidth can be somewhat arbitrary, but choosing a seven-day timeframe makes some sense given that it is the length of a week and campaign strategies likely vary depending on the week. To do so all we need to do is to adjust the value for Bandwidth, rerun the code, and rebind the rows (to the object `pop_vote_avg7`). Thus:

```
Bandwidth <- 7
PV_avg <- vector(length(all_dates), mode = "list") # holding variable

for (i in seq_along(all_dates)) {
  date <- all_dates[i]

  PV_avg[[i]] <- Pres2020.PV %>%
    filter(as.integer(EndDate - date) <= 0,
           as.integer(EndDate - date) > - Bandwidth) %>%
    summarize(Biden = mean(Biden),
```

```
              Trump = mean(Trump)) %>%
    mutate(date = date)

}

pop_vote_avg7 <- bind_rows(PV_avg)

pop_vote_avg7 %>% ggplot() +
  geom_line(aes(x = date, y = Trump), color = "red") +
  geom_line(aes(x = date, y = Biden), color = "blue")  +
    labs(title="7-day Avg. Support for Biden and Trump in 2020 National Popular Vote Polls Over Time") +
  labs(y = "Avg. Support") +
  labs(x = "Poll Window Ending Date (7 Days)")
```



Much better! Again, the "better" is an aethetic judgment. The plot with a 7 day bandwidth was just as accurate and correct as the earlier plot with a 2 day bandwidth. However, the trend is easier to see in the 7-day average and the 7-day average seems large enough to prevent the jumpiness of the 2-day bandwidth but short enough to be able to capture movement over time.

As the badnwidth increases, the trend will get smoother and smoother, but at a cost of preventing over time change. As a data scientist, you need to balance the need for smoothness (to help your reader summarize the pattern) with the need to ensure that the trend still captures important shifts in the data being plotted. This is not necessarily a statistical determination! Just as important is your evaluation of whether the trend "makes sense" given what you know about the data being plotted. Context matters!

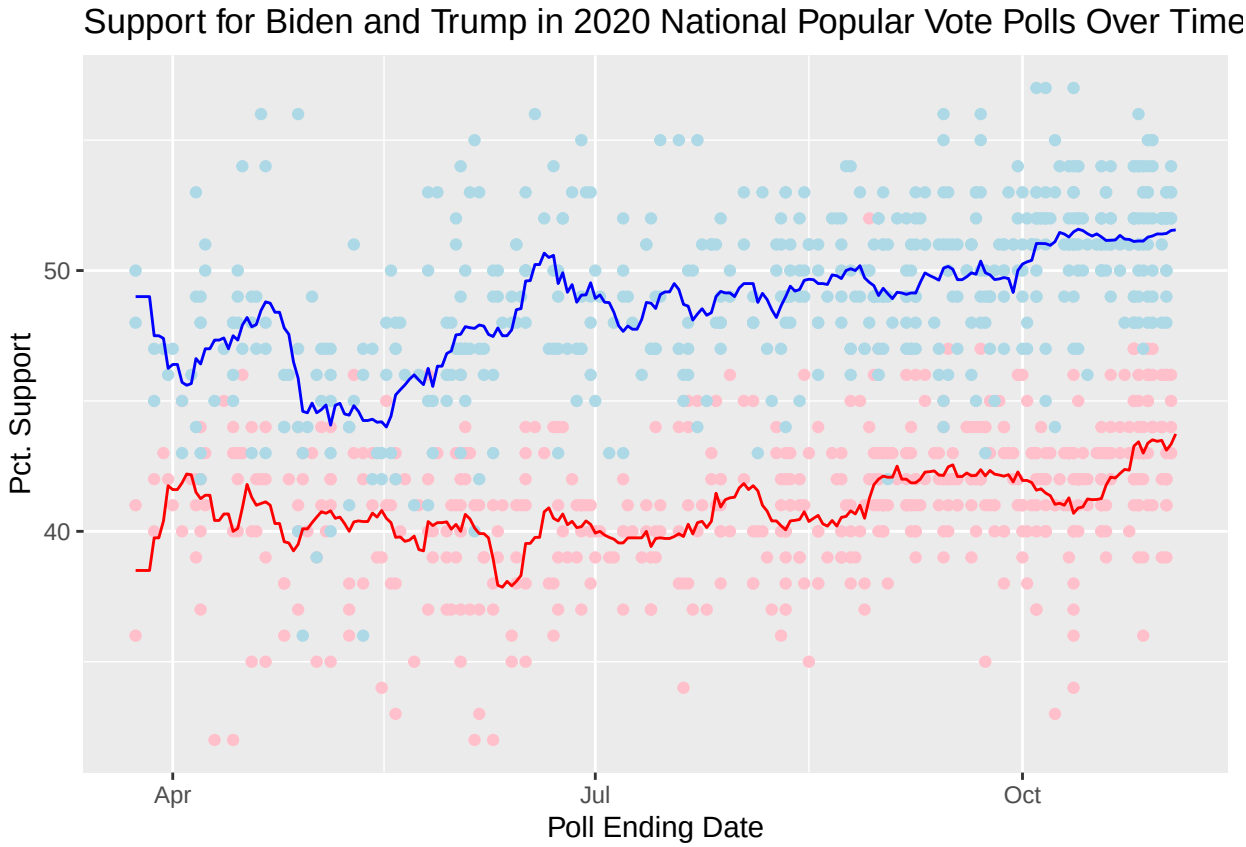
While the trend appears sensible, note what is missing. There is no sense of how many polls are being used to create the time trend, nor is there any indication of how certain we should be about the averages being plotting. These are clearly related concerns as more polls should mean more certainty if the polls

are independnet random samples of the electorate. So let's work to display this critical information. Recall that a quantity is only scientific if it includes some assessment of how certain you ar about the results being reported.

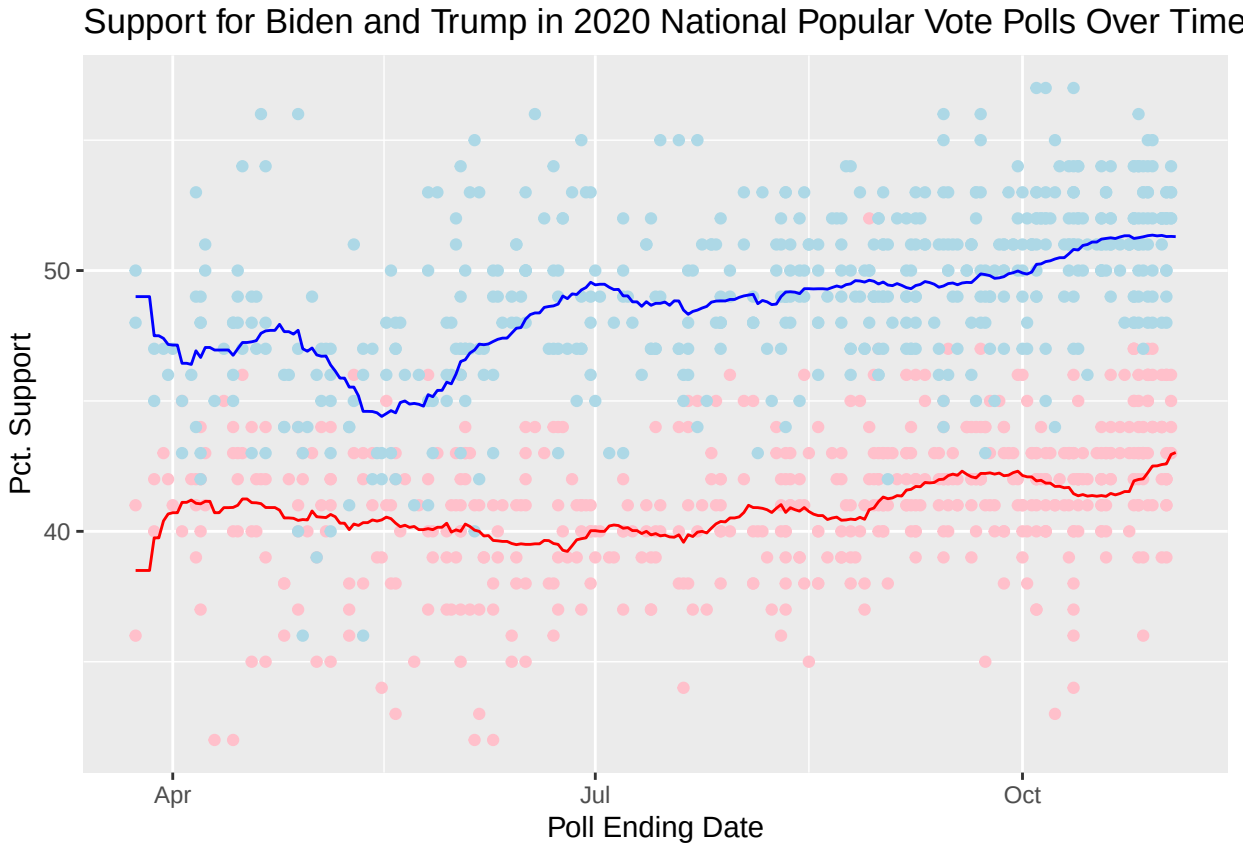
So now let's put it all together. We want to create a graph that not only plots every single public poll, but then presents the smoothed trend line. To this line we will then add a meausure of uncertainty in a bit. To start, plotting the number of data points is important for showing the underlying data being analyzed and proving a sense of how much data is being summarized. Smoothing 10 polls is very different than smoothing 1000 polls in terms of how much confidence we should have about the precision of our estimates!

To create the ggplot object we need to start by calling `ggplot`. Note that we do not start with a tibble because we are going to use different tibbles when creating the plot and we will identify the data being used in each function call for clarity. Then we use `geom_points()` to plot the polling data from the `Pres2020.PV` tibble. The code is identical to what we used before – define the variable being plotted along the x-axis and y-axis, and then any graphical elements (here `color`). We also want to plot the smoothed averages and to do so we use `geom_line()`. We tell R which tibble to use (`pop_vote_avg7`), the the x and y variable that are being plotted, and any graphical parameters. Note that it is up to you to ensure that the x and y variables are on comparable scales when plotting using different tibbles. R will attempt to plot whatever you tell it, even if it makes no sense and even if the scales are vastly different. As a data scientist you must ensure that what you are doing makes sense. Here we are OK because we are plotting dates that span the same timeframe and polling results that span the same y-axis. (If however, one was measured in percentages (0-100 scale) and the other was measured in proportions (0-1 scale) that would clearly be problematic – always check that the variables being plotted together make sense!)

```
# Now overlay on points!
PopVotePlot <- ggplot() +
  geom_point(data=Pres2020.PV,aes(x = EndDate, y = Trump), color = "pink") +
  geom_point(data=Pres2020.PV,aes(x = EndDate, y = Biden), color = "light blue") +
  geom_line(data=pop_vote_avg7, aes(x = date, y = Trump), color = "red") +
  geom_line(data=pop_vote_avg7, aes(x = date, y = Biden), color = "blue") +
  labs(title="Support for Biden and Trump in 2020 National Popular Vote Polls Over Time") +
  labs(y = "Pct. Support") +
  labs(x = "Poll Ending Date")
PopVotePlot
```



While the 7-day bandwidth makes sense, we can also do a 21-days bandwidth to illustrate the consequences of choosing a longer bandwidth. To replicate this graph on your own, all you need to do is to change the value of `Bandwidth` in the code we just ran to be 21. By pooling over a longer period of time the trend becomes even smoother. While the larger bandwidth provides a smoother characterization of the long term trends in the data, it does so at the expense of short term fluctuations – it becomes much harder to detect shifts that are shorter than the chosen bandwidth given the amount of averaging being done. So while the larger bandwidth prevents over-interpreting random changes (due to random sampling) or the effect of short term events as indicating a real change, it does so at the cost of being able to identify short-term fluctuations that are important. Hence the importance of data scientists using their knowledge of the context to help determine the best way to represent the relationships present in the analyzed data.



The analysis above works by taking the tibble `PV_avg`, converting it into a tibble using `row_bind()` and then plotting the trends for Biden and Trump separately. It does so because Biden and Trump exist as separate variables. We can use the power of tidyverse to create a plot using a slightly different approach.

We are going to create a new tibble called `pop_vote_avg_tidy` by using the `gather()` command piped thru the pop_vote_avg tibble. What the gather command does is to rearrange the tibble (note that we do not need to feed the `gather()` command a data argument because we are piping it thru – so that the rows are arranged by the key and the values being gathered are given by the value. What we are doing is rearranging the data by candidate average-date rather than just date.

```
pop_vote_avg_tidy <-
  pop_vote_avg7 %>%
  gather(key = candidate, value = share, -date, na.rm = TRUE)

dim(pop_vote_avg_tidy)
```

```
## [1] 450   3
```

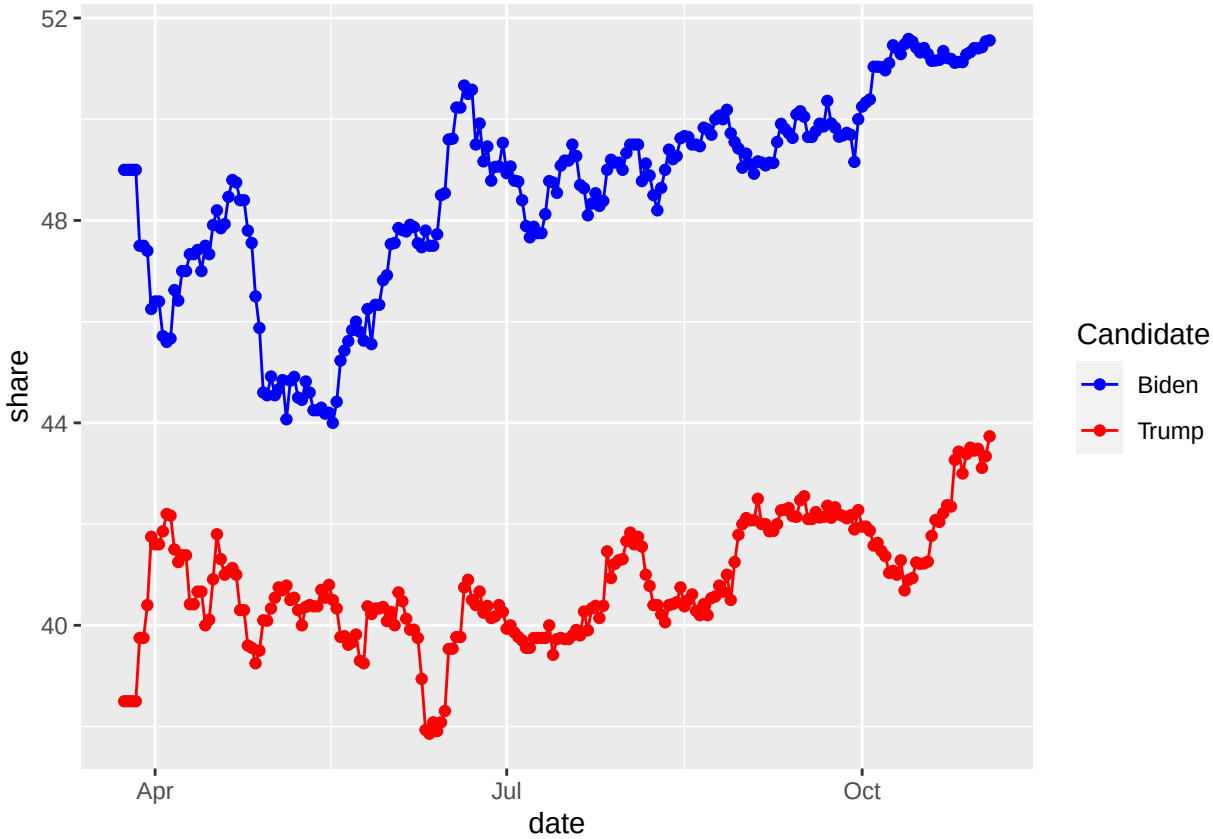
```
head(pop_vote_avg_tidy)
```

```
## # A tibble: 6 x 3
##   date       candidate share
##   <date>     <chr>     <dbl>
## 1 2020-03-24 Biden      49
## 2 2020-03-25 Biden      49
## 3 2020-03-26 Biden      49
## 4 2020-03-27 Biden      49
## 5 2020-03-28 Biden      47.5
## 6 2020-03-29 Biden      47.5
```

In other words, each row of `pop_vote_avg` included a variable for a `date`, and then the average poll results for Biden and also Trump. As a result, there are 225 rows and 3 columns. In contrast, pop_vote_avg7 collapsed the data so that each row is a date and either the average results for Biden or for Trump – resulting in double the number of rows – now 450 rows – but still 3 columns because of the addition of the `key` variable we called `candidate` that identifies whether the value of the `share` variable corresponds to Bush or Biden. In other words, what we have done in gathering the data is to essentially stack the Biden and Trump variables together into the variable called share and then create a new variable indicating whether the values of share are from the `Biden` or `Trump` variable (this has become the value of the key variable we defined to be `candidate`. Because of how the data is reshaped, the first half of the rows correspond with Biden and the second half correspond to Trump (because it is sorted alphabetically).

With this object, we can now plot the result using ggplot applid to the pop_vote_avg_tidy tibble. To plot both the points and the line associated with the smoothed candidate means we can define the variables using `aes()` – also indicating the x and y variables being plotted. We are also adding the color subcommand to be defined by the 2 dimensional factor that we are now plotting. The `fct_reorder2()` command is telling ggplot that the x and y variables being used are arranged according to the factor variable `candidate`.

```
pop_vote_avg_tidy %>%
  ggplot(aes(x = date, y = share,
        color = fct_reorder2(candidate, .x = date, .y = share))) +
  geom_point() +
  geom_line() +
  scale_color_manual(name = "Candidate",
                     values = c(Biden = "blue", Trump = "red"))
```



Once this ggplot object is defined, because the x and y variables are defined by the ggplot object we can then call the geom_point() and geom_line() to print the data points and the lined connecting the data points to the ggplot canvass. To bring color to the graph we use the `scale_color_manual` command to define the

color that will be assocated with observations associated with each of the values in the candidate factor. Note that we have also redefined the name of the factor that will be displayed on the screen in the legend from the factor name (i.e., `candidate`) to a nicer looking text (i.e., "Candidate").

Using functions: Defining a custom smoothing function

Another thing we could do is to write a custom function that will calculate the polling average for a specified bandwidth and then use that function to perform the required calculations. While we can do what we want to do using the tools we have already covered, it is useful to consider how we could have done things differently to help reveal the flexibility of R.

When we define a function we need to tell R that it is a function using the function command and then identify the arguments that will be used by the function and also the manipulations and calculations that we want the function to do given the arguments that are being passed to it. The syntax of a function is as follows:

```
function.name <- function(arguments){
  Calculation 1
  Calculation 2
  ....
  Calculation N
}
```

So defined, we can then use the function by calling:

```
function.name(arguments)
```

So let's make this concrete with a simple example. Suppose that we wanted to write a function that would convert any numeric variable that we give it to a numeric variable that has a mean 0 and a variance of 1. This is what we call a `standardized` variable and we sometimes see researchers do this when trying to compare different variables. Note that when we standardize we also remove the meaning of the variable because we are normalizing the variable to have a mean 0 and variance of 1 and this may be harder to interpret than the original variable. For example, it is easy to interpret what the average of `Biden` means – this is the average support that Biden received in the polls. But if we were to standardize `Biden` to have a mean 0 and variance 1 then the interpretation of the standardized variable is arguably more difficult. Now the variable is measuring variation relative to the average poll support and in terms of a standard deviation change. But putting that aside, how could we program a function to do this if we wanted to do so ourselves (obviously this function already exists, so this is for illustration only).

So let us create a new function called `norm.variable` that takes a variable (with the placeholder name of `VAR`) and then returns the normalized transformation of `VAR` (by subtracting off the mean of `VAR` (excluding missing values) to center the mean of `VAR` at 0 and dividing the re-centered variable by the standard deviation of `VAR` (also excluding missing values) to constrain the standard deviation to 1.

So the following code will create that function. Note that `VAR` is a placeholder for the variable that the user will supply and the expressions in the brackets is what R is going to do using the supplied variable `VAR`

```
norm.variable <- function(VAR){
  (VAR-mean(VAR, na.rm=TRUE))/sd(VAR, na.rm=TRUE)
}
```

So when we evaluate this code, we create a new function for R – a function that you should see in the "Functions" part of the R Studio Environment. If we want to use the function, however, we now need to call it. To do so we need to give it a variable to normalize. If we want to normalize the `margin` variable, we now simply call the function using that variable – recalling that we need to reference it thru the `Pres2020.PV` tibble using `$`. Thus:

```
norm.variable(Pres2020.PV$margin)
```

```
##   [1]  0.59747493  0.29559286 -0.30817128 -0.61005335 -0.61005335  0.59747493
##   [7] -0.61005335  1.20123906  0.89935699  0.59747493 -2.11946369  0.89935699
##  [13] -1.51569955  0.89935699  0.29559286 -0.61005335 -0.00628921 -0.30817128
##  [19] -0.00628921 -0.30817128  0.89935699 -0.00628921  0.59747493  0.59747493
##  [25] -0.61005335 -0.00628921 -0.00628921 -1.51569955  1.80500320  0.89935699
##  [31] -0.91193541  0.89935699 -0.00628921 -0.00628921 -1.21381748  0.29559286
##  [37] -0.91193541  0.29559286 -0.91193541 -0.30817128 -2.11946369 -0.30817128
##  [43]  1.20123906  1.80500320  0.59747493  1.80500320  1.80500320  1.20123906
##  [49]  0.59747493  0.59747493  1.50312113  0.29559286  0.89935699 -2.11946369
##  [55] -0.00628921 -1.81758162 -0.91193541  0.89935699 -1.81758162 -0.00628921
##  [61]  1.20123906 -0.30817128  0.29559286  1.80500320 -2.11946369 -0.30817128
##  [67]  1.20123906  1.50312113 -0.00628921 -0.61005335  0.59747493  0.59747493
##  [73] -0.30817128  0.59747493  0.29559286  0.29559286 -1.51569955  0.89935699
##  [79] -1.21381748  0.29559286  0.59747493 -0.00628921  0.89935699  0.59747493
##  [85]  0.59747493  0.29559286  0.29559286  0.29559286  0.89935699 -0.00628921
##  [91] -0.61005335  1.20123906  0.89935699  0.59747493  0.59747493  2.10688527
##  [97]  0.89935699  1.20123906  0.89935699  0.29559286  1.20123906  0.59747493
## [103] -0.00628921  0.89935699 -0.30817128  0.59747493 -0.00628921 -0.91193541
## [109]  1.80500320  0.59747493 -0.61005335  0.89935699  2.71064940 -0.00628921
## [115] -0.00628921 -0.00628921  0.29559286  1.50312113  1.20123906 -0.91193541
## [121]  1.20123906 -0.91193541 -0.00628921  0.59747493  0.29559286  2.40876734
## [127]  1.80500320  0.59747493  1.80500320  1.20123906  1.20123906  0.29559286
## [133]  1.20123906  1.80500320 -0.00628921  0.59747493  0.29559286  2.40876734
## [139]  0.59747493 -0.00628921 -0.00628921  0.29559286  0.89935699 -0.00628921
## [145]  0.59747493 -0.00628921  1.80500320 -1.81758162 -0.30817128  0.59747493
## [151] -0.61005335 -1.51569955  0.59747493  0.29559286  0.89935699  1.50312113
## [157]  0.29559286 -0.00628921 -0.00628921  0.89935699  0.89935699  0.59747493
## [163] -0.30817128 -0.61005335 -0.30817128 -0.00628921 -1.81758162 -0.91193541
## [169] -0.61005335  0.29559286 -0.00628921 -0.00628921 -0.61005335  0.29559286
## [175] -0.00628921 -0.91193541 -1.21381748 -0.91193541  0.29559286  0.89935699
## [181]  0.59747493 -0.30817128 -2.11946369  0.59747493 -0.30817128  0.59747493
## [187]  0.29559286  0.89935699 -0.30817128 -0.30817128 -0.00628921 -0.61005335
## [193] -0.61005335  0.29559286  0.59747493  1.20123906 -0.00628921 -0.00628921
## [199] -0.61005335  0.29559286 -0.30817128 -1.21381748  0.89935699  0.89935699
## [205] -1.81758162 -1.21381748 -2.72322782  0.29559286  0.29559286 -0.61005335
## [211]  0.29559286 -0.00628921 -0.61005335 -0.91193541 -1.21381748 -0.91193541
## [217]  0.29559286 -0.30817128  0.29559286 -0.91193541  0.29559286 -0.00628921
## [223] -0.61005335 -0.61005335 -1.81758162 -0.00628921 -0.30817128  1.20123906
## [229]  1.20123906 -0.61005335 -0.61005335  0.59747493 -0.00628921  1.50312113
## [235]  0.29559286  0.59747493  0.29559286 -0.00628921 -0.61005335 -0.00628921
## [241] -0.91193541  0.59747493  0.89935699 -0.00628921 -1.21381748 -0.30817128
## [247] -0.91193541 -0.00628921  2.10688527 -0.00628921 -0.00628921 -0.30817128
## [253] -1.51569955 -1.81758162 -3.02510989  0.29559286 -0.61005335 -0.61005335
## [259] -0.61005335  1.50312113  0.59747493  1.50312113  0.29559286 -2.11946369
## [265]  0.29559286 -0.30817128 -0.00628921  0.59747493  0.29559286  0.89935699
## [271]  0.59747493  0.59747493  0.29559286 -0.00628921  1.50312113  0.29559286
## [277]  0.89935699  0.59747493 -0.00628921  0.59747493 -1.21381748 -0.61005335
## [283] -0.00628921  2.40876734 -0.30817128  0.89935699  1.20123906 -1.21381748
## [289]  0.59747493  0.29559286  1.50312113  0.89935699  0.59747493 -0.30817128
## [295] -0.00628921 -0.00628921 -0.00628921  0.89935699  0.59747493 -1.21381748
## [301] -0.61005335  0.29559286 -0.61005335 -0.61005335  0.59747493 -0.00628921
## [307] -0.00628921  1.50312113  0.59747493 -1.51569955  0.59747493  0.29559286
```

```
## [313] -1.51569955  0.29559286 -0.30817128 -1.21381748  0.29559286  0.59747493
## [319] -0.30817128 -0.00628921 -0.61005335  0.29559286  0.29559286  0.29559286
## [325] -1.21381748 -0.00628921 -0.00628921  0.59747493  0.29559286  1.20123906
## [331]  0.59747493  1.50312113 -0.61005335  0.89935699  0.89935699 -0.30817128
## [337] -1.81758162 -0.30817128 -0.00628921 -0.30817128 -0.00628921  2.10688527
## [343] -0.00628921 -0.91193541  0.89935699  0.59747493  2.10688527 -1.51569955
## [349] -0.00628921  0.29559286 -0.00628921  0.59747493 -0.30817128 -0.00628921
## [355]  0.29559286  0.59747493 -1.21381748 -1.21381748 -1.21381748  0.29559286
## [361]  0.59747493 -0.61005335  0.29559286  0.29559286  0.59747493  1.20123906
## [367]  0.89935699 -0.91193541  1.20123906 -0.00628921  1.80500320 -0.00628921
## [373]  0.29559286 -0.00628921 -0.30817128  1.20123906  0.59747493 -1.21381748
## [379] -0.00628921 -0.00628921 -0.00628921 -1.21381748 -0.30817128  1.50312113
## [385]  0.29559286  0.59747493 -0.00628921 -0.61005335  1.80500320  0.29559286
## [391]  0.29559286  1.20123906 -0.00628921  0.59747493  1.20123906 -0.61005335
## [397] -0.00628921  0.89935699  0.29559286  1.50312113  0.59747493 -0.00628921
## [403]  0.29559286  1.20123906  1.80500320  1.50312113  0.89935699  0.29559286
## [409] -0.00628921 -0.00628921 -0.30817128  1.80500320 -1.21381748 -0.00628921
## [415]  0.29559286  0.59747493  0.59747493  1.20123906  1.80500320 -2.42134576
## [421] -0.30817128 -0.30817128  0.59747493 -0.91193541 -1.51569955 -0.00628921
## [427] -1.51569955 -1.21381748  0.89935699 -0.30817128 -0.30817128  0.59747493
## [433] -0.30817128 -1.21381748  1.20123906 -0.00628921 -1.21381748 -0.00628921
## [439]  0.59747493 -1.51569955  1.50312113 -1.21381748 -0.61005335 -1.51569955
## [445] -0.91193541 -0.61005335 -1.21381748  0.29559286 -1.81758162 -0.00628921
## [451] -2.11946369  0.29559286  1.20123906 -0.91193541 -0.91193541 -1.51569955
## [457] -0.91193541 -1.21381748 -0.00628921 -1.21381748 -4.23263816 -0.91193541
## [463] -1.21381748 -0.00628921 -1.81758162 -0.91193541 -0.61005335 -0.30817128
## [469] -1.51569955 -0.91193541 -0.30817128 -1.21381748 -1.51569955 -1.51569955
## [475] -1.81758162 -0.30817128 -1.21381748 -0.61005335 -2.42134576 -1.51569955
## [481] -4.23263816 -0.61005335 -1.21381748  1.20123906 -0.00628921  1.80500320
## [487] -1.21381748 -0.61005335 -0.00628921  0.29559286  0.59747493 -0.00628921
## [493] -0.61005335 -1.21381748 -0.00628921 -0.91193541 -0.00628921  0.29559286
## [499] -0.30817128 -0.00628921 -0.91193541 -0.91193541 -1.21381748 -1.21381748
## [505] -1.21381748  0.59747493 -1.81758162  1.80500320  0.29559286 -0.61005335
## [511] -1.21381748 -0.91193541 -0.91193541 -1.21381748  0.59747493 -0.91193541
## [517] -0.00628921 -0.91193541 -0.61005335 -0.30817128  1.50312113 -0.61005335
## [523] -3.02510989  0.89935699 -1.21381748 -2.42134576  0.29559286  1.20123906
```

If we run this we will get output that is dumped directly to the screen. It is likely more useful to save that output as a new object:

```
norm.margin <- norm.variable(Pres2020.PV$margin)

mean(norm.margin)
```

```
## [1] -1.881444e-16
```

```
sd(norm.margin)
```

```
## [1] 1
```

Perfect.

Of course we can far more complicated in the expressions that we do and the input we use for our functions. To push our understanding a bit more, let's now define a custom function called `poll_ma` that will be a function of adate that the user supplies (called `date`), a tibble that the user supplied that contains variables named `EndDate`, `Biden` and `Trump` (which will be called generically `.data` by the function), and a bandwidth for smoothing called `days`. Note that we have set `days = 7` in the arguments we are sending to the function

we are defining. What this means is that if the user does not provide a value for days that the function will choose a default value of 7. Note that we are not providing a default value for `date` and `.data` – meaning that if the user does not provide a value the function will not run.

```
poll_ma <- function(date, .data, days = 7) {
  .data %>%
    filter(as.integer(EndDate - date) <= 0 &
         as.integer(EndDate - date) > - days) %>%
    summarize(Biden = mean(Biden , na.rm = TRUE),
              Trump = mean(Trump, na.rm = TRUE)) %>%
    mutate(date = date)
}
```

So what does the function do? It begins by filtering the supplied data frame (which the function is referring to as .data – a name that will be replaced with the actual name that is supplied by the user) based on whether the difference between the `EndDate` variable in the supplied tibble and the `date` value given to the function by the user is less than or equal to 0 (after converting the difference in dates to an integer) and also that the difference is within the number of specified `days` (note that the value is negative because the Election date of interest occurs later in the year than the last polling day).

Working with the filtered data, the function then takes the average support for Biden and Trump (after removing any missing data) and it creates a variable called date that takes on the value of date supplied to the function.

To see this in action, let's call the function as follows. So what we are doing is taking the average poll values for all polls in the `Pres2020.PV` tibble that are within 7 days (the default value – notice that there is no 3rd argument in the function call) of the date that we are providing – here `election.day` (so Election Day 2020).

```
poll_ma(date = election.day, .data = Pres2020.PV)
```

```
## # A tibble: 1 x 3
##   Biden Trump date
##   <dbl> <dbl> <date>
## 1  51.6  43.7 2020-11-03
```

From the output being printed to the screen you can see that we have the 7-day polling average for Trump and Biden ending on 11-3-2020 – Election Day 2020. But with this function we can now see the effects of different bandwidths by changing one value when calling the function we have just defined.

So if we want to see what the average support is for all polls done in the 100 days leading up to Election Day (so `days=100`), or the last two days (so `days=2`), or even the last day (so `days=1`) we can run the following:

```
poll_ma(election.day, Pres2020.PV, days=100)
```

```
## # A tibble: 1 x 3
##   Biden Trump date
##   <dbl> <dbl> <date>
## 1  50.4  41.9 2020-11-03
```

```
poll_ma(election.day, Pres2020.PV, days=2)
```

```
## # A tibble: 1 x 3
##   Biden Trump date
##   <dbl> <dbl> <date>
## 1  52.8  44.2 2020-11-03
```

```
poll_ma(election.day, Pres2020.PV, days=1)
```

```
## # A tibble: 1 x 3
##   Biden Trump date
```

```
##   <dbl> <dbl> <date>
## 1   NaN   NaN 2020-11-03
```

So that is useful because instead of having to rerun an entire snippet of code as we were doing before when changing the bandwidth, we can now change one argument of the function. This is important because if we have to change our code to do a different calculation, we now only have to make 1 change rather than changing every snippet. Having multiple snippets doing the same manipulations is always a source of possible error because of the inevitability of having to change the code. The ability to use a function helps protect against this by limiting the need to make multiple alterations.

Another benefit is that if we have a function we can then use the `map_df` command to apply a function to every element of a tibble to return a tibble of outputs. So if we want to apply the `poll_ma` function that we just wrote to the `Pres2020.PV` data frame.

```
map_df(all_dates, poll_ma, Pres2020.PV)
```

Where this is going to send the values associated with `all_dates` to the `poll_ma` function applied to the `Pres2020.PV` data frame. This is going to result in the function being applied 225 times – once for each date in `all_dates` (and each value of is going to be evaluated as a date in `poll_ma`).

To run this we can use this to define a new tibble object – called `PollAvg7Day` – and then determine the dimensions of the tibble as well as the first few observations.

```
PollAvg7Day <- map_df(all_dates, poll_ma, Pres2020.PV)
dim(PollAvg7Day)
```

```
## [1] 225   3
```

```
head(PollAvg7Day)
```

```
## # A tibble: 6 x 3
##   Biden Trump date
##   <dbl> <dbl> <date>
## 1  49    38.5 2020-03-24
## 2  49    38.5 2020-03-25
## 3  49    38.5 2020-03-26
## 4  49    38.5 2020-03-27
## 5  47.5  39.8 2020-03-28
## 6  47.5  39.8 2020-03-29
```

With this basic structure we can also explore other ways of smoothing the data. The code we have just worked through treats every poll equally, regardless of sample size. But we know from the Law of Large Numbers that larger random samples are more likely to be closer to the true mean so perhaps it makes sense to compute a weighted average of the polls that weights each poll by the proportion of respondents that are in the poll. In other words, if there are 3 polls of 500, 500, and 1000 respondents respectively, we can compute the polling average after accounting for the relative size of each. If our confidence is a linear function of sample, size, we we compute the weighted mean by weighting each poll mean according to the proportion of respondents that belong to the poll.

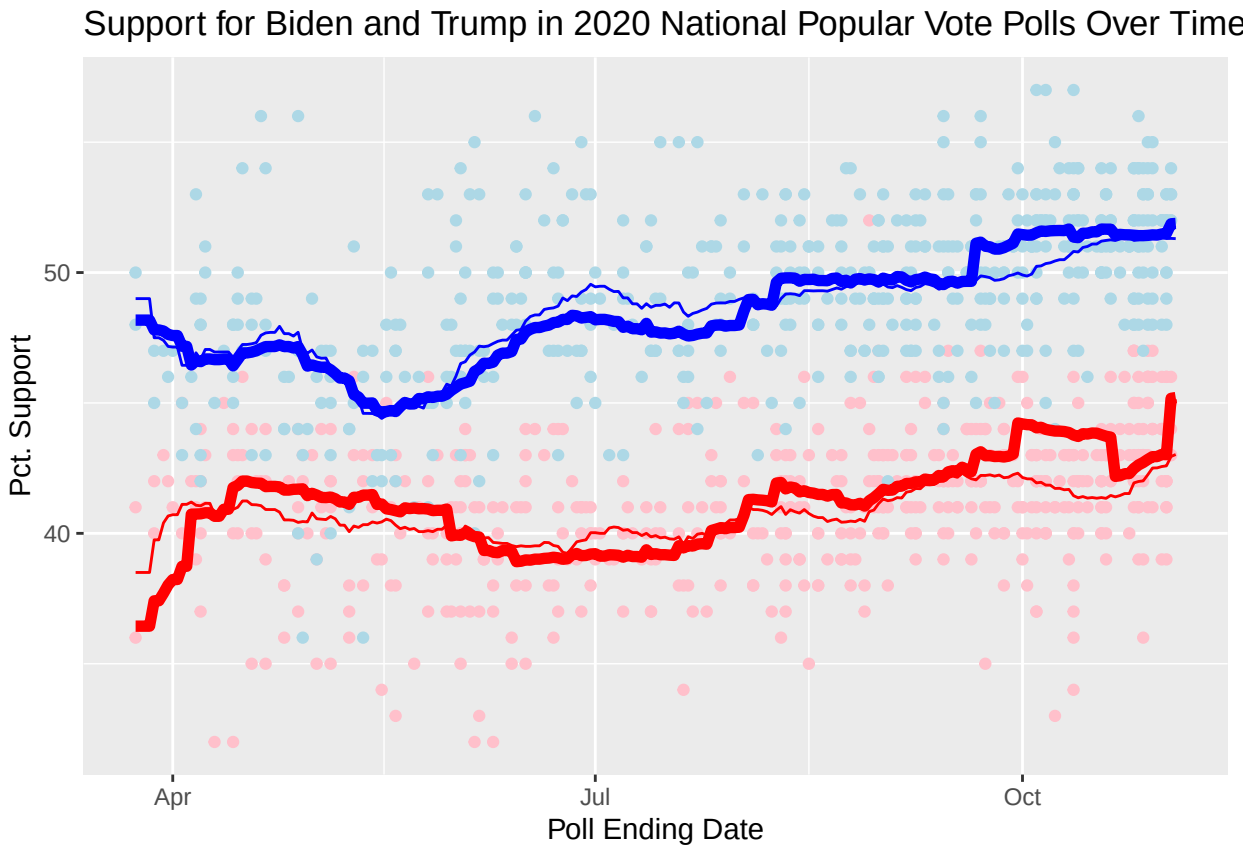
To implement in R we can return to our original code and add two changes. First, we are going to use `mutate()` to create a new variable that is the percentage of total respondents that were interviewed in the polls conducted during the specified bandwidth belonging to each poll. This variable – `pct.size` – is simply the sample size for each poll over the sum of the sample sizes for all polls that were done during the period.

The second chage to the code uses this proportion to compute a `weighted.mean()` instead of a mean. Now we define `Biden` and `Trump` using `weighted.mean()` – with the weights given by the relative sample size `pct.size` just described.

```
Bandwidth <- 21
PV_avg <- vector(length(all_dates), mode = "list") # holding variable

for (i in seq_along(all_dates)) {
  date <- all_dates[i]

  PV_avg[[i]] <- Pres2020.PV %>%
    filter(as.integer(EndDate - date) <= 0 &
           as.integer(EndDate - date) > - Bandwidth) %>%
    mutate(pct.size = SampleSize/sum(SampleSize)) %>%
    summarize(Biden = weighted.mean(Biden, w = pct.size, na.rm = TRUE),
              Trump = weighted.mean(Trump, w = pct.size, na.rm = TRUE))  %>%
    mutate(date = date)
}

pop_vote_avg_wgt <- bind_rows(PV_avg)
```

Plotting this uses the same commands as before. To see the effect of weighting by sample size rather than weighting every poll equally we can plot both sets of lines because we have saved both. We have increased the width of the population-weighted averages to make the comparison clearer.

```
ggplot() +
  geom_point(data=Pres2020.PV,aes(x = EndDate, y = Trump), color = "pink") +
  geom_point(data=Pres2020.PV,aes(x = EndDate, y = Biden), color = "light blue") +
  geom_line(data=pop_vote_avg21, aes(x = date, y = Trump), color = "red") +
  geom_line(data=pop_vote_avg21, aes(x = date, y = Biden), color = "blue") +
  geom_line(data=pop_vote_avg_wgt, aes(x = date, y = Trump), color = "red", lwd=2) +
  geom_line(data=pop_vote_avg_wgt, aes(x = date, y = Biden), color = "blue", lwd=2) +
  labs(title="Support for Biden and Trump in 2020 National Popular Vote Polls Over Time") +
  labs(y = "Pct. Support") +
  labs(x = "Poll Ending Date")
```



Somewhat interestingly, you can see that the weighted poll averages is actually slightly jumpier than the average that treats every poll equally. This makes sense if you think about it, however, because the sample-size weighted poll averages is going to be more sensitive to the inclusion and exclusion of polls with large sample sizes. So when the set of public pols consists of telephone based polls of around 1000 respondents and online polls that can interview 10,000 individuals in a day, the sample size weighted results will be more impacted by the results of larger polls.

In a perfect world in which the polls are all random samples from the electorate then this may make sense for the reasons we examined earlier when talking about the Law of Large Numbers. But our data is rarely so pure and we *always* have to think about where our data is coming from. Do we really think that the polls are all random samples of the electorate and are we so confident in that so as to more heavily weight the result of a poll simply because it has more people who answer the poll? Maybe not.

But the general point is that we have decisions to make as to *how* to smooth. Not only in terms of the bandwdth we use – i.e., how many days worth of polls to use – but also how we want to combine them. Do we treat every poll as equally informative? Do we weight polls with larger sample sizes more? Perhaps we should weight based on time – right now every poll in the bandwidth is equally weighted but perhaps we want to discount polls that were done futher away in time? And if we do want to use time-dicounting, how should we discount? A linear function of time? An exponential function? How differntly should we treat polls that were done 5 days ago versus 2 days ago in a world of 24x7 newscoverage (but also hardened partisan opinions and identities)? All important questions that must be decided and persuasivelly argued for.

Accounting for Uncertainty: Resampling the polling average

So far we have explored many tools for characterizing a pattern, but we still have yet to quantify our uncertainty. Let's do so now.

To do so we are going to assume that the polls we are analyzing are all random samples. The uncertainty

that we are going to quantify is the error associated with random sampling – so what would happen if we had repeated realizations of a set of public polls that we will generate by sampling with replacement from the original data frame. Note that we are assuming several non-trivial assumptions.

So what are we going to do. First we are going to create a new data frame by sampling polls (with replacement from Pres2020.PV) using sample_n. To this new tibble we are going to use the poll_ma function we defined earlier and the map_df command to compute the 7-day average for every date in the all_dates sequence. So that will give us the realization of 1 alternative time-series based on resampling from our dataset but we want to know how much the trend lines themselves may vary. To do this we need to see how those trends vary as we vary the sample of polls we analyze.

In particular, we are going to draw 100 hypothetical data sets from the Pres2020.PV tibble we collected. For each sampled tibble (sample.dat) we want to calcualte the smoothed moving average using map_df and poll_ma applied to that sampled tibble. Because we want to collect the results into a single tibble so we can see how much the average varies *for every date in all_dates* we take the output of that mapping – which consists of Biden, Trump, Date in that order – and drop Date (by dropping the third column) and then column binding (cbind) the remaining columns of poo (i.e., Biden, Trump) onto the existing object `samp`.

So in each iteration the loop rewrites `sample.dat` and `itersample` and it takes the object `itersample` and adds it to the left of the existing tibble `samp`. So every interaction increases the number of columns of `samp` by 2.

```
sample.dat <- sample_n(Pres2020.PV,nrow(Pres2020.PV),replace = TRUE)
samp <- map_df(all_dates, poll_ma, sample.dat)
dim(samp)
```

```
## [1] 225   3
```

```
for(i in 1:100){
  sample.dat <- sample_n(Pres2020.PV,nrow(Pres2020.PV),replace = TRUE)
  itersample <- map_df(all_dates, poll_ma, sample.dat)
  itersample <- itersample[,-3]
  samp <- bind_cols(samp,itersample)
}
```

This results in the `samp` tibble ending up with with a dimension of `{r} dim(samp)`.

So now `samp` is a tibble that, after the 3rd column, alternates between Biden and Trump. To collect the Biden and Trump polling averages associated with each date – recall that each row is a date – we want to extract all of the columns associated with Biden and Trump. To do this we are going to us the ability to access particular columns in a data frame by extracting all of the columns associated with Biden and Trump. For Biden, this includes columns 1 and then every other column starting with column 4 (because column 2 is the Trump average from the first resampling and column 3 is the variable `all_dates`). To select the variables related to Trump's averages requires selecting column 2 (because column 1 was Biden) and then every other column starting with 5 (because 3 was date, and 4 was the Biden average associated with the second resampling).

Now that we have extracted just the averages, lets take a peak by looking at the first five rows and first five columns.

```
TrumpSamp <- samp[,c(2,seq(5,ncol(samp),by=2))]
BidenSamp <- samp[,c(1,seq(4,ncol(samp)-1,by=2))]
```

```
BidenSamp[1:5,1:5]
```

```
## # A tibble: 5 x 5
##   Biden...1 Biden...4 Biden...6 Biden...8 Biden...10
##       <dbl>     <dbl>     <dbl>     <dbl>      <dbl>
## 1      49.6     NaN        48        50         48
```

```
## 2      49.6      NaN        48         50         48
## 3      49.6      NaN        48         50         48
## 4      49.6      NaN        48         50         48
## 5      48.1       46.3      47.5       48.3       46.5
```

As expected, each tibble is a matrix where the rows indicate dates and the columns indicate the polling average associated with each date.

To the tibble of smoothed averages we add the date associated with each average in a variable called date by selecting the 3rd column of the tibble we created.

```
TrumpSamp <- bind_cols(date=samp[,3],TrumpSamp)
BidenSamp <- bind_cols(date=samp[,3],BidenSamp)
```

But now we have a tibble where each row is a date – now included – and each column is a separate polling mean for that day based on the resampled polls that occurred for that iteration. So if we want to calculate the interval containing 95% of the polling averages given the repeating samples of public polls that we have drawn, we can simply calculate the empirical quantiles for each day. Because each row in the tibble reflects a different day, looping over the rows of the tibbles we just created and calculating the quantile for each day in a variable called `LCI` (for the 2.5% percentile) and `UCI` (for the 97.5 percentile).

To see the outcome of this loop we can look at the first 10 observations for the upper and lower quantiles for the Biden average. Note that each value is a percentile for a different day with the first observation belonging to the first date of national public presidential polls. So on the first day of polling, 95% of the polling averages fell between 48 and 50. So does this rather small interval mean that we were very certain where the race stood on the first day of polling? Not really. Recall that to account for the uncertainty we have we use the sample of data we observe - and the assumption that it is a random sample of the population – to resample from the observed data to determine how much the estimates may fluctuate in response to the random sampling. We can prove that this estimate is consistent – which means that it will provide the right answer as the number of data points (and resampling samples) increases. The issue is that for any particular dataset such as the data set we observe that the number of polls to sample from may be limited. For example, if there is only 1 poll that is done during the period of our bandwidth then there will be no variation around the estimated mean because there are no other polls to sample from! So our ability to account for uncertainty using the observed data is limited if the data points we are sampling from are limited. In such cases we can use the result of the CLT – and the normality of the sample mean – to compute the 95% confidence interval analytically, but even then we need to make some assumptions. (In particular, we need to know the standard deviation of the sample mean. But if we think that the support in the electorate is changing over time it can be difficult to know what the standard deviation of the sample mean at a particular point in time should be.)

```
BidenSamp$LCI[1:10]
```

```
##  [1] 48.00000 48.00000 48.00000 48.00000 45.33333 45.33333 45.93214 45.25000
##  [9] 45.42500 45.42500
```

```
BidenSamp$UCI[1:10]
```

```
##  [1] 50 50 50 50 50 50 50 47 47 47
```

So now we can pull it all together to create the final characterization. We already know how to plot the values associated with each poll as well as the smoothed 7-day average that summarizes the trend for each candidate. To this we want to denote how much that smoothed 7-day average fluctuates as we resample from the tibble of public polls using the quantiles we just calculates.

To produce the plot we are going to use the same code we have already used – including the labels, points, and 7 day averages. The new addition s the `geom_ribbon()`. Here we again need to specify the data we are using – `TrumpSamp` for Trump and `BidenSamples` for Biden – the variable being plotted along the x-axis (`date`) and the variable that defines the upper and lower values of the "ribbon" that is going to be plotted along the y-axis. Recall that the ribbon we are plotting is the 95% confidence interval that, for every date,

ranges from LCI to UCI. To help denote it from the other objects we are going to make the lines dashed (`linetype=2`) and we are going to make the shading of the ribbon semi-transparent (`alpha=.1`).

```
ggplot() + geom_point(data=Pres2020.PV,
                      aes(x = EndDate, y = Trump),
                      color = "pink") +
  geom_point(data=Pres2020.PV,
             aes(x = EndDate, y = Biden),
             color = "light blue") +
  geom_line(data=pop_vote_avg7,
            aes(x = date, y = Trump),
            color = "red") +
  geom_line(data=pop_vote_avg7,
            aes(x = date, y = Biden),
            color = "blue") +
  geom_ribbon(data=TrumpSamp,
              aes(x=date,ymin=LCI, ymax=UCI),
              linetype=2, alpha=0.1) +
  geom_ribbon(data=BidenSamp,
              aes(x=date,ymin=LCI, ymax=UCI),
              linetype=2, alpha=0.1) +
  labs(title="Support for Biden and Trump in 2020 National Popular Vote Polls Over Time") +
  labs(y = "Pct. Support") +
  labs(x = "Poll Ending Date")
```