Dyanamic Graphics

MA386 - Statistical Programming

Summary: Static graphics are very valuable for telling a vast number of stories, but some stories are best told with interactive or animated graphics. This is still a growing and changing area in `R`, but we will explore some of the tools for making a graphic dyanamic. We will also examine tools for making graphics with a map acting as the background, which is useful for economic and climate data.

We should preface this unit by stating that the best way to provide interactivity is through Shiny, a method for creating websites (or documents) that are reactive. That is, the user makes a selection (like whether they want a smoother displayed), and the graphic is then updated as a result. A very simple example along with tutorials can be found on their website: http://shiny.rstudio.com/. Instead of diving into Shiny, which has a bit of a learning curve, we will be examining alternative routes which provide some minimal interactivity.

1 Plotting on Maps

Summary: Several datasets are best represented on a map. Political polls may choose to show toward which party each state is leaning. Predicted hurricane paths are overlayed on a specific region of the country. And, crime rates might be reported at the city level. Within `R`, we examine a package which integrates with the foundation we formulated using the grammar of graphics.

The `ggplot2` package has provided both geoms and generic functions for adding map layers to a plot. These layers are often added first in order to provide a background for a plot. Additional layers are then plotted on top. Once a map layer has been specified, the coordinate system is then in terms of latitude (y-axis) and longitude (x-axis). So, in a sense, we are not truly working with anything new. However, while the ability to add map layers is built into the `ggplot2` package, the maps themselves must be obtained from additional sources. One very helpful package is `ggmap` (and its dependencies), which provides built-in maps as well as tools for geocoding (obtaining GPS coordinates) locations.

Example: *(Houston Murder)* A dataset (available in the `ggmap` package) provides data on crimes committed between January 2010 and August 2010 in Houston, Texas. Suppose we are interested in examining where *murders* have taken place in Houston over this time period. It would be beneficial to plot this on a map.

```
# Obtain data
#  Consider only murders
data(crime, package="ggmap")

murder.df <- crime %>%
  filter(offense=="murder")


# Obtain Base Map
#  Obtain base map from Google, which we zoom out to city.
Houston.map <- get_map("Houston Texas", zoom=11, maptype="roadmap")


# Create map
ggmap(Houston.map) + # acts like a ggplot() call
  geom_point(data=murder.df, aes(x=lon, y=lat),
             size=3, colour="darkred") + # Add murders
  labs(x="", y="") + # Remove labels on axes
```

```
  theme(axis.ticks=element_blank(),
        axis.text=element_blank()) # Remove axes
```

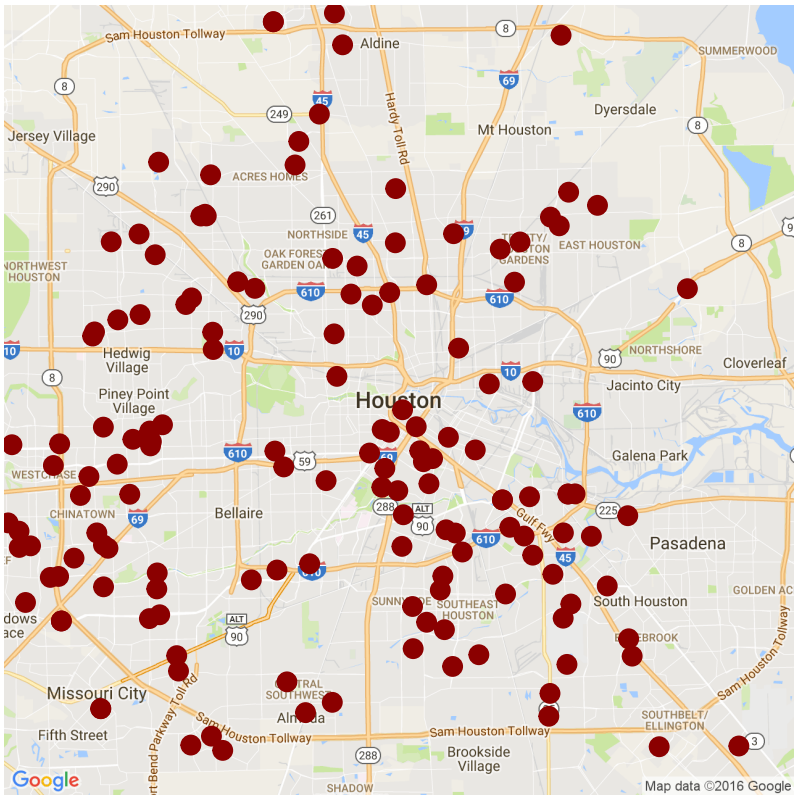


Figure 1: Map of Houston murders between January and August 2010.

The above graphic is useful as there are not many (relatively speaking) murders over this time period. However, we can imagine that overlaying several points on a map may become overwhelming. It may be beneficial to consider a summary instead of plotting the raw data. This leads to a desire to fine tune both the layer added to the map, as well as the underlying map which functions as the backdrop.

As an example, consider looking at all violent crimes that occur in Houston instead of only murders. In order to address the additional number of points, we consider a summary of the data. Specifically, consider a two-dimensional density estimate, a generaliation of a histogram to multiple dimensions. This allows for us to have a sense of the spatial distribution of the crimes across Houston.

Example: *(Houston Violent Crimes)* Suppose we are interested in examining the distribution of *violent crimes* in the city across this time period.

```
# Obtain data
#  Obtain the location of violent crimes.
violent.list <- c("murder", "robbery", "aggravated assault", "rape")
violent.df <- crime %>%
  filter(is.element(offense, violent.list))


# Obtain Base Map
```

```
#  Make the map black and white to avoid colour confusion.
Houston.map <- get_map("Houston Texas", zoom=11, maptype="roadmap", color="bw")


# Create map
ggmap(Houston.map, legend="bottomright", extent="device") +
  stat_density2d(data=violent.df,
                 mapping=aes(x=lon, y=lat, fill=..level..), alpha=0.6,
                 size=2, bins=4, geom="polygon") +
  labs(fill="Violent Crime\nDensity") +
  scale_fill_continuous(low="grey25", high="red") +
  theme(legend.direction="horizontal")
```

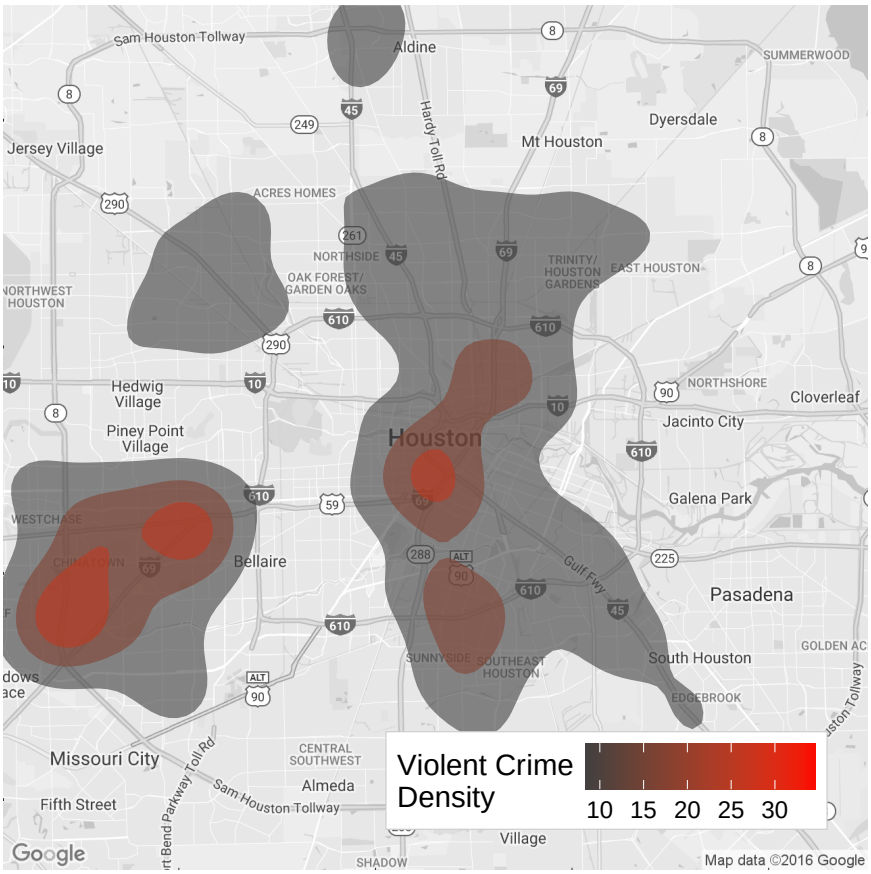
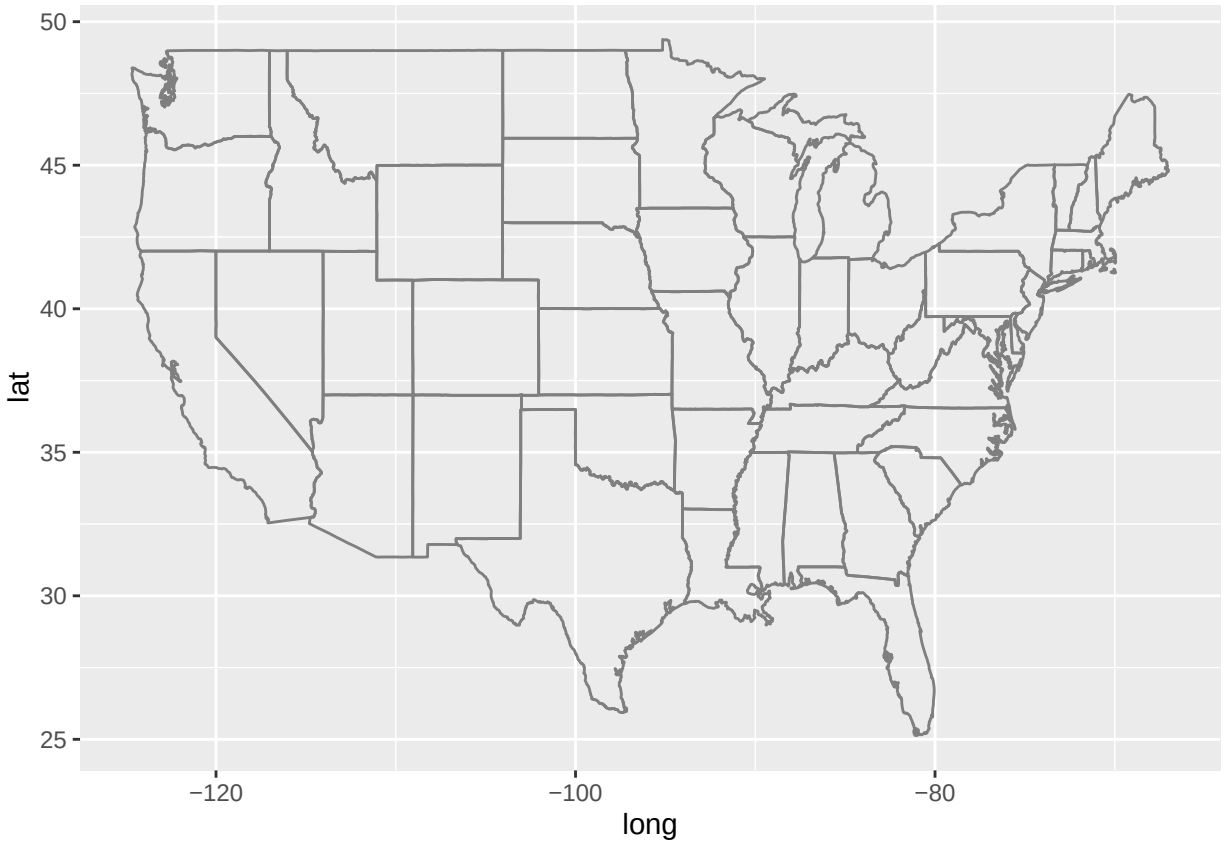


Figure 2: Density of violent crimes across Houston between January and August 2010.

The `get_map()` function is useful when looking at the detail on the plot is important. For example, in the above examples, knowing which part of the city corresponds to high crime rates is important. However, when comparing "red states" to "blue state", the roads that run through the country are not important. For these applications, an outline of the states is sufficient. The function `borders()` in the `ggplot2` package adds a layer which constructs the states. This layer can then be added to:

```
ggplot() + borders("state")
```



Again, we might want to clean this up by removing axes, changing the colour of the background panel, etcetera, but it provides a good starting point if the states provide a backdrop. If, however, we want the map of the states to convey information, such as filling the color of each state, the `borders()` function is not appropriate because it does not accept aesthetic mappings. Instead, we need to construct two datasets: one for the state borders, and the other for the aesthetics used in the mapping. The two datasets are linked through a `map_id` aesthetic.

Example: *(Murder Rate in US)* Which states have the highest (per capita) murder rates? Use the `USArrests` dataset provided with the base `R` installation to address this question. Note: this data is from 1973 and is therefore dated.

In order to attach aesthetics, we need more than a backdrop for our plot. We need a dataset which contains information on how each state (or region in general) should be drawn, and this information must be linked to the state. The `ggplot2` package contains a function with provides such a dataset for various regions. We request the one for the (continental) US.

```
# Obtain Map Data
#  Contains drawing regions in terms of lat/long and state id's.
States.df <- map_data("state")
```

We then ensure that our raw data file has a variable which can be mapped to the region in the map data. We do this by appending the state names (all lowercase) to our existing dataset.

```
# Alter Crime Data
#  Add a column of state names, matching what we get in map data for id.
USArrests <- USArrests %>%
  mutate(State = tolower(rownames(USArrests)))
```

Now, we are prepared to construct the graphic. We need to make use of the `map_id` aesthetic to ensure the link between the borders and raw data is established. We fill each state corresponding to the murder rate.

```
# Construct Graphic
ggplot(data=USArrests, aes(map_id=State, fill=Murder)) +
  geom_map(map=States.df) +
  expand_limits(x=States.df$long, y=States.df$lat) +
  labs(title="Murder Rate by State in 1973",
       fill="Murder Arrests\n(per 100,000)") +
  theme(axis.title=element_blank(),
        axis.ticks=element_blank(),
        axis.text=element_blank(),
        panel.background=element_blank(),
        panel.grid=element_blank(),
        legend.position="bottom")
```

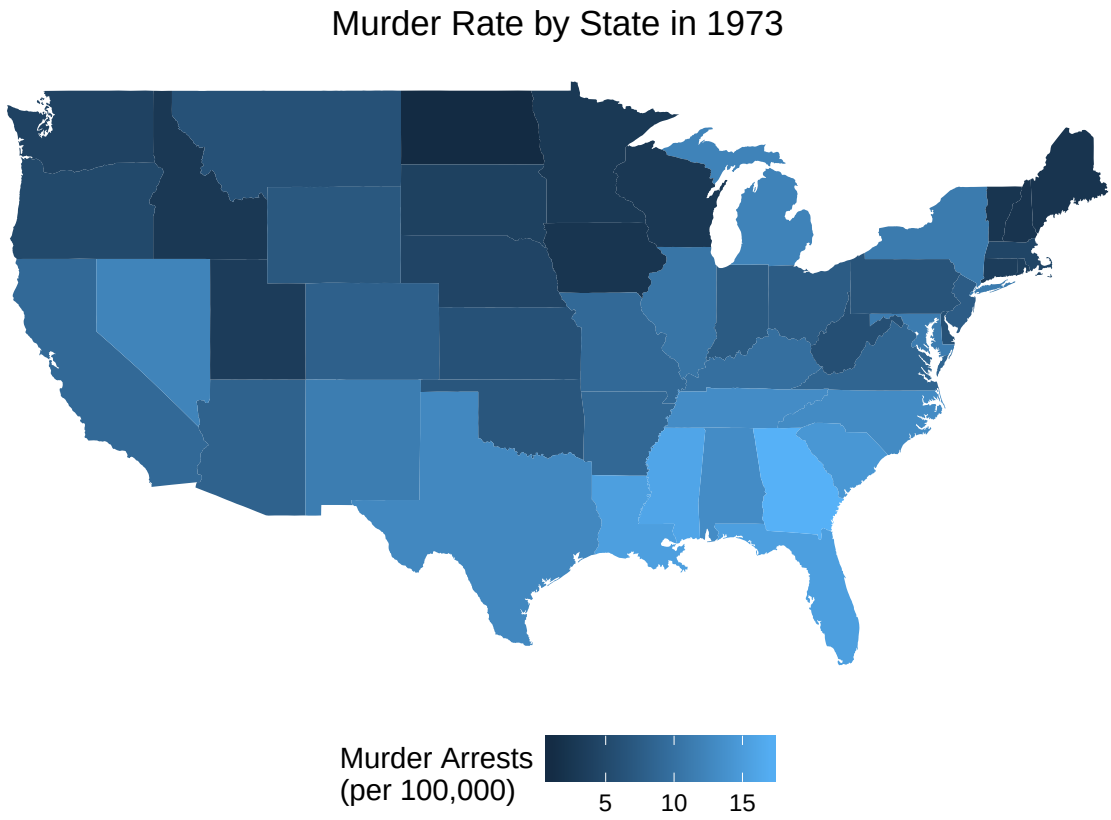


Figure 3: Murder rate per capita across the US in 1973.

One unique feature of the above graph is the `expand_limits()` layer. This is required because the latitude and longitude which form the scales for the x- and y-axis are not present in the dataset which defined the graphic in the `ggplot()` statement. Therefore, this statement ensures that the scales are properly set.

From the graphic, we are not suprised that the highest murder rates (per capita) are in states with large cities and the lower murder rates occur in less populated areas of the country.

2 Animation via Flip Charts

Summary: The simplist form of animation in `R` is to construct several graphics for which a small change has been made, similar to a flip chart cartoon, and then combine those graphics in rapid succession. These can then be embedded into an HTML file for viewing.

The `animation` package in `R` provides tools for constructing animated graphics. The basic constructor is the `saveHTML()` function, which takes a list of images, as well as instructions about the duration of each image, and adds the HTML code necessary for creating an animation. The burden is then to construct each individual graphic that will appear in the flip chart. Caution needs to be taken to construct the appropriate spacing. If the graph changes too much from one image to the next, it will not seem fluid. If, on the other hand, the graph changes too little, you will require a large number of images to construct the final animation.

Generally, instead of replicating code thousands of times, we construct a function which makes the incremental changes to some base graphic and updates the final product. As a simple example, consider the density of the *t-distribution* with ν degrees of freedom. It is a well-known result that as ν increases, the t-distribution converges to that of a Standard Normal. We can illustrate this process with an animation.

We can sketch out what we would like this animation to accomplish. We would like to have the density of the Standard Normal distribution permanently displayed. We also plot the density associated with the t-distribution, incrementing the degrees of freedom with each "flip."

```
# Need Initial Data to Define Scales
Normal.df <- data_frame(x = seq(-5, 5, length.out=1000),
                        fx = dnorm(x))

# Create Static Example Plot
ggplot() +
  geom_line(data=Normal.df,
            mapping=aes(x=x, y=fx, linetype="Normal", colour="Normal"),
            size=1.25) +
  stat_function(fun=dt, args=list(df=3),
                mapping=aes(linetype="t", colour="t")) +
  labs(x="", y="Density", colour="Distribution", linetype="Distribution",
       title="Comparison of Standard Normal to t-Distribution with df = 3") +
  scale_color_manual(values=c("t"="red", "Normal"="black")) +
  scale_linetype_manual(values=c("t"=2, "Normal"=1)) +
  theme_bw(12) +
  theme(legend.position="bottom")
```

Now that we have a prototype of what each frame in the animation will look like, we construct a function which constructs the above plot for a specific choice of the degrees of freedom. We will then be able to use this function to construct several plots.

```
# function: plot.t
# description: Construct the frame in the animation for a specific value of the
#              degrees of freedom.
plot.t <- function(dfs){
  # save plot to print
  out <- ggplot() +
    geom_line(data=Normal.df,
              mapping=aes(x=x, y=fx, linetype="Normal", colour="Normal"),
              size=1.25) +
    stat_function(fun=dt, args=list(df=dfs),
                  mapping=aes(linetype="t", colour="t")) +
```

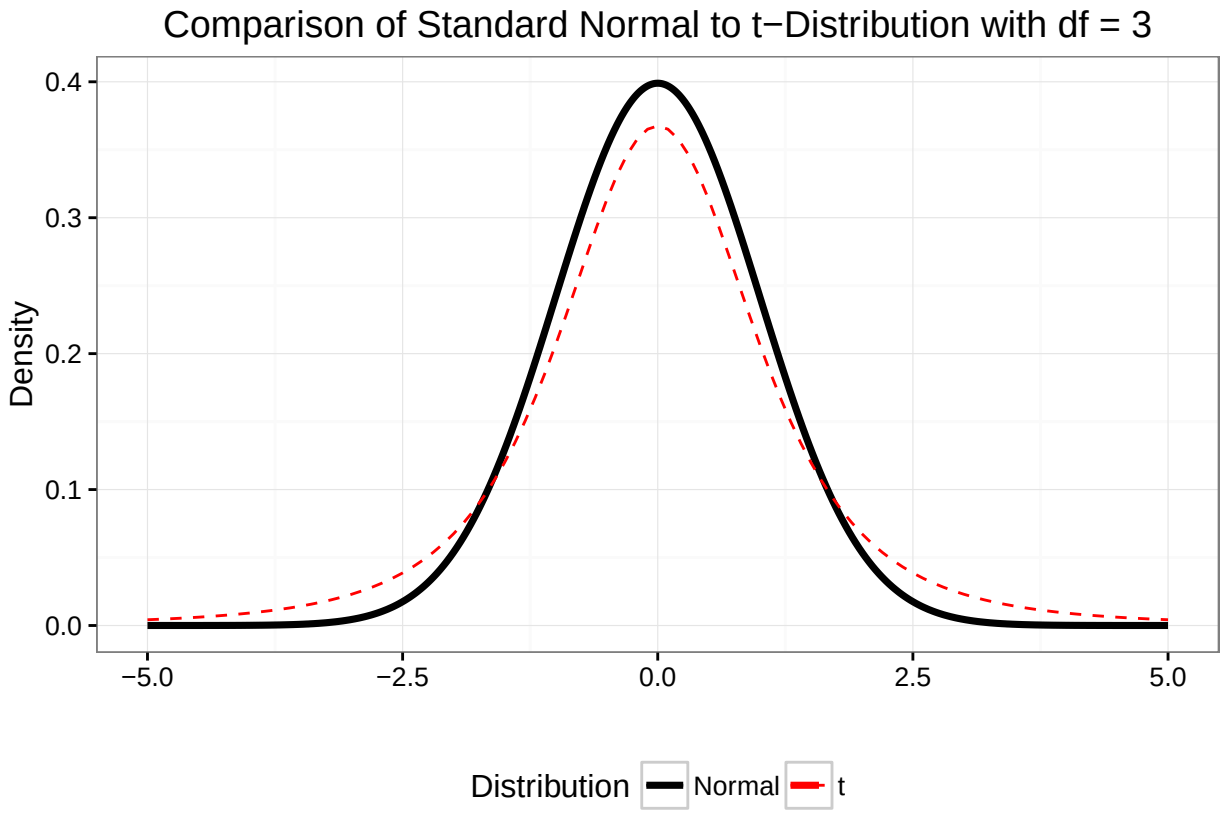


Figure 4: Example static image from a flip chart showing the convergence of the t-distribution to a Standard Normal. This acts as a sketch of what will appear in the animation.

```
    labs(x="", y="Density", colour="Distribution", linetype="Distribution",
         title=str_c("Comparison of Standard Normal to t-Distribution\nwith ",
                     "df = ", dfs)) +
    scale_colour_manual(values=c("t"="red", "Normal"="black")) +
    scale_linetype_manual(values=c("t"=2, "Normal"=1)) +
    theme_bw() +
    theme(legend.position="bottom")

  # print graph
  print(out)
}
```

Now, we use the animation package to actually construct the flipchart. The `saveHTML()` function dumps the animations to the working directory; so, we should change the working directory to the location we would like the animation to be deployed. The primary input to the function is a *list* of plots over which to iterate. Finally, we need to specify the interval between graphics.

```
# Deploy Animation
#  Write files to working directory, specifying the base name of the image.
saveHTML(lapply(seq(3, 100), plot.t),
         img.name="tAnimation",
         htmlfile="index.html",
         interval=0.5)
```

We use the `lapply()` function to construct a list by applying the `plot.t()` function we created in the previous step for values of the degrees of freedom between 3 and 100. We also specify that the primary file for lauching the animation be known as `index.html` within our directory. Additional files needed to construct the HTML are also placed in the directory. We should note that multiple animations can be combined into the same HTML file.

One drawback of this process is that each of the graphics must also be stored individually. This creates a bit of overhead. Also, this prohibits a user from downloading the animation and playing it later; they must revisit the site where it was posted (minor limitation). To overcome these limitations, the `animation` package provides the `saveGIF()` function. It works similarly to the `saveHTML()` function; however, the original files are combined into a GIF and then the original files are removed. The final movie can be uploaded to a website or saved locally. However, in order for this function to work, you must install a third party program: ImageMagick.

3 Interactivity

Summary: Creating a graphic that accepts some types of user-defined inputs allows for interactive exploratory data analysis. These graphics can also be very useful on interactive dashboards in which a user can highlight the information of interest.

Throughout this section, we will be making use of the `animint` package. This package is not currently available on CRAN (where you typically obtain packages). Instead, it must be obtained directly from the developers site on GitHub. This is accomplished using the following code:

```
# Install Helper Package, if Necessary
if (!require(devtools)) install.packages("devtools")

# Install Animint Package
devtools::install_github("tdhock/animint", upgrade_dependencies=FALSE)
```

The idea of the `animint` package is to build onto the grammar of graphics developed in `ggplot2`. It does The package implements a few additional features. Before we discuss these in depth, let's revisit the `crime` dataset which contains crimes committed in Houston over a period of several months. Consider a graphic of all crimes across Houston, color-coded by the type of crime, as shown below.

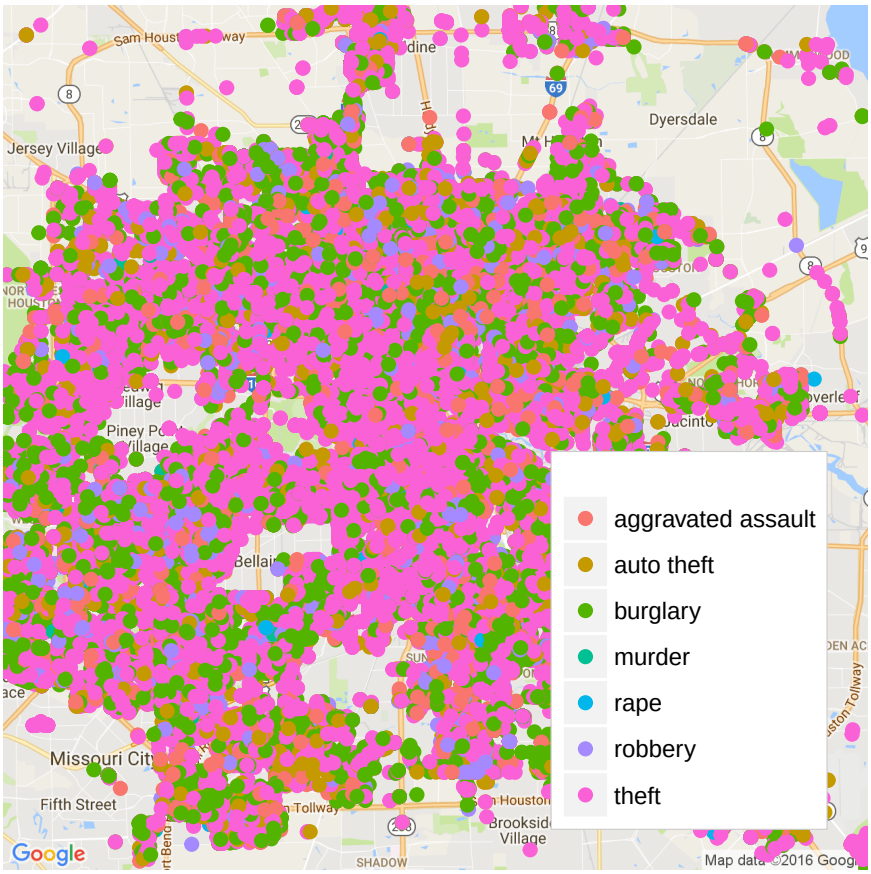


Figure 5: Static image of all crimes in Houston between January and August 2010.

This graphic is extremely cluttered, but it does have relevant information. What happens if we view this exact same graphic through the `animint` lens? Below is code which constructs the graphic and places it within the `animint` class in order to allow for some minor interactivity.

```
# Obtain Base Map
#  Obtain base map from Google, which we zoom out to city.
Houston.map <- get_map("Houston Texas", zoom=11, maptype="roadmap", color="bw")


# Create map
#  Store plot for future printing.
Houston.Crimes <- ggmap(Houston.map, legend="bottomright", extent="device") +
  geom_point(data=crime, aes(x=lon, y=lat, colour=offense),
             size=2) +
  labs(x="", y="", colour="") +
  theme(axis.ticks=element_blank(),
        axis.text=element_blank())
```

```
# Deploy Graphic
#  Add the animint class to the structure of the plot and display.
structure(list(HoustonMap = Houston.Crimes),
          class="animint")
```

Notice that in the above code to generate the plot, we did nothing special. The code is exactly what we would put together to create the static image. However, by adding some additional structure (in particular, adding a class of "animint") then additional features become available. The graphic has buttons to control animation and interactivity. Try clicking on the legend; notice that clicking toggles certain data "off" or "on" within the graphic.

The above graphic was printed to the console, which opened a server within RStudio for displaying the graphic. We can also save these images to our local directory. This is done using the `animint2dir()` function. It takes a *named* list of plots and the directory in which to store the files. As an example, the following chunk saves the Houston Crimes graphic to my local machine, the resutling file which can be opened in a web browser.

‘‘‘

```
# Export Graphic
animint2dir(list(HoustonMap=Houston.Crimes),
            out.dir="./Animations/HoustonCrimes",
            open.browser=FALSE)
```

We are then left to determine how to control and extend the interactivity added within the `animint` framework. The `animint` class adds to additional features through aesthetic mappings. These are not parsed by `ggplot2` directly, but are understood within the `animint` class. These aesthetic mappings are

- **clickSelects**: changes the currently selected value of a variable in the plot.
- **showSelected**: shows only the subset of the data that corresponds to the selected value of the variable in the plot.
- **time**: animate the plot according to this variable.
- **duration**: smooth transitions of these variables.

The last two are not aesthetics so much as additional options that are passed to the creation of the animated plot. There are other options which are also available for more precision in controlling the resulting graphic. However, these four features are the essential components. These are added by default to some degree whenever a discrete variable is used in the construction of a legend within `ggplot2`. However, their explicit use is where we gain the power of this package.

Let's consider a simple example. Consider the `tips` dataset examined in a previous unit. In brief, a waiter kept records on the amount of tips received from patrons as well as other characteristics about them, such as their smoking status. We load the data and update the day to ensure a proper ordering.

```
# Load Data
data(tips, package="reshape2")

# Update Data
tips <- tips %>%
  mutate(day = factor(day, levels=c("Thur", "Fri", "Sat", "Sun"), order=TRUE))
```

Now, consider the following static graphic.

The upper-left panel displays the number of patrons across each night. The upper-right corner displays the distribution of the tip amount, and the bottom plot shows the relationship between the tip and the total

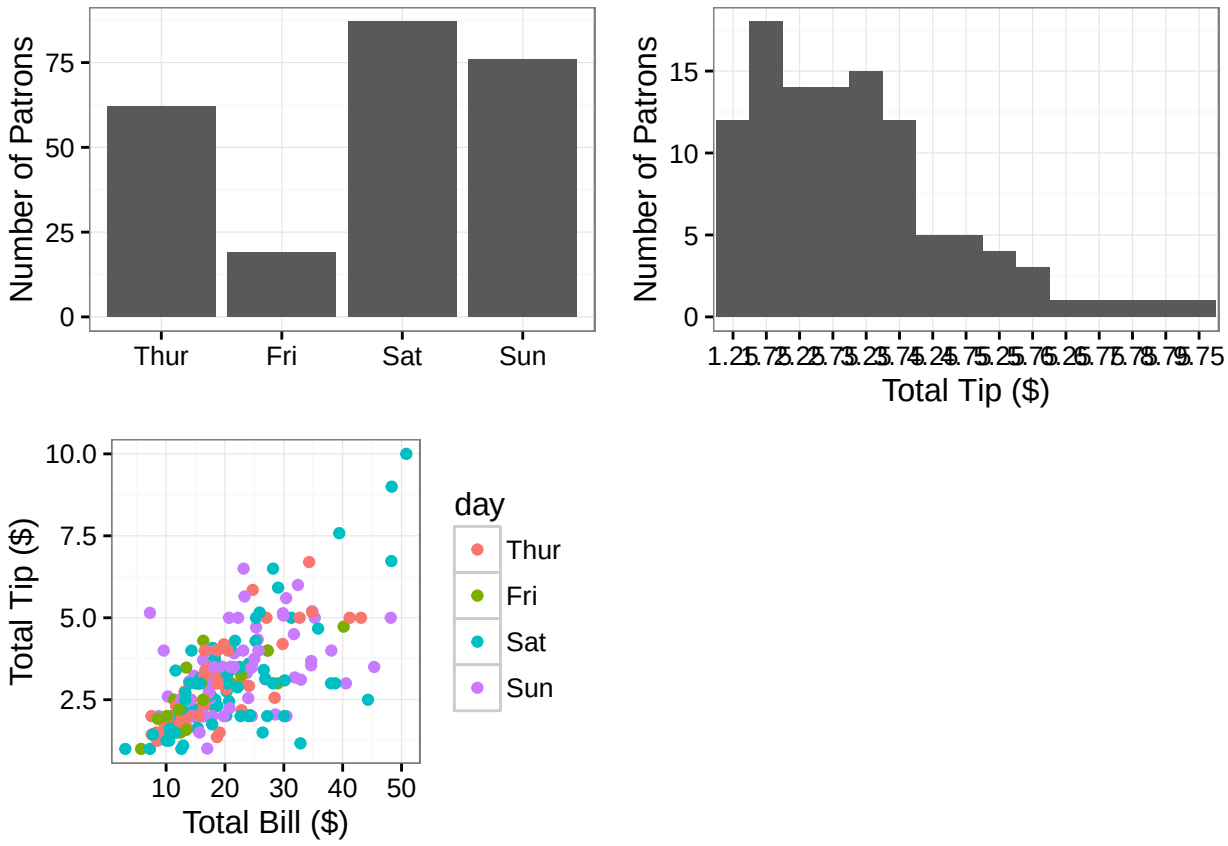


Figure 6: Static version of a summary of the smoking status and its affect on overall tip left for a waiter.

bill for both across all days of the week. We would like to update this graphic to include some interactivity. Specifically, we would like to accomplish the following tasks:

- Clicking on the legend will highlight the corresponding day of the week.
- Clicking on the day of the week in the bar chart will highlight the corresponding day of the week.
- The histogram should only show the data from the selected day, where the selection is made in the *other graphics*.
- The three graphics should be linked, as implied by the previous goals.

In order to do this, we consider building each plot individually. First, consider the bar graph.

There are a couple of differences we need to point out from a standard `ggplot` call. First, instead of allowing `geom_bar()` to summarize the data for us, we perform the summarization outside of the plot call. This requires us to ensure that no summarization is done inside the geom (using `stat = "identity"`). The new feature is the use of `clickSelects=day` as part of the aesthetic mappings. The `clickSelects` (multiple can exist) states that we should highlight the day selected by a mouse click. Next, we implement the histogram.

Again, we need to perform the summarization outside of the plot commands. The `animint` package currently does not work well with any statistical summaries of the data because it is trying to avoid additional computations. Instead, it works with data which is simply displayed on the graphic. We can only hope that in a future release this break from the `ggplot2` framework will be resolved. Currently, the suggested work-around is to construct the summaries manually. As an aside, the `ggvis` package allows for some interaction with graphics and is similar to `ggplot2` (in fact, it will probably replace `ggplot2` in the next few years). However, it currently does not allow for multiple graphics to interact with one another outside of the Shiny framework. Returning to our histogram, we break the data into multiple groups prior to passing it to the plotting feature. Since the data has already been summarized, we use `geom_bar()` instead of `geom_histogram()` for plotting. Again, we require that there be no summaries of the data. The `position="identity"` is to ensure no "stacking" is done when data share the same values across multiple groups (try removing that option and rendering the graphic to see the results). We also note that we use the `showSelected=day` aesthetic which tells the plot that only a subset of the data based on the day currently selected should be shown. Finally, we examine the scatterplot.

Since scatterplots do not perform any statistical summaries on the data, no data manipulation is needed prior to the construction of the graphic. Instead, we only need add the `clickSelects=day` aesthetic to inform the graphic that we want the day of the week highlighted in the plot. The link between these graphics is not specified in the construction itself but when rendered to the same directory.

```
structure(list(bar=tips.bar, hist=tips.hist, scatter=tips.scatter,
               selector.types=list(day="single")),
          class="animint")
```

There is one additional feature regarding the above rendering: the inclusion of `selector.types=` within the list of graphics. This is an additional option which states that only one day can be selected at a time. Selecting multiple days (while nice in theory) will not result in the appropriate histogram in this case (since `position="identity"`). Therefore, we only consider selecting one day at a time.

The above example illustrates our ability to interact with graphics, but it does not address animation. Animation in `animint` does not use the flip-chart method as described above. Instead, we essentially show subsets of a data frame and indicate the key value that connects these graphics. As a simple illustration of animation, and to provide contrast with the requirements of the flip-chart approach, we reconsider the animation showing the convergence of the t-distribution to a Standard Normal.

We first set up a dataframe which contains the density of the Normal distribution (similar to before) and another which contains similar information for the t-distribution.

```
# Normal Data
Normal.df <- data_frame(
  x = seq(-5, 5, length.out=1000),
  fx = dnorm(x),
  dens=factor("Normal", levels=c("Normal", "t")))

# T-Distribution Data
#  Construct the sequence for each potential value of degrees of freedom
t.df <- expand.grid(x = seq(-5, 5, length.out=1000),
                    df = seq(3, 100, 1)) %>%
  mutate(fx = dt(x, df=df),
         dens = factor("t", levels=c("Normal", "t"))) %>%
  arrange(df, x)
```

It is important that the dataframe for the t-distribution have the information for drawing the density for all values of the degrees of freedom. The inclusion of the factor within each dataset is not obvious here, but it will be important in order to appropriately render the legend in the graphic in a moment. Now, we are prepared to construct the graphic. Similar to the flip chart, we need to make clear which element to be selected each time. We have already touched on this above when we introduced the `showSelected` aesthetic. So, we add this to the specification of the graphic.

```
# Construct Line Plot
dens.plot <- ggplot() +
  geom_line(data=Normal.df, size=1.1,
            mapping=aes(x=x, y=fx, colour=dens, linetype=dens)) +
  geom_line(data=t.df, size=1,
            mapping=aes(x=x, y=fx, colour=dens, linetype=dens, group=df,
                        showSelected=df)) +
  geom_text(data=t.df, x=-3, y=0.3,
            mapping=aes(label=str_c("df = ", df),
                        showSelected=df)) +
  labs(x="", y="Density", colour="Distribution", linetype="Distribution") +
  scale_colour_manual(values=c("Normal"="black", "t"="red")) +
  scale_linetype_manual(values=c("Normal"=1, "t"=2)) +
  theme_bw() +
  theme(legend.position="bottom")
```

We have not used anything different than previously. The specification of the movement actually occurs when deploying the graphic.

```
# Deploy Graphic
structure(list(density=dens.plot,
               time=list(variable="df", ms=500)),
          class="animint")
```

In addition to specifying the list of graphics, we also given an option `time` which is a list identifying the variable which governs the animation and the duration between each step in milliseconds. We have not yet discussed the controls at the bottom of each `animint` graphic. These allow for direct manipulation without clicking on the graphic. It also provides the ability to step through the graphic, as well as play and pause the animation.