The “Great Restructuring” of our economy is underway. That’s the official name for what we know is happening: the best are rising to the top, and the mediocre are sinking to the bottom. It’s the Matthew Principle in-motion.

In Brynjolfsson and McAfee’s 2011 book Race Against the Machine, they detail how this New Economy will favor those that have the skill set or the capital to interface and invest in new technologies such as deep learning and robotics, which are becoming more ubiquitous every day.

Cal Newport’s Deep Work outlines two core abilities for thriving in this new economy:

1. Be able to quickly master hard things
2. Be able to produce at an elite level, in terms of both quality and speed. This need for speed (sorry) will be our focus.

Don’t repeat yourself (DRY) is a well-known maxim in software development, and most R programmers follow this rule and build functions to avoid duplicating code. But how often do you:

- Reference the same dataset in different analyses
- Create the same ODBC connection to a database
- Tinker with the same colors and themes in ggplot
- Produce markdown docs from the same template

and so on? Notice a pattern? The word “same” is sprinkled in each bullet point. I smell an opportunity to apply DRY!

If you work in a corporate or academic setting like me, you probably do these things pretty often. I’m going to show you how to wrap all of these tasks into a minimalist R package to save you time, which, as we’ve learned, is one of the keys to your success in the New Economy.

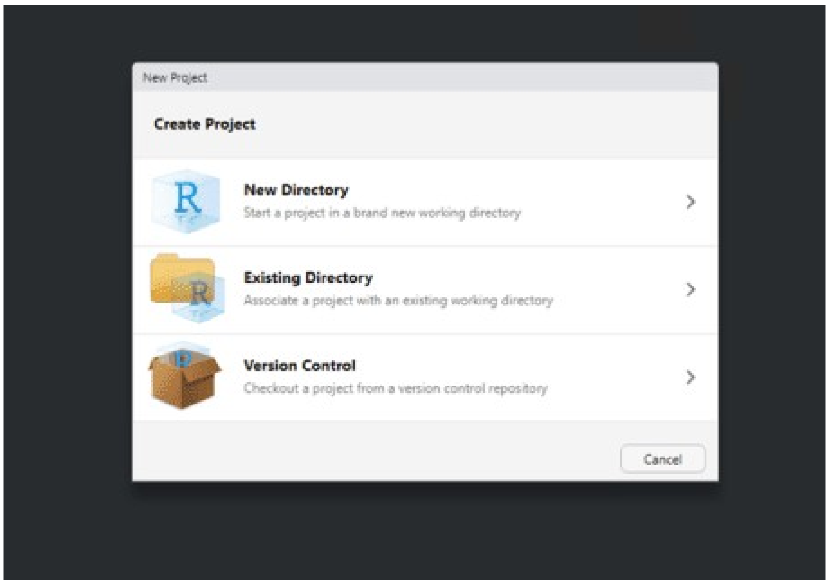
Tools

First some groundwork. I’ll assume if you work in R that you are using RStudio, which will be necessary to follow along. I’m using R version 3.5.1 on a Windows 10 machine (ahh, corporate America…). Note that the package we are about to develop is minimalist, which is a way of saying that we’re gonna cut corners to make a minimum viable product. We won’t get deep into documentation and dependencies much, as the packages we’ll require in our new package are more than likely already on your local machine.

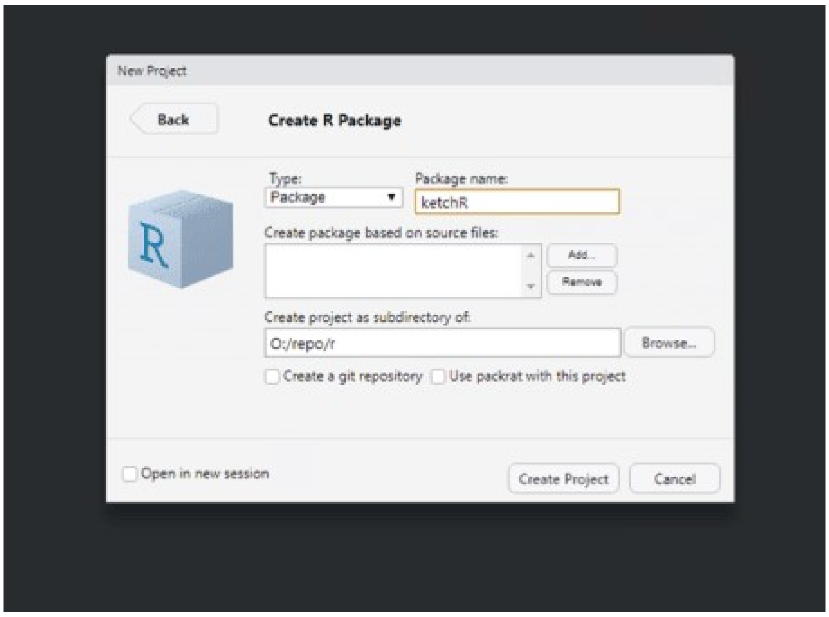
Create an empty package project

We’ll be creating a package for the consulting firm Ketchbrook Analytics, a boutique shop from Connecticut who know their way around a %>% better than anyone.

Open RStudio and create a project in a new directory:


New Project
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel

Select **R Package** and give it a name. I'll call mine ketchR.



RStudio will now start a new session with an example "hello" function. Looks like we're ready to get down to business.

Custom functions

Let's start by adding a function to our package. A common task at Ketchbrook is mapping customer data with an outline for market area or footprint. We can easily wrap that into a simple function.

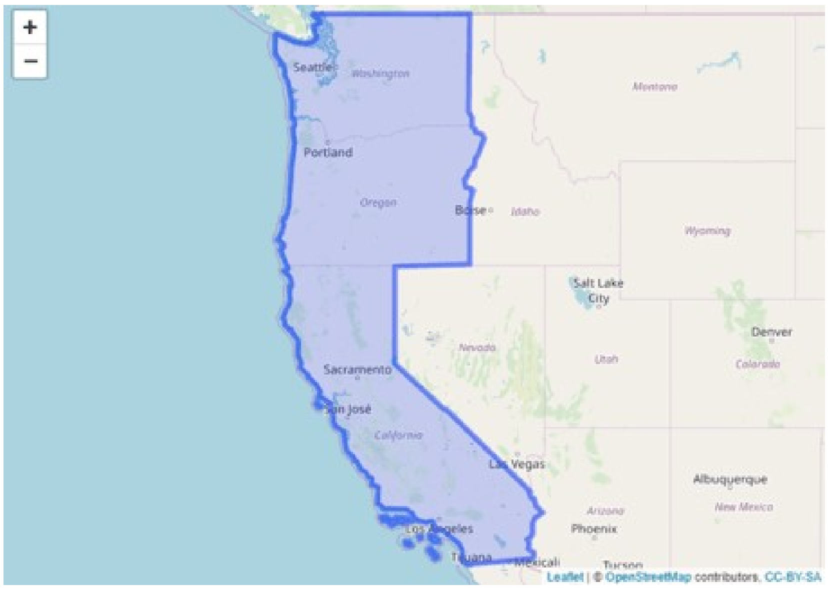
Create a new R file and name it ketchR.R. We'll put all of our functions in here.

```
# To generate the footprint_polys data
footprint_poly <- function() {

  #' Returns object of class SpatialPolygons of the AgC footprint.
  #' Utilizes the Tigris:: package.


  require(tidyverse)
  require(tigris)
  require(sf)
  # Get County Polygons
  states.raw <- tigris::states()
  states <- states.raw[states.raw@data$STUSPS %in% c("CA", "OR", "WA"),]

  states <- sf::st_as_sfc(states)
  states <- sf::st_union(states)
  states <- as(states, 'Spatial')

  return(states)


}
```

So what we've done is create a function that utilizes the tigris package to grab shapefiles for states in our footprint. The function then unions those states into one contiguous polygon so we can easily overlay this using leaflet, ggmap, etc.

Try your new function out:

```
library(leaflet)

leaflet() %>%
  addTiles() %>%
  addPolygons(data = footprint_poly())
```



There is no limit to what kinds of custom functions you can add in your package. Machine learning algs, customer segmentation, whatever you want you can throw in a function with easy access in your package.

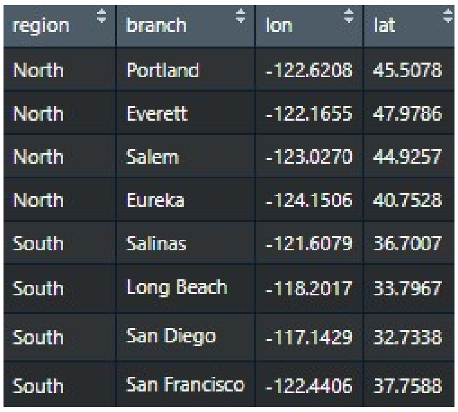
Datasets

Let's stay on our geospatial bent. Branch or store-level analysis is common in companies spread out over a large geographical region. In our example, Ketchbrook's client has eight branches from Tijuana to Seattle. Instead of manually storing and importing a CSV or R data file each time we need to reference these locations, we can simply save the data set to our package.

In order to add a dataset to our package, we first need to pull it into our local environment either by reading a csv or grabbing it from somewhere else. I simply read in a csv from my local PC:

```
branches <- read.csv("O:\\exchange\\branches.csv", header = T)
```

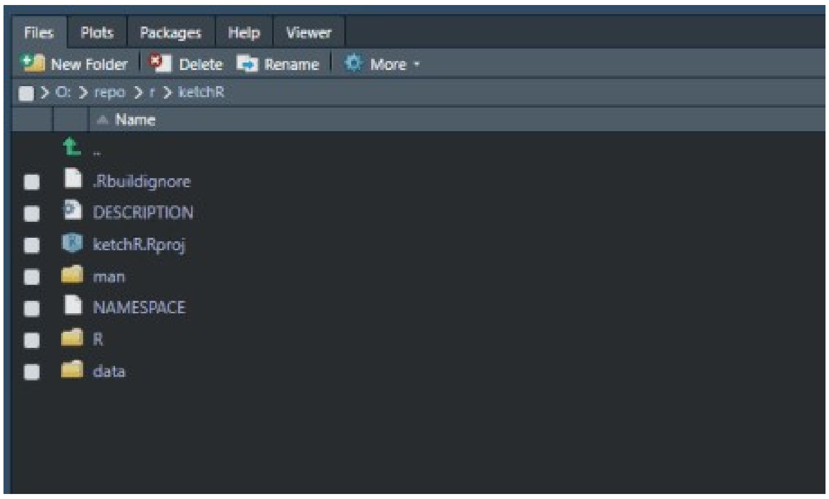
This is what the data set looks like:



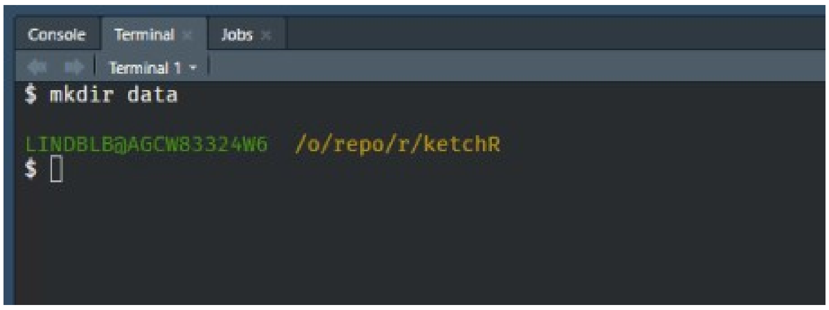
region	branch	lon	lat
North	Portland	-122.6208	45.5078
North	Everett	-122.1655	47.9786
North	Salem	-123.0270	44.9257
North	Eureka	-124.1506	40.7528
South	Salinas	-121.6079	36.7007
South	Long Beach	-118.2017	33.7967
South	San Diego	-117.1429	32.7338
South	San Francisco	-122.4406	37.7588

Now, we have to put this data in a very specific place, or our package won't be able to find it. Like when my wife hides the dishwasher so I'm reluctantly forced to place dirty dishes on the counter.

First, create a folder in your current directory called "data." Your directory should look like this now, btw:



Bonus points: use the terminal feature in RStudio to create the directory easily:



```
$ mkdir data

LINDBLB@AGCW83324W6  /o/repo/r/ketchR
$ 
```

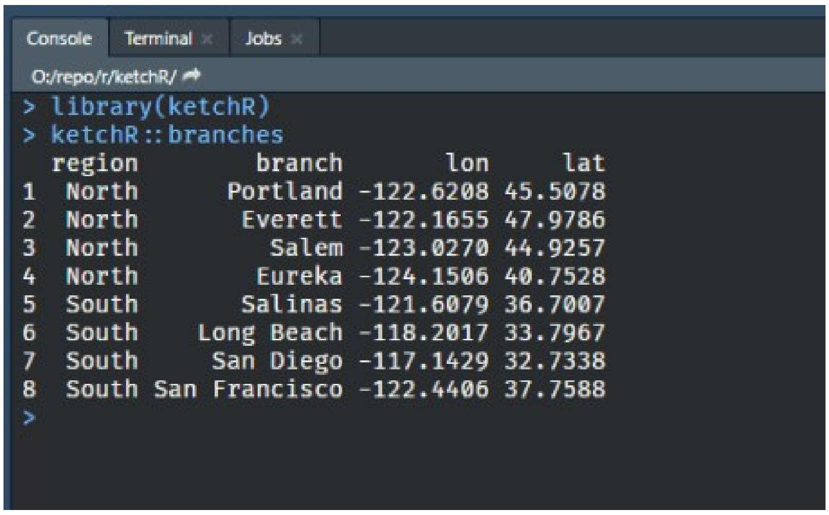
Now we need to save this branches data set into our new folder as an .RData file:

```
save(branches, file = "data/branches.RData")
```

Now, we build

Let's test this package out while there's still a good chance we didn't mess anything up. When we build the package, we are compiling it into the actual package as we know it. In RStudio, this is super simple. Navigate to the "Build" tab, and click "Install and Restart."
If you've followed along, you shouldn't see any errors, but if you do see errors, try updating your local packages.

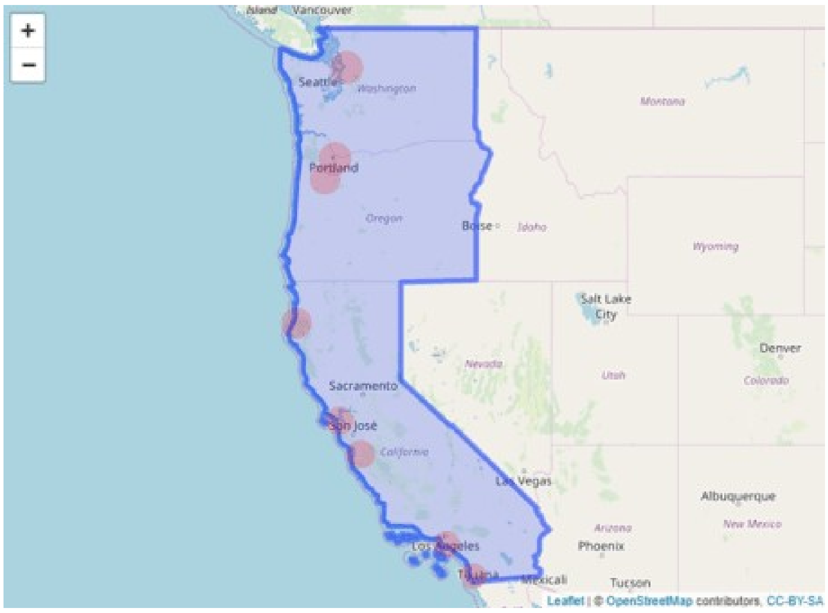
Now, we should be able to call our package directly and use our branches dataset:



```
> library(ketchR)
> ketchR::branches
  region        branch       lon     lat
1  North      Portland -122.6208 45.5078
2  North       Everett -122.1655 47.9786
3  North         Salem -123.0270 44.9257
4  North        Eureka -124.1506 40.7528
5  South       Salinas -121.6079 36.7007
6  South    Long Beach -118.2017 33.7967
7  South     San Diego -117.1429 32.7338
8  South San Francisco -122.4406 37.7588
>
```

Cool, that works. Now let’s plot our branches with Leaflet quick to make sure footprint_poly() worked:

```
library(leaflet)

leaflet() %>%
  addTiles() %>%
  addPolygons(data = ketchR::footprint_poly()) %>%
  addCircles(data = branches,
             lat = branches$lat,
             lng = branches$lon,
             radius = 40000,
             stroke = F,
             color = "red")
```



Niiiice.

Database connections

One of the most common tasks in data science is pulling data from databases. Let’s say that Ketchbrook stores data in a SQL Server. Instead of manually copy and pasting a connection script or relying on the RStudio session to cache the connection string, let’s just make a damn function.

```
get_db <- function(query = "SELECT TOP 10 * FROM datas.dbo.Customers") {

  #' Pull data from g23 database
  #' @param query: enter a SQL query; Microsoft SQL syntax please

  require(odbc)
  con <- dbConnect(odbc(),
                   Driver = "SQL Server",
                   Server = "datas",
                   Database = "dataserver",
                   UID = "user",
                   PWD = rstudioapi::askForSecret("password"),
                   Port = 6969)



  z <- odbc::dbGetQuery(con, query)

  return(z)

  odbc::dbDisconnect(con)

}
```

Here, we're building a function that lets us enter any query we want to bang against this SQL Server. The function creates the connection, prompts us to enter the password each time (we don't store passwords in code…) and closes the connection when it's through.

Let's take it a step further. Many times you may pull a generic SELECT * query in order to leverage dplyr to do your real data munging. In this case, it's easier to just make a function that does just that.

Let's make another function that pulls a SELECT * FROM Customers.

```
get_customers <- function() {

  #' Pull most recent customer data from G23 - datascience.agc_Customers

  require(odbc)
  con <- dbConnect(odbc(),
                   Driver = "SQL Server",
                   Server = "datas",
                   Database = "dataserver",
                   UID = "user",
                   PWD = rstudioapi::askForSecret("password"),
                   Port = 6969)


  query1 <- "SELECT * FROM datas.dbo.Customers"
  z <- odbc::dbGetQuery(con, query1)

  return(z)

  odbc::dbDisconnect(con)


}
```

Ahh, this alone saved me quarters-of-hours each week once I started using it in my own practice. Think hard about any piece of code that you may copy and paste on a regular basis — that's a candidate for your packages stable of functions.

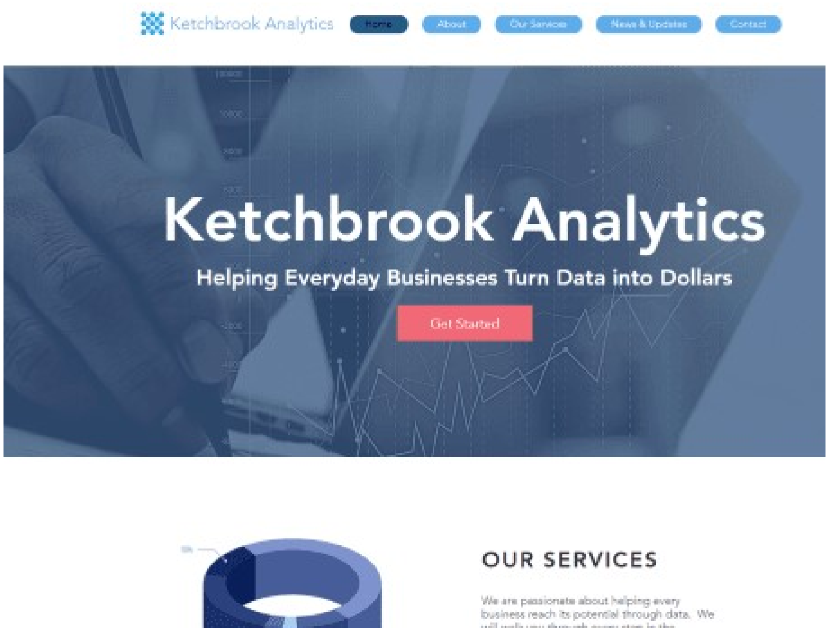
Branded ggplot visualizations

Ok now we're getting to the *primo* honey, the real time-savers, the analyst-impresser parts of our package. We're going to make it easy to produce consistent data visualizations which reflect a company's image with custom colors and themes.

Although I personally believe the viridis palette is the best color scheme of all time, it doesn't necessarily line up with Ketchbrook's corporate color palette. So let's make our own set of functions to use Ketchbrook's palette is a 'lazy' way. (Big thanks to this Simon Jackson's great article).

Get the colors

Let's pull the colors directly from their website. We can use the Chrome plugin Colorzilla to pull the colors we need.


Ketchbrook Analytics
Home
About
Our Services
News & Updates
Contact
Ketchbrook Analytics
Helping Everyday Businesses Turn Data into Dollars
Get Started
OUR SERVICES
We are passionate about helping every business reach its potential through data. We

Take those hex color codes and paste them into this chunk like so:

```
# Palette main colors
ketch.styles <- c(
  `salmon` = "#F16876",
  `light_blue`= "#00A7E6",
  `light_grey` = "#E8ECF8",
  `brown`  = "#796C68")
```

This will give us a nice palette that has colors different enough for categorical data, and similar enough for continuous data. We can even split this up into two separate sub-palettes for this very purpose:

```
# Create separate palettes
agc.palettes <- list(
  `main`  = styles('salmon','light_blue', 'brown', 'light_grey'),

  `cool`  = styles('light_blue', 'light_grey')
)
```

Create the functions

I'm not going to go through these functions line by line; if you have questions reach out to me at bradley.lindblad[at]gmail[dot]com, create an issue on the Github repo. Here is the full code snippet:

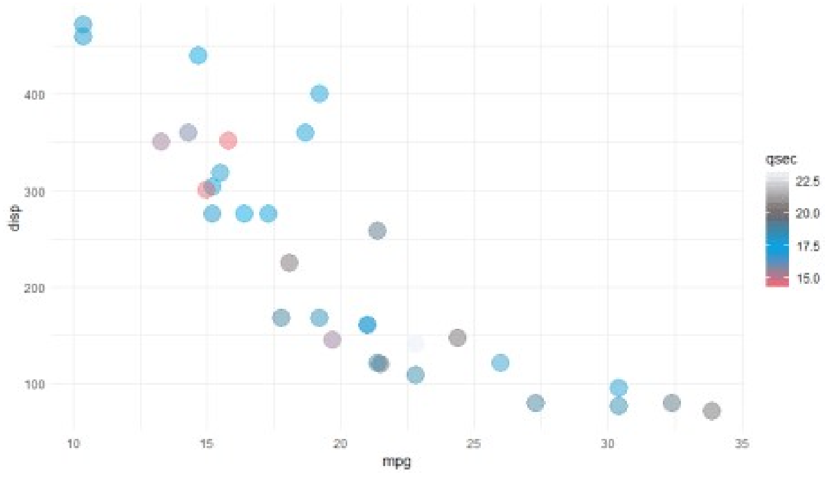
```
# Palette main colors
ketch.styles <- c(
  `salmon` = "#F16876",
```

```r
  `light_blue`= "#00A7E6",
  `light_grey` = "#E8ECF8",
  `brown`  = "#796C68")

# Fn to extract them by hex codes
styles <- function(...) {
  cols <- c(...)

  if (is.null(cols))
    return (ketch.styles)

  ketch.styles[cols]
}

# Create separate palettes
ketch.palettes <- list(
  `main`  = styles('salmon','light_blue', 'brown', 'light_grey'),

  `cool`  = styles('light_blue', 'light_grey')
)

# Fn to access them
ketch_pal <- function(palette = "main", reverse = FALSE, ...) {
  pal <- ketch.palettes[[palette]]

  if (reverse) pal <- rev(pal)

  colorRampPalette(pal, ...)
}

# Fn for customer scale
scale_color_ketch <- function(palette = "main", discrete = TRUE, reverse =
FALSE, ...) {
  pal <- ketch_pal(palette = palette, reverse = reverse)

  #' Scale color using AgC color palette.
  #' @param palette: main, greens or greys
  #' @param discrete: T or F
  #' @param reverse: reverse the direction of the color scheme

  if (discrete) {
    discrete_scale("colour", paste0("ketch_", palette), palette = pal, ...)
  } else {
    scale_color_gradientn(colours = pal(256), ...)
  }
}


scale_fill_ketch <- function(palette = "main", discrete = TRUE, reverse = FALSE,
...) {

  #' Scale fill using AgC color palette.
  #' @param palette: main, greens or greys
  #' @param discrete: T or F
  #' @param reverse: reverse the direction of the color scheme

  pal <- ketch_pal(palette = palette, reverse = reverse)
  if (discrete) {
    discrete_scale("fill", paste0("ketch_", palette), palette = pal, ...)
  } else {
    scale_fill_gradientn(colours = pal(256), ...)
  }
}
```

Let’s test it out:

```
ggplot(mtcars) +
  geom_point(aes(mpg, disp, color = qsec), alpha = 0.5, size = 6) +
  ketchR::scale_color_ketch(palette = "main", discrete = F) +
  theme_minimal()
```

produces:



Markdown templates

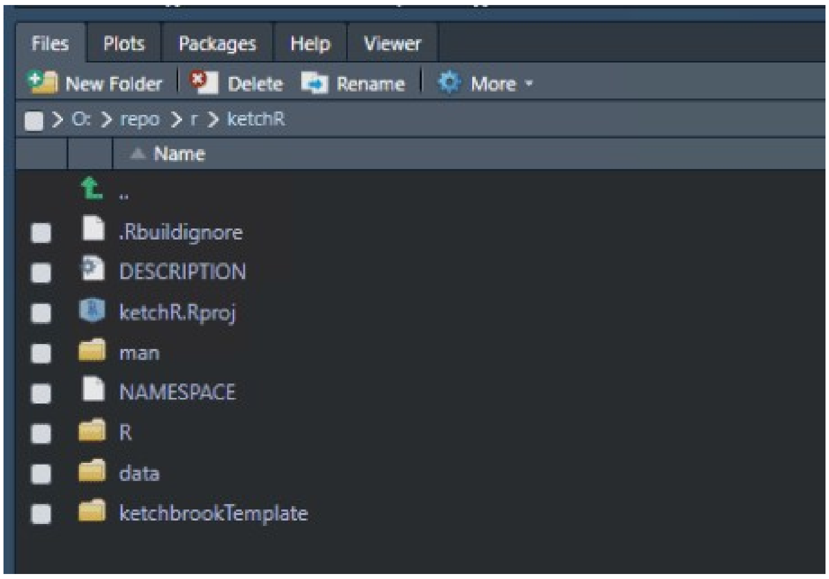
Now that we’ve fetched the data and plotted the data much more quickly, the final step is to communicate the results of our analysis. Again, we want to be able to do this quickly and consistently. A custom markdown template is in order.

I found this part to be the hardest to get right, as everything needs to be in the right place within the file structure, so follow closely. (Most of the credit here goes to this article by Chester Ismay.)

1. Create skeleton directory

```
dir.create("ketchbrookTemplate/inst/rmarkdown/templates/report/skeleton",
    recursive = TRUE)
```

This creates a nested directory that will hold our template .Rmd and .yaml files. You should have a new folder in your directory called “ketchbrookTemplate”:

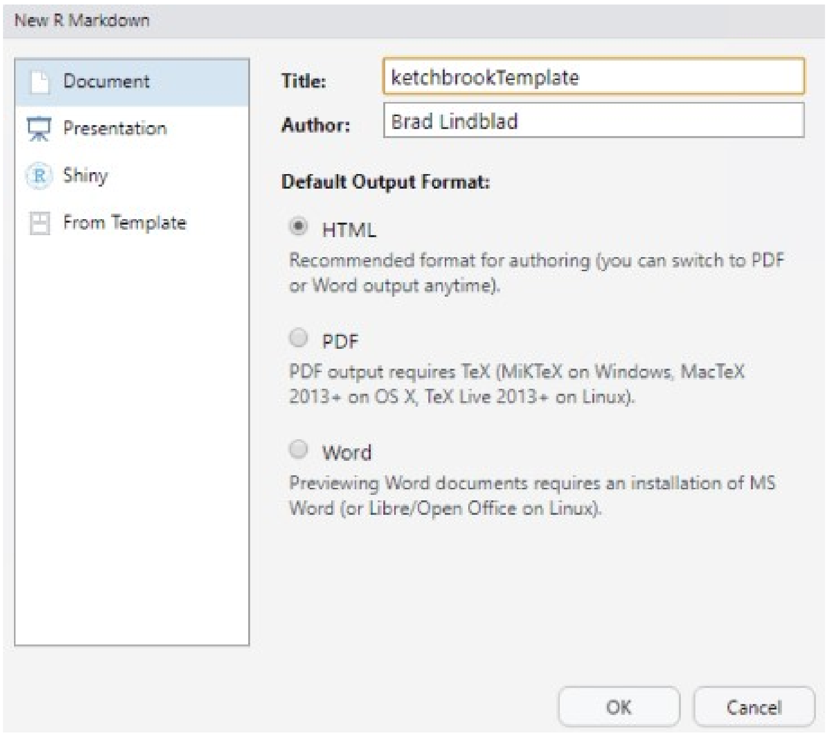


O: > repo > r > ketchR

- ..
- .Rbuildignore
- DESCRIPTION
- ketchR.Rproj
- man
- NAMESPACE
- R
- data
- ketchbrookTemplate

2. Create skeleton.Rmd

Next we create a new RMarkdown file:



Title: ketchbrookTemplate

Author: Brad Lindblad

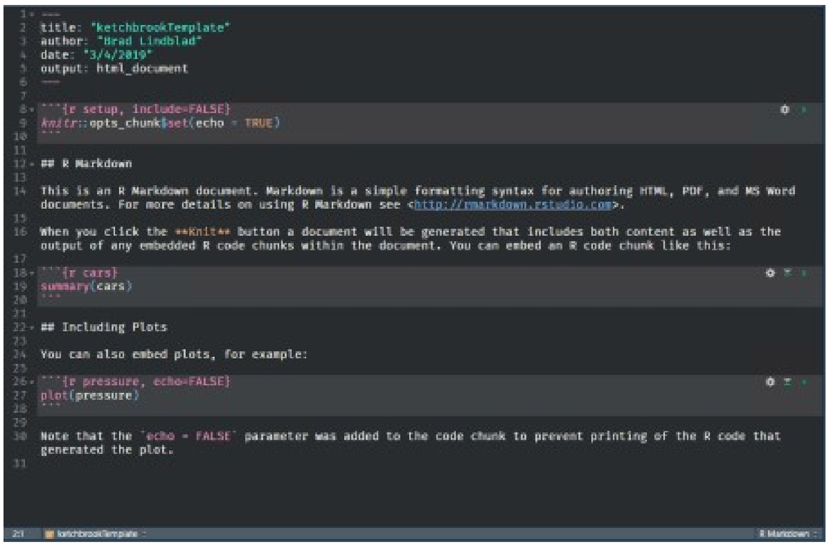
Default Output Format:

[✓ HTML]
Recommended format for authoring (you can switch to PDF or Word output anytime).

PDF
PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

Word
Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux).

This will give us a basic RMarkdown file like this:



~~~
---
title: "ketchbrookTemplate"
author: "Brad Lindblad"
date: "3/4/2019"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
~~~

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word
documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as the
output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```{r cars}
summary(cars)
```

Including Plots

You can also embed plots, for example:

```{r pressure, echo=FALSE}
plot(pressure)
```

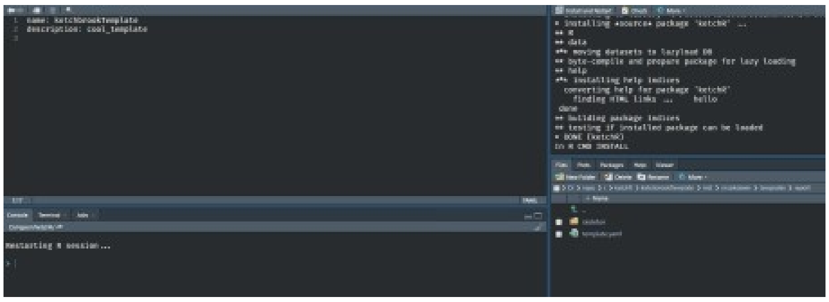
Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that
generated the plot.
```

2:1
ketchbrookTemplate
R Markdown

At this point let's modify the template to fit our needs. First I'll replace the top matter with a theme that I've found to work well for me, feel free to rip it off:

```
---
title: "ketchbrookTemplate"
author: Brad Lindblad
output:
  prettydoc::html_pretty:
    theme: cayman
    number_sections: yes
    toc: yes
  pdf_document:
    number_sections: yes
    toc: yes
  rmarkdown::html_document:
    theme: cayman
  html_notebook:
    number_sections: yes
    theme: journal
    toc: yes
header-includes:
  - \setlength{\parindent}{2em}
  - \setlength{\parskip}{0em}
date: February 05, 2018
always_allow_html: yes
#bibliography: bibliography.bib
abstract: "Your text block here"
---
```

I like to follow an analysis template, so this is the top matter combined with my basic EDA template:

```
--
title: "Customer Service Survey EDA"
author: Brad Lindblad, MBA
output:
  pdf_document:
    number_sections: yes
    toc: yes
  html_notebook:
    number_sections: yes
    theme: journal
    toc: yes
```
```

```
  rmarkdown::html_document:
    theme: cayman
  prettydoc::html_pretty:
    theme: cayman
    number_sections: yes
    toc: yes
header-includes:
  - \setlength{\parindent}{2em}
  - \setlength{\parskip}{0em}
date: September 20, 2018
always_allow_html: yes
bibliography: bibliography.bib
abstract: "Your text block here"

---

Writing Your Report
Now that you've done the necessary preparation, you can begin writing your
report. To start, keep in mind there is a simple structure you should follow.
Below you'll see the sections in order along with descriptions of each part.

Introduction

Summarize the purpose of the report and summarize the data / subject.
Include important contextual information about the reason for the report.
Summarize your analysis questions, your conclusions, and briefly outline the
report.
Body - Four Sections

Data Section - Include written descriptions of data and follow with relevant
spreadsheets.
Methods Section - Explain how you gathered and analyzed data.
Analysis Section - Explain what you analyzed. Include any charts here.
Results - Describe the results of your analysis.
Conclusions

Restate the questions from your introduction.
Restate important results.
Include any recommendations for additional data as needed.
Appendix

Include the details of your data and process here.
Include any secondary data, including references.



# Introduction

# Data

# Methods

# Analysis

# Results

# Conclusions
# Appendix

# References
```

Save this file in the skeleton folder and we're done here.

3. Create the yaml file

Next we need to create a yaml file. Simply create a new text document called "template.yaml" in RStudio and save it like you see in this picture:

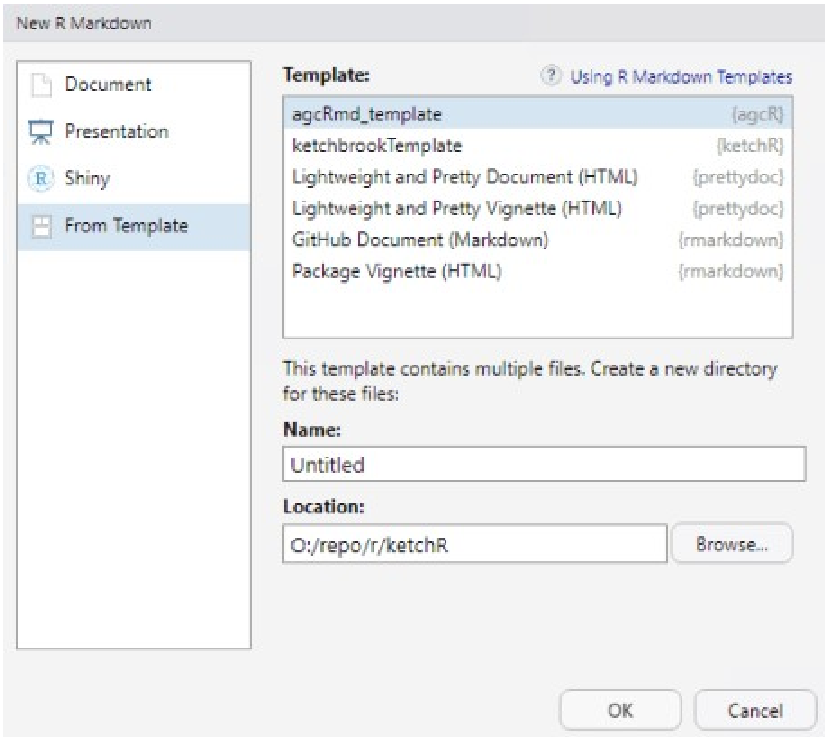


```
name: ketchbrookTemplate
description: cool_template
```

```
* installing *source* package 'ketchR' ...
** R
** data
*** moving datasets to lazyload DB
** byte-compile and prepare package for lazy loading
** help
*** installing help indices
  converting help for package 'ketchR'
    finding HTML links ...    hello
  done
** building package indices
** testing if installed package can be loaded
* DONE (ketchR)
In R CMD INSTALL
```

```
Restarting R session...

>
```

Rebuild the package and open a new RMarkdown document, select "From Template" and you should see your new template available:



New R Markdown

- Document
- Presentation
- Shiny
- From Template

Template: Using R Markdown Templates

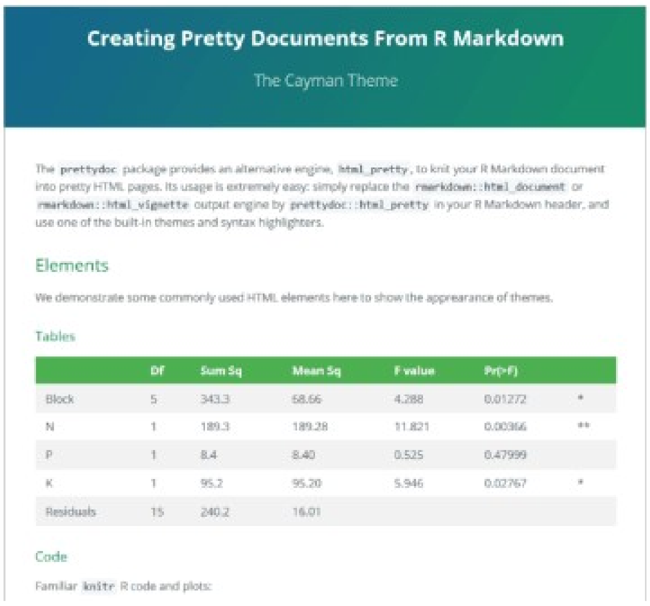
agcRmd_template	{agcR}
ketchbrookTemplate	{ketchR}
Lightweight and Pretty Document (HTML)	{prettydoc}
Lightweight and Pretty Vignette (HTML)	{prettydoc}
GitHub Document (Markdown)	{rmarkdown}
Package Vignette (HTML)	{rmarkdown}

This template contains multiple files. Create a new directory for these files:

Name: Untitled

Location: O:/repo/r/ketchR

Sweet. You can now knit to html pretty and have sweet output like this:



Creating Pretty Documents From R Markdown

The Cayman Theme

The `prettydoc` package provides an alternative engine, `html_pretty`, to knit your R Markdown document into pretty HTML pages. Its usage is extremely easy: simply replace the `rmarkdown::html_document` or `rmarkdown::html_vignette` output engine by `prettydoc::html_pretty` in your R Markdown header, and use one of the built-in themes and syntax highlighters.

Elements

We demonstrate some commonly used HTML elements here to show the appearance of themes.

Tables

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
Block	5	343.3	58.66	4.288	0.01272	*
N	1	189.3	189.28	11.821	0.00366	**
P	1	8.4	8.40	0.525	0.47999	
K	1	95.2	95.20	5.946	0.02767	*
Residuals	15	240.2	16.01			

Code

Familiar `knitr` R code and plots:

If you run into problems, make sure your file structure matches this:

```
├───inst
│   └───rmarkdown
│       └───templates
│           └───ketchbrookTemplate
│               │   template.yaml
│               │
│               └───skeleton
│                       skeleton.nb.html
│                       skeleton.Rmd
│
```