Lecture 8: So Let’s Build a Reporting Engine

A Big, Messy Survey: Continued

Setting: A real world marketing analytics project with a big company.

The Challenges:

- Understand everything about their surveys.
- Organize, clean, and process all of their data.
- Develop models for each state of customer engagement for each product.
- Identify the idiosyncratic factors that impact each model.
- Drill down into the important subgroups for each product and state of engagement.
- **Develop dynamic reporting tools to summarize all of these results.**
- AND do all of this in 10 weeks. (Yikes!)

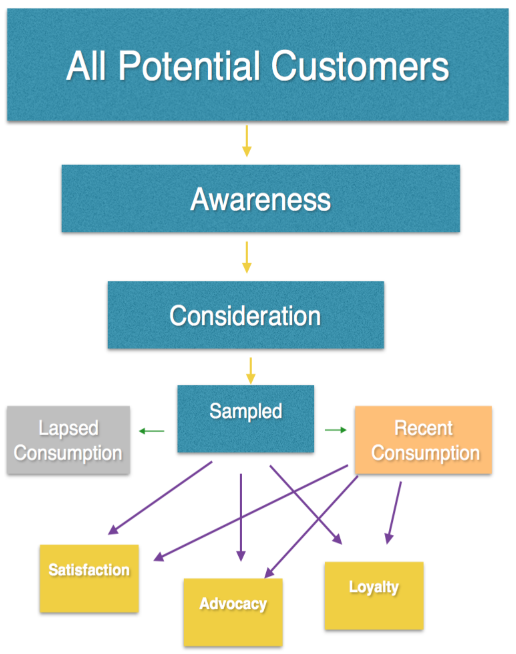
Reporting Engines

- So far we’ve done a lot of work on the marketing survey. From messy, poorly designed data, we have been able to create a more effectively organized structure.

- From that structure, we have been able to explore the relationships between the important variables and the customers’ rates of engagements. We even developed multivariable models.

- With so many products, states of engagement, and variables to consider, it would be very difficult to summarize the findings in a simple way. There is too much information for a single report. We need to build a reporting engine that will help us in a wide variety of settings.

A Reminder: States of Engagement



A Reminder: the Melted Data

Show 10 entries

Search:

id	age	gender	income	region	persona	Product	Awareness	BP_For_Me_0_10	BP_F
1	49	Male	57000	West	Millenial Muncher	Cookie_Crumble	0		
2	65	Male	133000	West	Righteous Reviewer	Cookie_Crumble	0		
3	18	Male	31000	West	Mainstream Maynard	Cookie_Crumble	0		
4	54	Female	85000	West	Mainstream Maynard	Cookie_Crumble	1	2	
5	33	Male	133000	West	Millenial Muncher	Cookie_Crumble	0		
6	64	Female	43000	West	Righteous Reviewer	Cookie_Crumble	1	5	

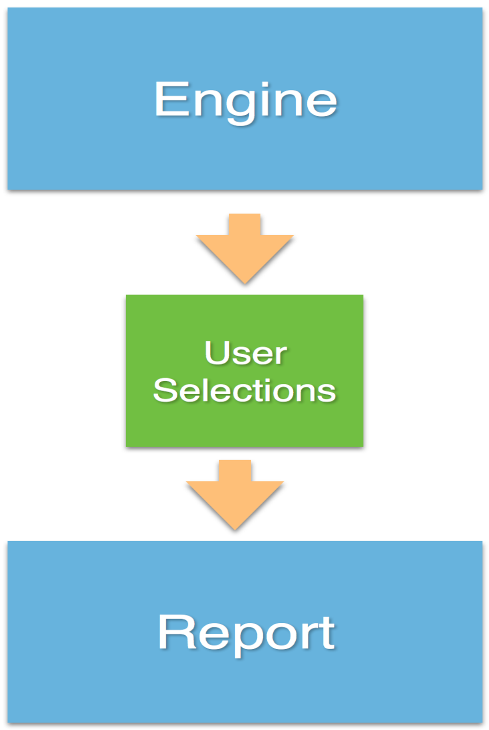
Showing 1 to 6 of 6 entries

Previous 1 Next

Static Reports

- All files are in a finished form.
- No changes to the display are made.
- The report may depend on data (e.g. writing a homework assignment in RMarkdown), but all of the outputs are pre-determined.

Dynamic Reporting



Advantages of Dynamic Reporting

- More reflective of the real world.
- Depends fundamentally on both prior information and the user's selections.
- Displays customized content.

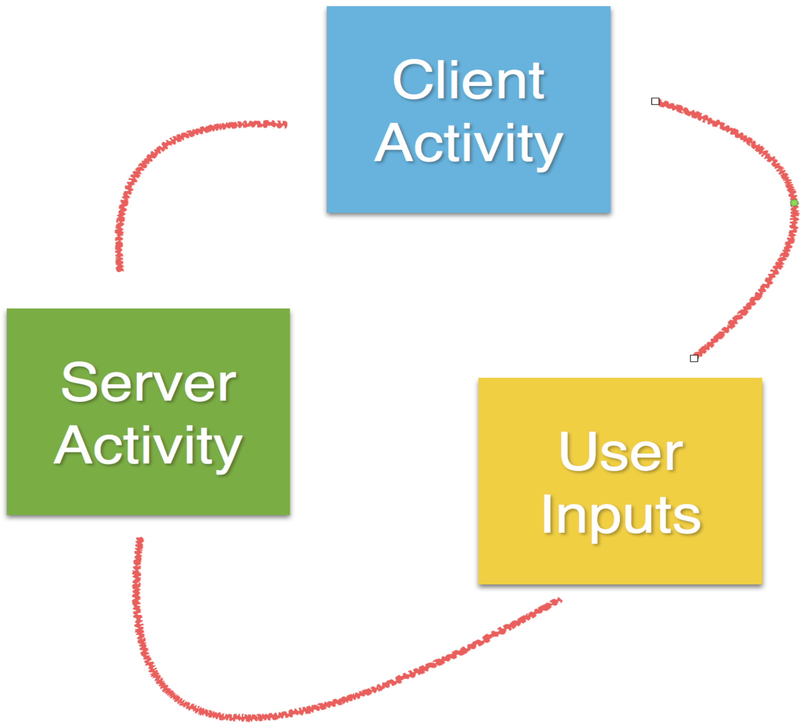
Building a Reporting Engine

- R’s **shiny** package offers a means of constructing user interfaces and working with *reactive* content.

- R’s **flexdashboard** package offers a framework for constructing webpage layouts.

- This is all integrated into **RMarkdown**, which allows for development in a reproducible framework.

The shiny Package



Software Development in RMarkdown

- You can certainly build your own Shiny application with standalone R scripts.
- These involve complicated interactions between the server and the client's activities.

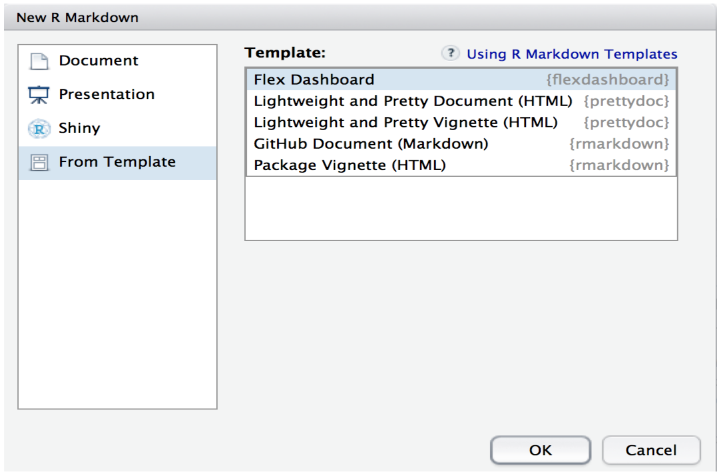
RMarkdown simplifies all of this as much as possible. You can focus more on everything else that's important:

- The analyses.
- The look and feel of the site.
- Telling the story of your work.

So Let’s Build a Reporting Engine!

Over the course of today’s lecture, we will build a reporting engine for the Marketing Analytics project. The resulting software can then be used as a prototype for the reporting engines you might build in the future.

Piece 1: Setting Up a Flexdashboard

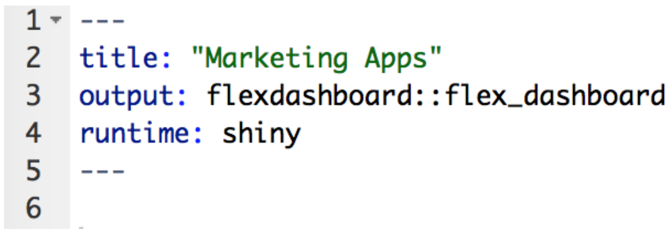


Template: Using R Markdown Templates

Flex Dashboard	{flexdashboard}
Lightweight and Pretty Document (HTML)	{prettydoc}
Lightweight and Pretty Vignette (HTML)	{prettydoc}
GitHub Document (Markdown)	{rmarkdown}
Package Vignette (HTML)	{rmarkdown}

Or you can start from a completely blank .Rmd file.

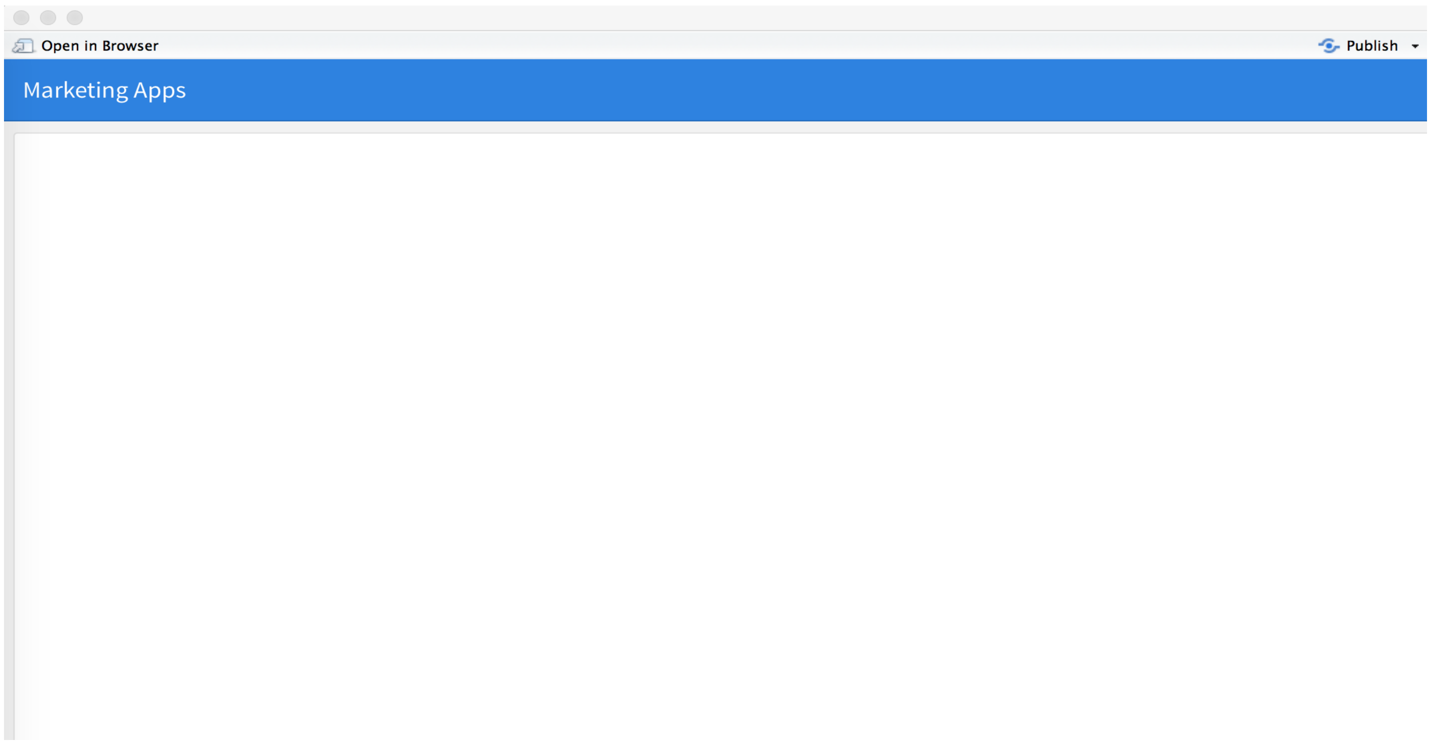
The High Level Parameters



```
---
title: "Marketing Apps"
output: flexdashboard::flex_dashboard
runtime: shiny
---

```

- Title: This will show up on your page in the upper left corner.
- Output: What does the compiled document look like? Previously we saw how to build word processing documents, PDF files, and HTML pages. The above setting will create a Flex Dashboard.
- Runtime: This specifies that the shiny package will be used to handle reactive content.

Piece 1's Output



Piece 2: RMarkdown's Setup

```
library(flexdashboard)
library(shiny)
library(rmarkdown)
library(knitr)
library(Hmisc)
library(DT)

library(data.table)
assignInNamespace(x = "cedta.override", value = c(data.table:::cedta.override,"rmarkdown"), ns = "data.table")

opts_chunk$set(echo = FALSE, comment="", warning = FALSE, message = FALSE, tidy.opts=list(width.cutoff=55), tidy = TRUE)
```

options for code truck

Components of the Setup

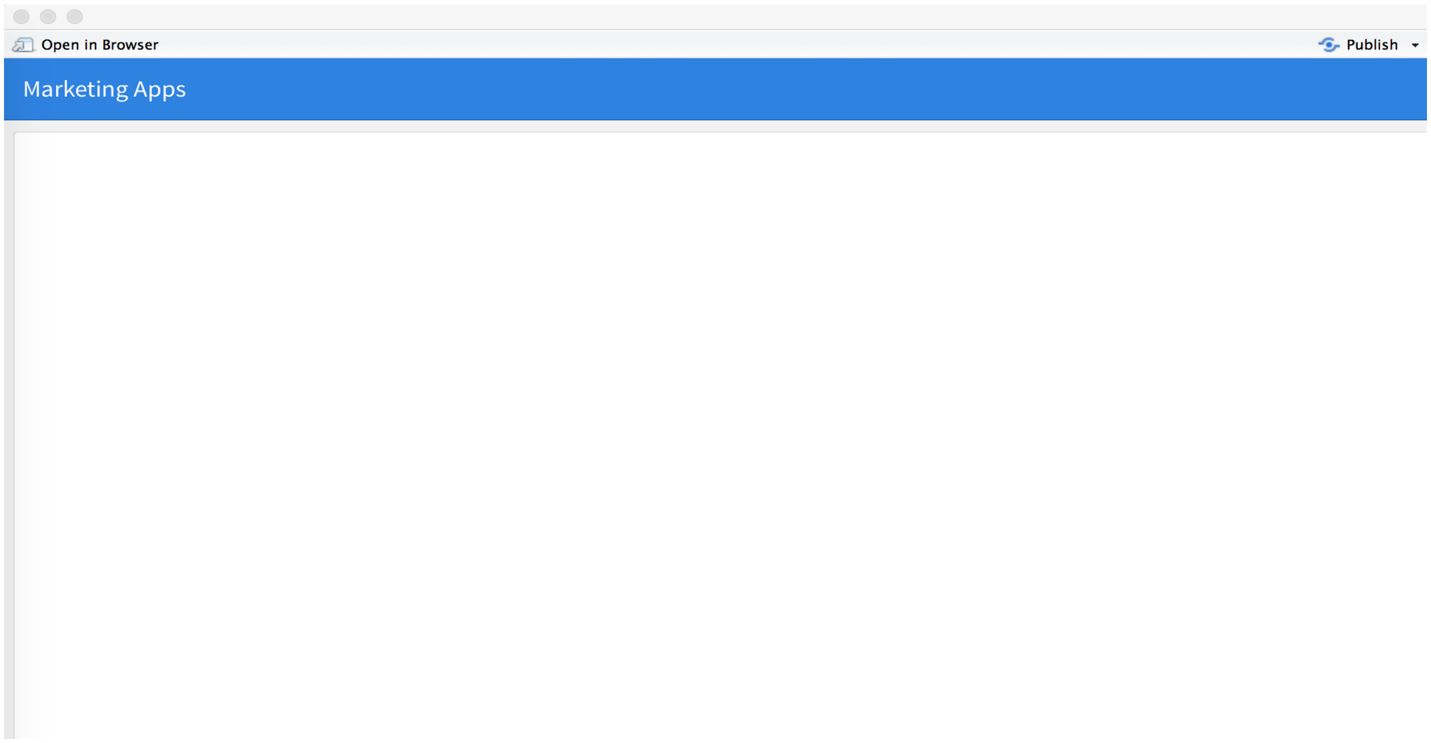
- Loading the R packages you need with the **library** command.
- Setting up the **options** for the code chunks.
- A small technical workaround…

The Technical Workaround

- RMarkdown and data.table have some small, technical incompatibilities.
- RMarkdown's environments – the frame of reference for the code – differs from that of the data.table package.
- This can all be resolved with one line of code:

```
library(data.table)
assignInNamespace(x = "cedta.override", value = c(data.table:::cedta.override,"rmarkdown"), ns = "data.table")
```

Output from Pieces 1-2



Nothing in this step changed the output in any way.

Additional Functions

- The **functions** code chunk includes a variety of functions for later use in the reporting engine.
- The majority of these functions are familiar to us from the earlier lectures.
- A few of the functions (e.g. reduce.formula, create.formula) have some updates to account for new challenges. We'll be discussing these issues more in the next lecture.

Additional Variables

- Then we have a **constants** code chunk:

```
id.name <- "id"
age.name <- "age"
gender.name <- "gender"
income.name <- "income"
region.name <- "region"
persona.name <- "persona"

product.name <- "Product"
awareness.name <- "Awareness"
consideration.name <- "Consideration"
consumption.name <- "Consumption"
satisfaction.name <- "Satisfaction"
advocacy.name <- "Advocacy"

bp.pattern <- "BP_"

age.group.name <- "age_group"
income.group.name <- "income_group"

cuts.age <- c(18, 35, 50, 65, 120)
cuts.income <- 1000* c(0, 25, 50, 75, 100, 200)
```

Piece 3: Reading the Data

We have our usual way of reading in a .csv file:

```
dat <- fread(input = "Simulated Marketing Data -- Melted.csv", verbose = FALSE)
```

More Variables

- Then, after reading in the data, we can define these variables:

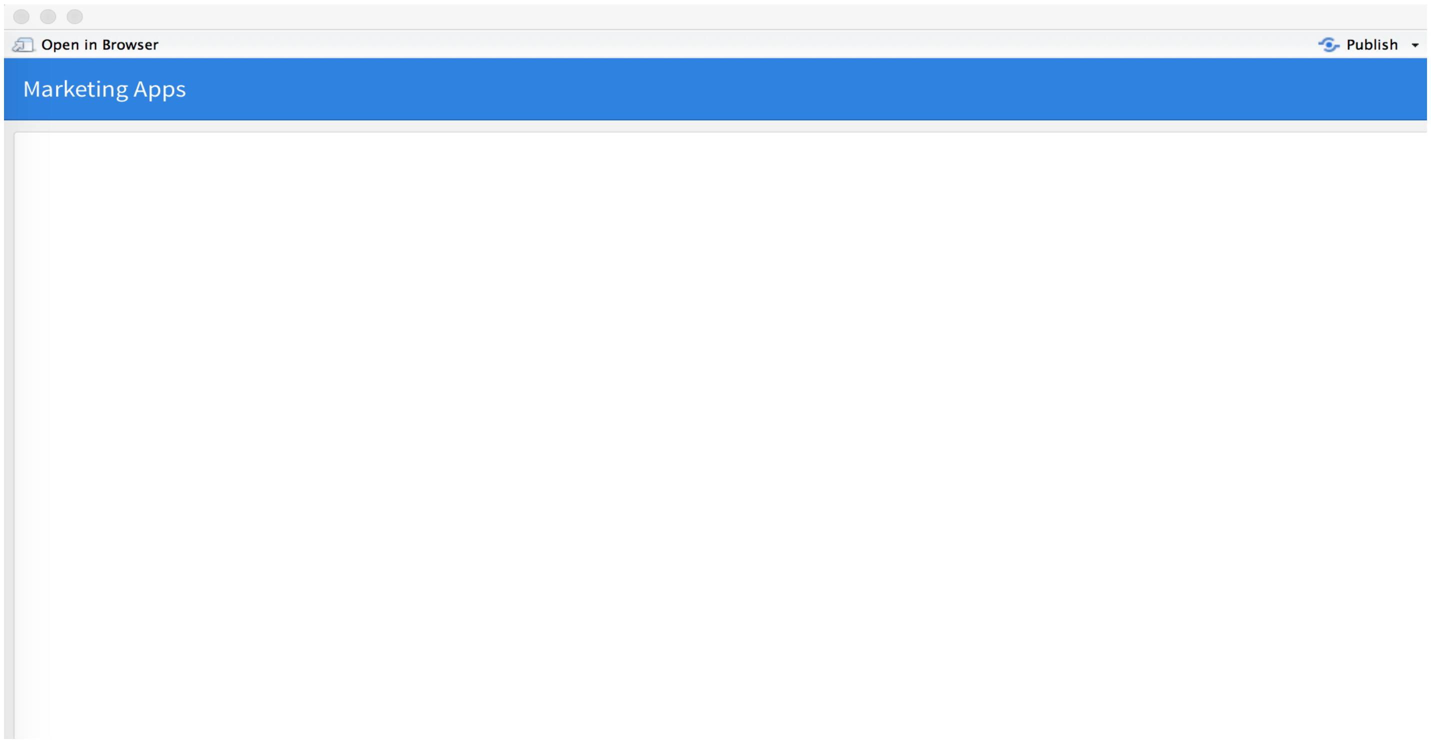
在原始数据中添加新列

```
dat[, eval(age.group.name) := cut2(x = get(age.name), cuts = cuts.age)]
dat[, eval(income.group.name) := cut2(x = get(income.name), cuts = cuts.income)]
dat[, eval(satisfaction.name) := get(satisfaction.name) / 10]

unique.age.groups <- dat[, sort(unique(get(age.group.name)))]
unique.genders <- dat[, sort(unique(get(gender.name)))]
unique.income.groups <- dat[, sort(unique(get(income.group.name)))]
unique.regions <- dat[, sort(unique(get(region.name)))]
unique.personas <- dat[, sort(unique(get(persona.name)))]

unique.products <- dat[, unique(get(product.name))]

respondent.variables <- c(age.group.name, gender.name, income.group.name, region.name, persona.name)
states.of.engagement <- c(awareness.name, consideration.name, consumption.name, satisfaction.name, advocacy.name)
bp.traits <- names(dat)[grep(pattern = bp.pattern, x = names(dat))]
```

由小到大输出我们感兴趣的量的类别

创建之后要用到的类

Output From Pieces 1-3



Nothing in this step changed the output in any way.

Piece 4: Tabs

- With the preliminary pieces in place, now we are ready to think about what information we want to report on.
- The **tabs** of a web interface are good ways of organizing different kinds of information.

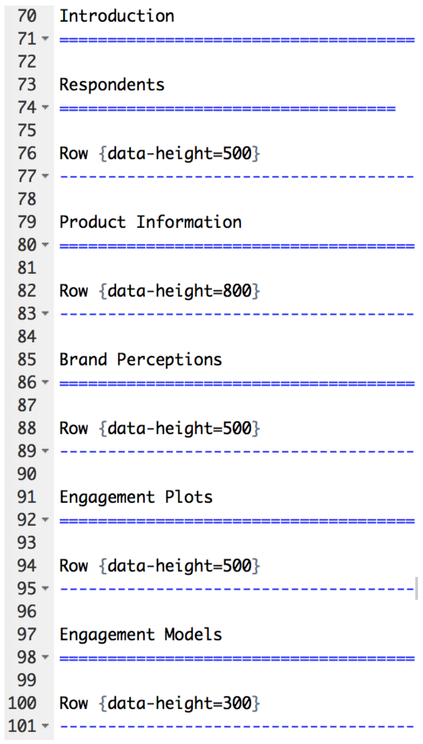
Deciding on the Tabs

We will summarize the report with the following tabs:

- **Introduction**: Some basic info.
- **Respondents**: Exploring the person-specific variables.
- **Product Information**: Exploring the brand-specific variables.
- **Brand Perceptions**: Exploring the respondents' perceptions of each brand.
- **Engagement Plots**: Investigating the states of engagement, rates, and subgroups.
- **Engagement Models**: Multivariable models for the states of engagement.

RMarkdown Code for the Tabs

给report设定子标题

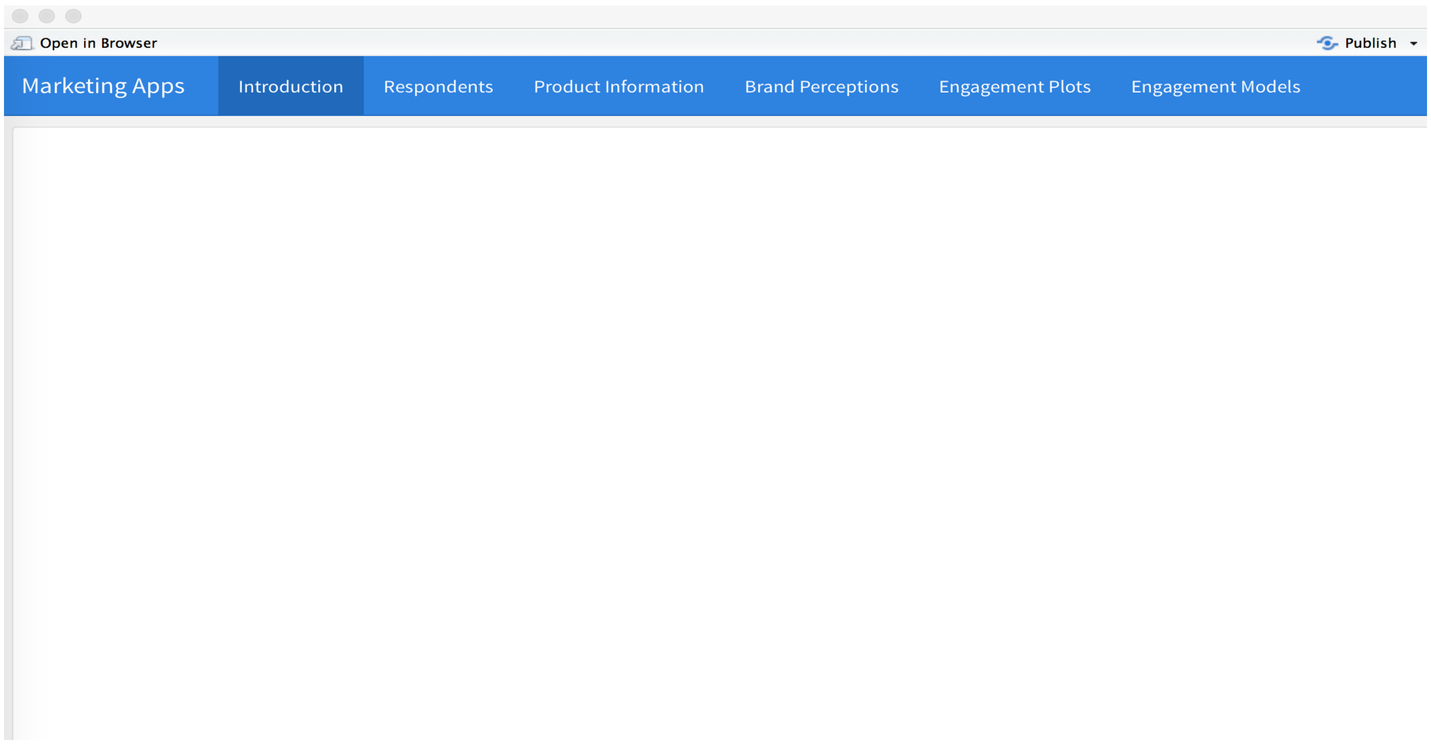


```
Introduction
=====================================

Respondents
===================================

Row {data-height=500}
-------------------------------------

Product Information
=====================================

Row {data-height=800}
-------------------------------------

Brand Perceptions
=====================================

Row {data-height=500}
-------------------------------------

Engagement Plots
=====================================

Row {data-height=500}
-------------------------------------

Engagement Models
=====================================

Row {data-height=300}
-------------------------------------
```

A Tab’s Layout

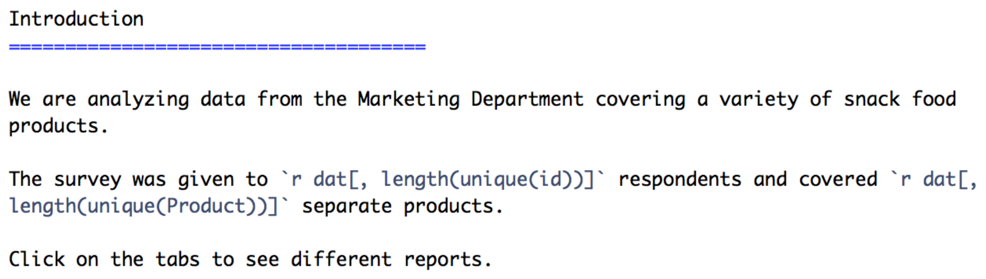
- The title of the tab is specified over the row of equal signs ====
- Each tab is divided, roughly into a 1000x1000 grid of pixels.
- The tab can be divided into rows and columns of specified sizes.

Output From Pieces 1-4



Piece 5: The Introduction Tab

This tab is meant to provide a brief overview of the data:

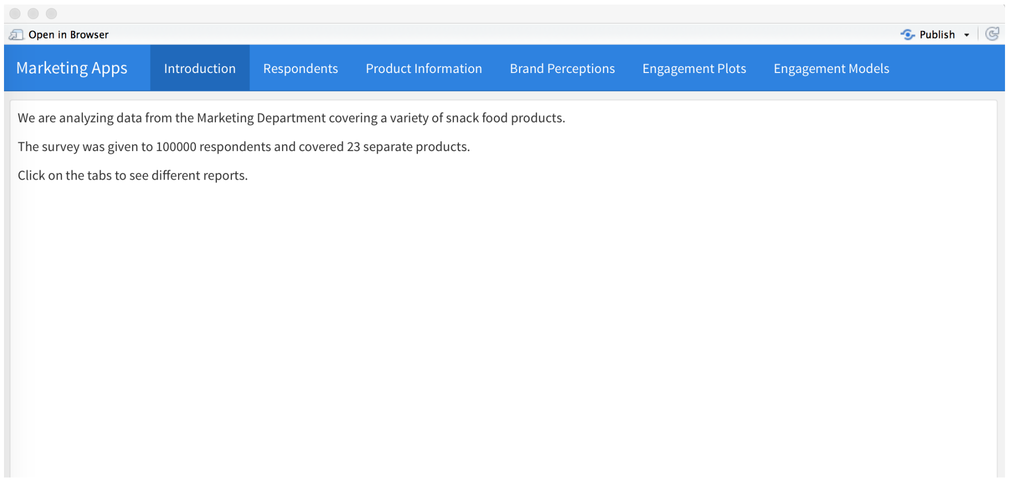


```
Introduction
=====================================

We are analyzing data from the Marketing Department covering a variety of snack food
products.

The survey was given to `r dat[, length(unique(id))]` respondents and covered `r dat[,
length(unique(Product))]` separate products.

Click on the tabs to see different reports.
```

Note that this tab is written just like a regular RMarkdown report. There is not any dynamic content here.

可以在文字部分以公式形式取数

Output from Pieces 1-5



We are analyzing data from the Marketing Department covering a variety of snack food products.

The survey was given to 100000 respondents and covered 23 separate products.

Click on the tabs to see different reports.

Piece 6: The Respondents Tab

This tab is meant to summarize the person-specific variables in the data:

- Age Group
- Gender
- Income Group
- Region
- Persona

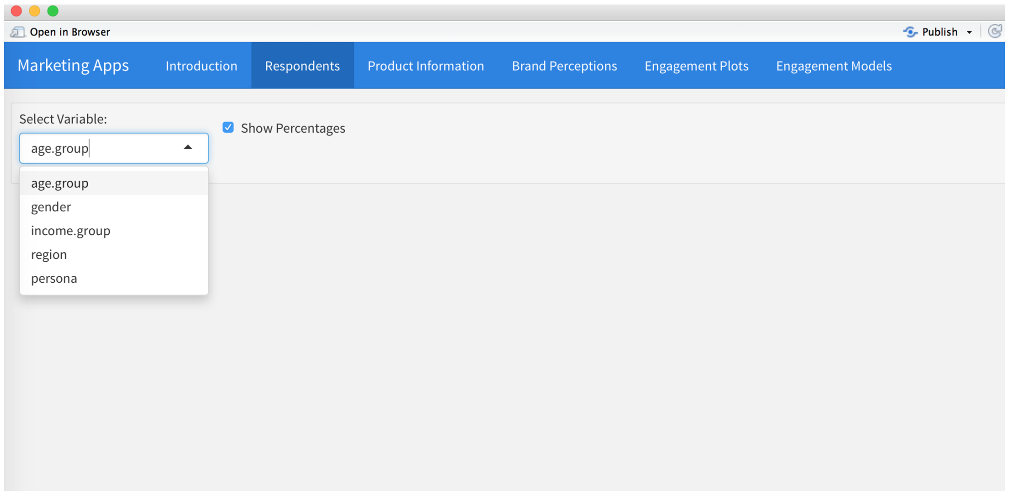
Too Much Information

- We really can’t summarize all of these variables in one plot.
- However, we can go one variable at a time.
- This is a great place to ask the user which variable to summarize.

An Input Panel

selectinput为下拉菜单中的内容，
checkboxinput 为右侧勾选菜单中的内容

```
inputPanel(
  selectInput(inputId="respondent_variable", label = "Select Variable:", choices = respondent.variables, selected =
 respondent.variables[1]),
  checkboxInput(inputId = "respondent_show_percentages", label = "Show Percentages", value = TRUE)
)
```



Two Inputs Requested

The panel is asking for two pieces of information:

- Which variable to select;
- Whether to display the percentages (as text on a bargraph).

Input Mechanics

- RMarkdown creates a global variable called **input**, which is a **list** object.
- Each item in an input panel creates a subvariable within the **input** object.
- Each new item has a name and a value:

```
input <- list(respondent_variable = respondent.variables[1], respondent_show_percentages = TRUE)
print(input)
```

```
$respondent_variable
[1] "age_group"

$respondent_show_percentages
[1] TRUE
```

- I recommend always including an example calculation of the input variable for each Input Panel. Write this as a commented line of code in your reporting engine that won't run. It will make testing your work much easier!

Dropdown Menus with selectInput

The first line of the input panel selects the variable to summarize:

inputID下拉菜单里面的内容

```
selectInput(inputId="respondent_variable", label = "Select Variable:", choices = respondent.variables, selected =
  respondent.variables[1])
```

- Creates a dropdown menu with the **label** “Select Variable:” on the screen.
- Fills the menu with the **choices** contained in **respondent.variables**: age.group, gender, income.group, region, and persona.
- Assigns the chosen value of the dropdown menu to the value of **input$respondent_variable**.
- As a default value, uses the **selected** value corresponding to the first entry of respondent.variables. That is, **input$respondent_variable** has the value “age.group” unless another item is selected by the user.

Yes/No Menus with checkboxInput

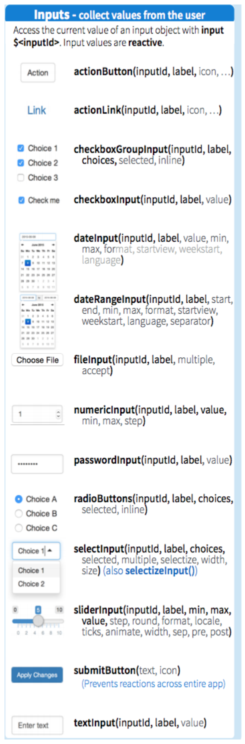
The first line of the input panel selects the variable to summarize:

```
checkboxInput(inputId = "respondent_show_percentages", label = "Show Percentages", value = TRUE)
```

- Creates a checkbox with the **label** “Show Percentages” on the screen.
- Assigns the chosen value of the dropdown menu to the value of **input$respondent_show_percentages**:
  1. *Checked: TRUE*
  2. *Unchecked: FALSE*
- As a default value, uses the specified **value** (in this case TRUE). That is, **input$respondent_show_percentages** has the value TRUE unless the user unchecks the box.

Many Other Kinds of Input Functions:

[http://shiny.rstudio.com/images/shiny-cheatsheet.pdf]

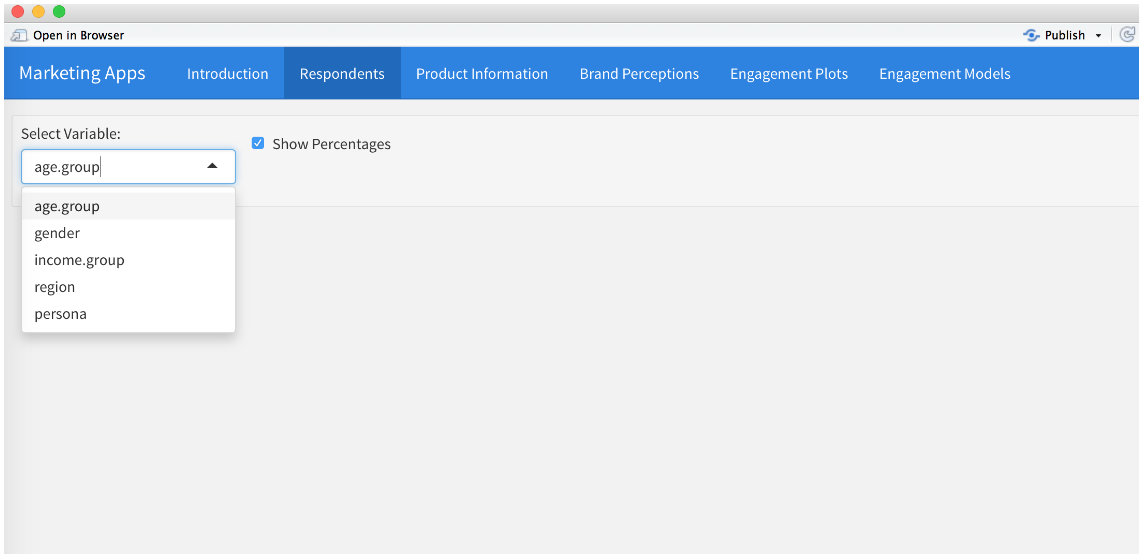


Inputs - collect values from the user

Access the current value of an input object with **input $<inputId>**. Input values are **reactive**.

Action	**actionButton(inputId, label,** icon, …**)**
Link	**actionLink(inputId, label,** icon, …**)**
✓ Choice 1 ✓ Choice 2 ☐ Choice 3	**checkboxGroupInput(inputId, label, choices,** selected, inline**)**
✓ Check me	**checkboxInput(inputId, label,** value**)**
	dateInput(inputId, label, value, min, max, format, startview, weekstart, language**)**
	dateRangeInput(inputId, label, start, end, min, max, format, startview, weekstart, language, separator**)**
Choose File	**fileInput(inputId, label,** multiple, accept**)**
1	**numericInput(inputId, label, value,** min, max, step**)**
••••••••	**passwordInput(inputId, label,** value**)**
● Choice A ○ Choice B ○ Choice C	**radioButtons(inputId, label, choices,** selected, inline**)**
Choice 1 Choice 1 Choice 2	**selectInput(inputId, label, choices,** selected, multiple, selectize, width, size**)** (also **selectizeInput()**)
0 5 10 0 2 4 6 8 10	**sliderInput(inputId, label, min, max, value,** step, round, format, locale, ticks, animate, width, sep, pre, post**)**
Apply Changes	**submitButton(**text, icon**)** (Prevents reactions across entire app)
Enter text	**textInput(inputId, label,** value**)**

Output from Pieces 1-6



Select Variable:

age.group

- age.group
- gender
- income.group
- region
- persona

☑ Show Percentages

Piece 7: The Respondents Output

- Now that we have the input panel, the user can make some selections.
- We need to be able to respond to these inputs.

In this case, we want to:

- Take the selected variable;
- Create a barplot graphing the percentage of respondents in each category; and
- Display the percentages above the graph if the checkbox is checked.

Reactive Content with renderPlot

一个表格的每一列计算百分比

User's input variable

```
renderPlot({
  tab <- percentage.table(x = dat[get(product.name) == get(product.name)[1], get(input$respondent_variable)])
  barplot(height = tab, space=0.01, las = 1, main = input$respondent_variable, ylab = "Percentage", xlab = input$respondent_variable,
 ylim = c(0, 1.2*max(tab, na.rm = TRUE)), col = "dodgerblue")

  if(input$respondent_show_percentages == TRUE){
    space_val = 0
    text(x = -0.4 + 1:length(tab) * (1+space_val), y = tab, labels = sprintf("%.1f%%", tab), pos = 3)
  }
})
```

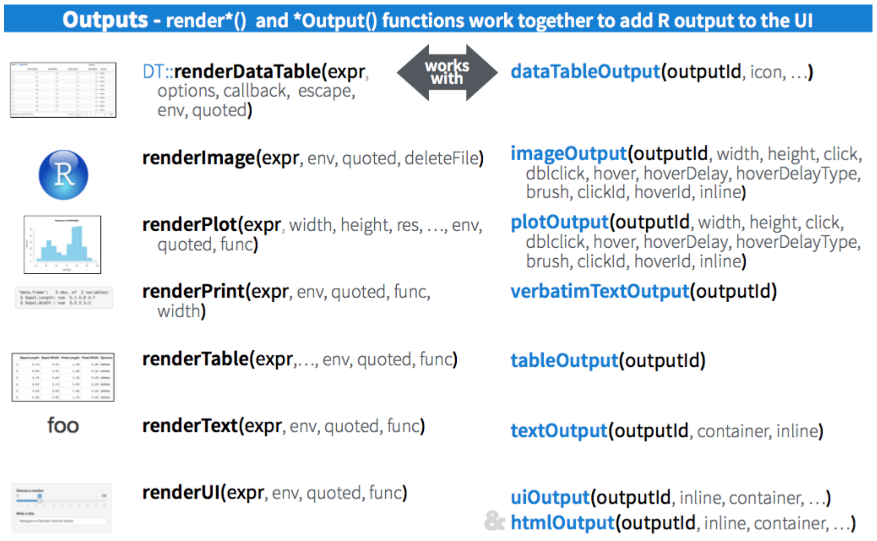
add percentage to the plot

- The first line uses **get** on the user's selected variable.
- Percentages for each category are computed and graphed.
- The percentages are also displayed as text **if** the checkbox is checked.

```
percentage.table <- function(x, digits = 1){
  tab <- table(x)
  percentage.tab <- 100*tab/(sum(tab))
  rounded.tab <- round(x = percentage.tab, digits = digits)
  return(rounded.tab)
}

round.numerics <- function(x, digits){
  if(is.numeric(x)){
    x <- round(x = x, digits = digits)
  }
  return(x)
}
```

Many Other Kinds of Rendering Functions:

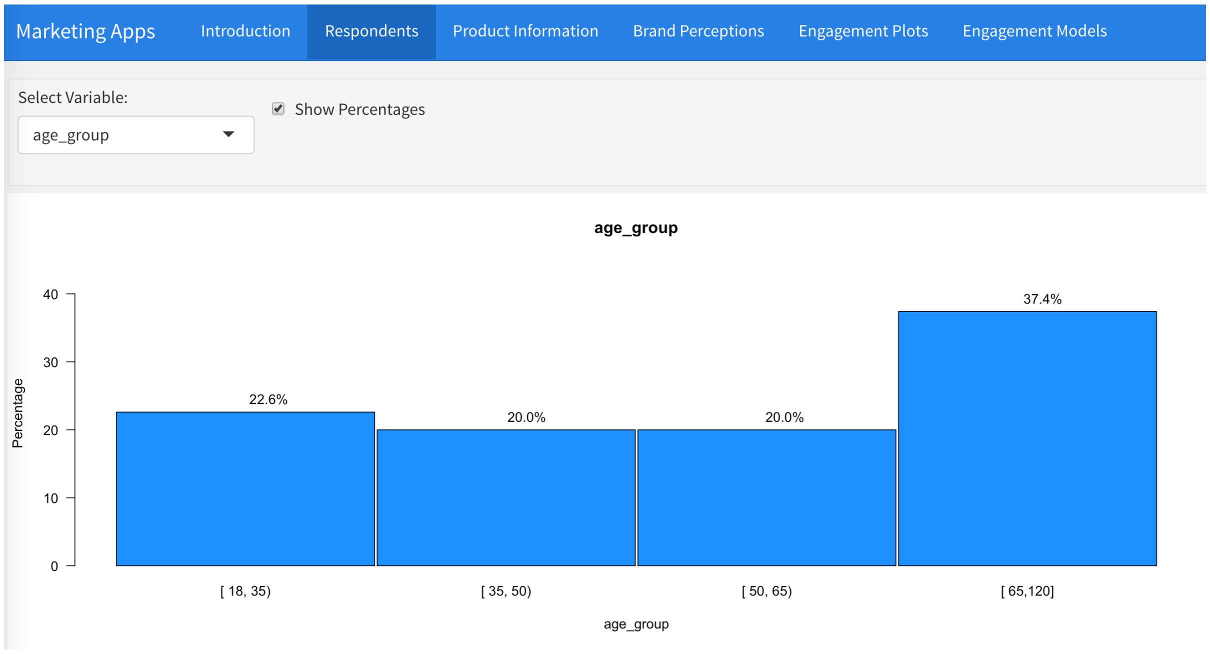
[http://shiny.rstudio.com/images/shiny-cheatsheet.pdf]



Outputs - render*() and *Output() functions work together to add R output to the UI

render*()	works with	*Output()
DT::**renderDataTable(expr**, options, callback, escape, env, quoted**)**	works with	**dataTableOutput(outputId**, icon, …**)**
renderImage(expr, env, quoted, deleteFile**)**		**imageOutput(outputId**, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline**)**
renderPlot(expr, width, height, res, …, env, quoted, func**)**		**plotOutput(outputId**, width, height, click, dblclick, hover, hoverDelay, hoverDelayType, brush, clickId, hoverId, inline**)**
renderPrint(expr, env, quoted, func, width**)**		**verbatimTextOutput(outputId)**
renderTable(expr,…, env, quoted, func**)**		**tableOutput(outputId)**
renderText(expr, env, quoted, func**)**		**textOutput(outputId**, container, inline**)**
renderUI(expr, env, quoted, func**)**		**uiOutput(outputId**, inline, container, …**)** & **htmlOutput(outputId**, inline, container, …**)**

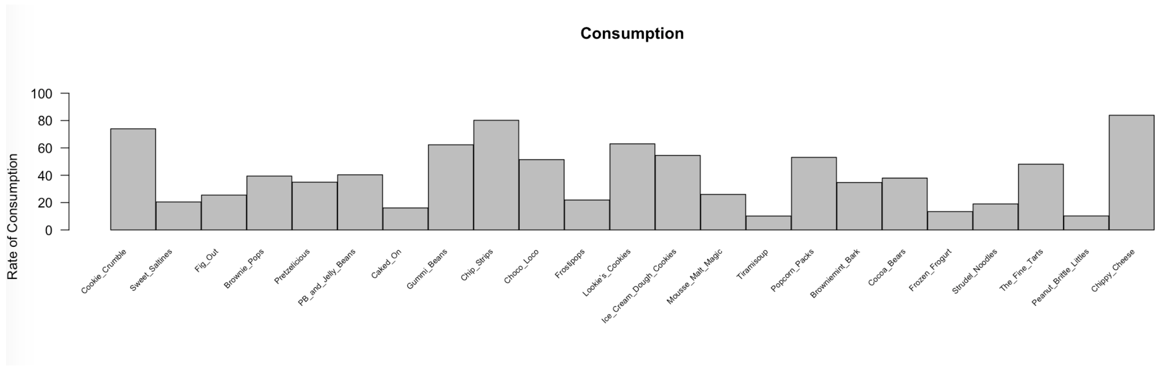
Output from Pieces 1-7



Piece 8: Product Information

- How can we visually compare the products?
- Which products have the highest rates of engagement? The lowest?
- What is the best way to display the results?

A Single Bar Graph



- This graph is a good start.

即为我们想分析的内容：consumption、awareness、etc.

- A dropdown menu to select the state of engagement would help.

Visual Problems with Numerous Graphs

It can be difficult to find one set of parameters that optimizes the display for all of the potential graphs that a reporting engine will generate. It can help to create the capability for the user to customize the visual display to solve these problems:

- The labels may be too small or large.
- You may want to display the values for some graphs and not others.

You may want to focus attention:

- Sorting values
- Only viewing the largest values, etc.

A Plan for an Input Panel

- Dropdown menu for the state of engagement.
- Check boxes to **sort** the values and **display the percentages**.
- Slider values to adjust the size of the labels and which values to plot.

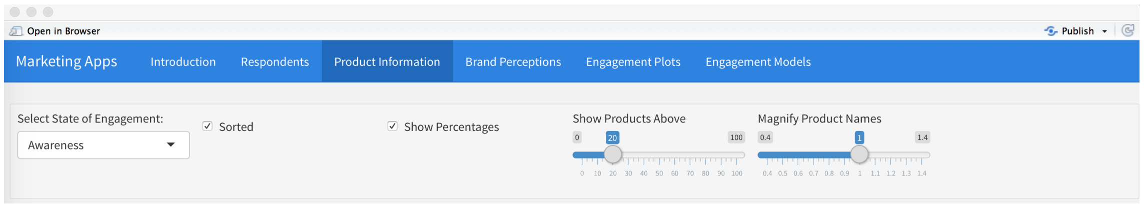
The Product Information Input Panel

```
inputPanel(
  selectInput(inputId = "product_info_engagement_state", label = "Select State of Engagement:", choices = states.of.engagement,
  selected = states.of.engagement[1]),
  checkboxInput(inputId = "product_info_decreasing", label = "Sorted", value=TRUE),
  checkboxInput(inputId = "product_info_show_percentages", label = "Show Percentages", value = TRUE)
,
   sliderInput(inputId = "product_info_min_threshold", label = "Show Products Above", min = 0, max = 100, value = 20, step = 5),
  sliderInput(inputId = "product_info_names_magnification", label = "Magnify Product Names", min = 0.4, max = 1.4, value = 1, step =
  0.1)
)
```

在图中显示的product name的大小

input$product_info_engagement_state tells us which state of engagement to use. （即为选那个产品）
We can create a barplot of the mean engagement by product.
Then the other variables tell us how to customize the plot:
1. input$product_info_decreasing, if TRUE, tells us to sort the results.
2. input$product_info_show_percentages, if TRUE, tells us to label the barplot with the percentages.
3. info$product_info_min_threshold tells us to only display the products with rates above the selected threshold.
4. info$product_info_names_magnification tells us how large to make the products' names under the plot.

Product Information's Menus



Select State of Engagement: Awareness

☑ Sorted

☑ Show Percentages

Show Products Above: 20 (0–100)

Magnify Product Names: 1 (0.4–1.4)

Reactive Content for Product Info

- **input$product_info_engagement_state** tells us which state of engagement to use.

- We can create a barplot of the mean engagement *by* product.

- Then the other variables tell us how to customize the plot:

  1. ***input$product_info_decreasing***, *if TRUE, tells us to sort the results.*
  2. ***input$product_info_show_percentages***, *if TRUE, tells us to label the barplot with the percentages.*
  3. ***info$product_info_min_threshold*** *tells us to only display the products with rates above the selected threshold.*
  4. ***info$product_info_names_magnification*** *tells us how large to make the products' names under the plot.*

Product Information’s Reactive Content

tells us which state of engagement to use

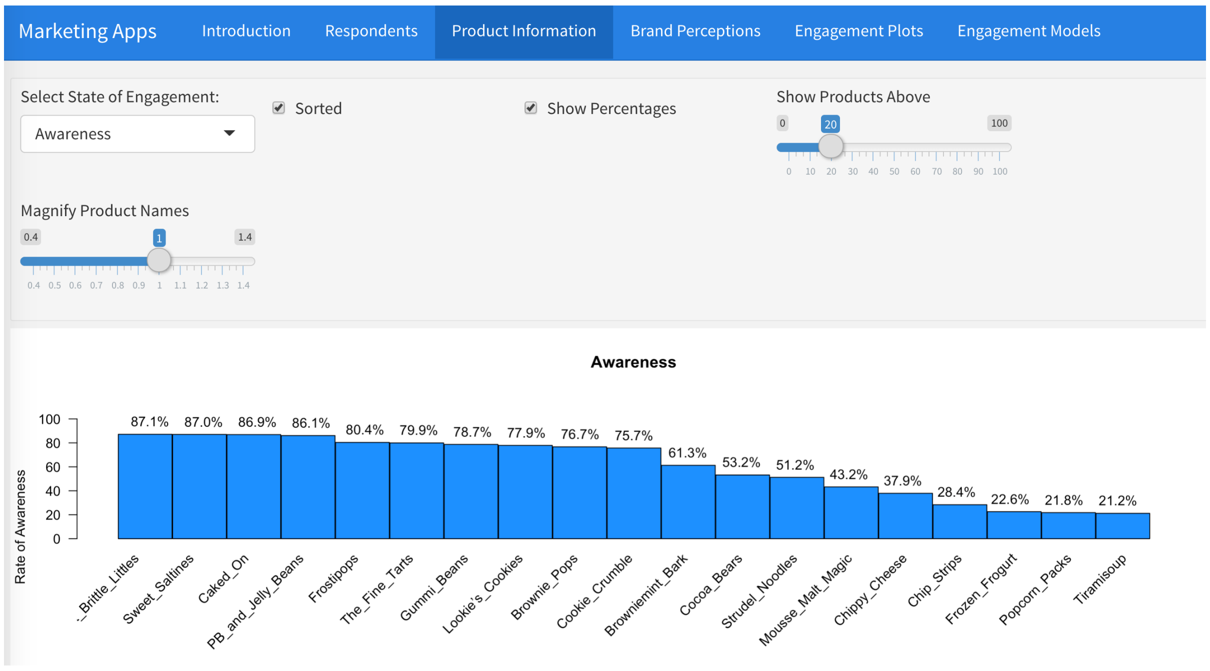
```
renderPlot({
  rates <- dat[, .(Mean = 100*mean(get(input$product_info_engagement_state), na.rm=TRUE)/max(get(input$product_info_engagement_state),
 na.rm = TRUE)), by = product.name]

  if(input$product_info_decreasing == TRUE){
    setorderv(x = rates, cols = "Mean", order = -1)
  }
  barplot(height = rates[Mean > input$product_info_min_threshold,  Mean], names.arg = rates[Mean > input$product_info_min_threshold,
 get(product.name)], space=0.01, las = 1, main = input$product_info_engagement_state, ylab = sprintf("Rate of %s",
 input$product_info_engagement_state), cex.names = input$product_info_names_magnification, ylim = c(-100, 120), xaxt = "n", axes = F,
 col = "dodgerblue")
  axis(side = 2, at = 20*(0:5), las = 2)

  text(x = -0.5 + 1.02*1:rates[Mean > input$product_info_min_threshold, .N], y = -15, labels = rates[Mean >
 input$product_info_min_threshold, get(product.name)], srt = 45, cex = input$product_info_names_magnification, pos = 2)

  if(input$product_info_show_percentages == TRUE){
    space_val = 0
    text(x = -0.4 + 1:rates[Mean > input$product_info_min_threshold, .N] * (1+space_val), y = rates[Mean >
 input$product_info_min_threshold, Mean], labels = sprintf("%.1f%%", rates[Mean > input$product_info_min_threshold, Mean]), pos = 3)

  }
})
```

xaxt: character giving the type of x axis
axes: logical indicating if axes should be drawn.

axis: add a suitable axis to the current plot
at：画刻度线的位置　las:　for vertical/horizontal label orientation

cex.names: expansion factor for axis names (bar labels).

Output for Pieces 1-8



Select State of Engagement: Awareness

☑ Sorted

☑ Show Percentages

Show Products Above: 20

Magnify Product Names: 1

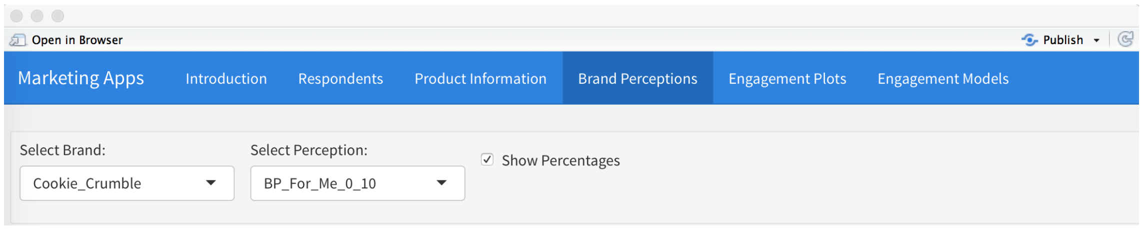
Piece 9: Brand Perceptions

- For each product, there are many questions about the respondents' perceptions of the brand.
- All of the Brand Perceptions are on a 0-10 Scale.
- We would like to display the distribution of answers for each perception of each product.

Brand Perception's Input Panel

```
inputPanel(
  selectInput(inputId="bp_product", label = "Select Brand:", choices = unique.products, selected = unique.products[1]),
  selectInput(inputId="bp_trait", label = "Select Perception:", choices = bp.traits, selected = bp.traits[1]),
  checkboxInput(inputId = "bp_show_percentages", label = "Show Percentages", value = TRUE)
)
```

Brand Perceptions' Menus



Select Brand: Cookie_Crumble

Select Perception: BP_For_Me_0_10

☑ Show Percentages

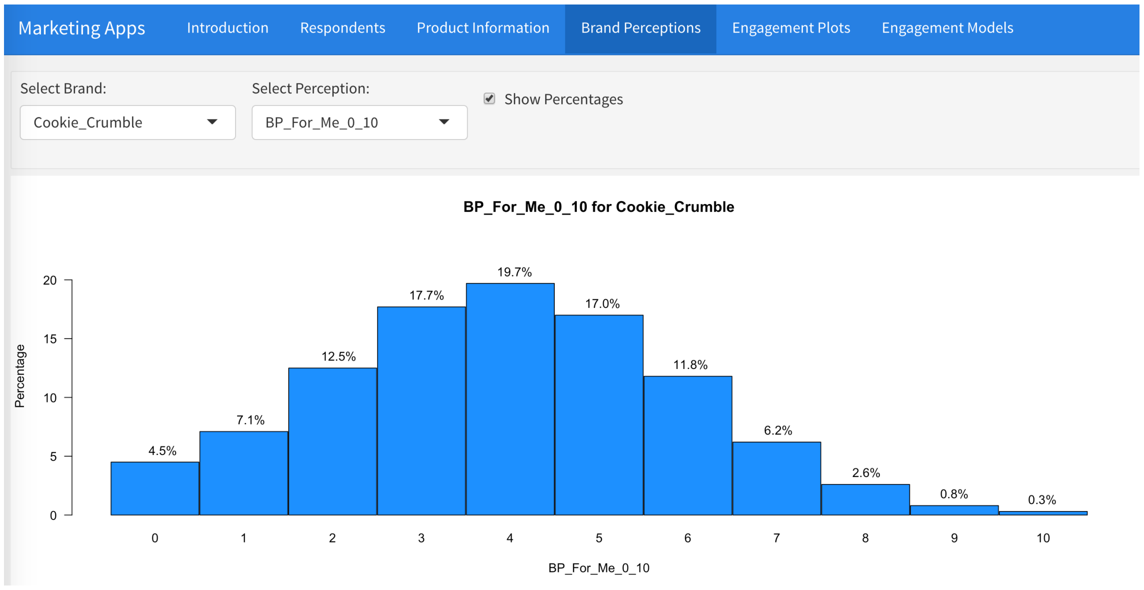
Reactive Content for Brand Perceptions

- The **input$bp_product** will tell us the name of the product.
- The **input$bp_trait** will tell us which Brand Perception variable is selected.
- We'll compute the percentage of respondent who chose each value and display the result in a barplot.
- Then **input$bp_show_percentages**, if TRUE, will prompt us to write in the percentages over the barplot.

Brand Perceptions' Reactive Content

```
renderPlot({
  tab <- percentage.table(x = dat[get(product.name) == input$bp_product, get(input$bp_trait)])
  barplot(height = tab, space=0.01, las = 1, main = sprintf("%s for %s", input$bp_trait, input$bp_product), ylab = "Percentage", xlab
 = input$bp_trait, ylim = c(0, 1.2*max(tab, na.rm=TRUE)), col = "dodgerblue")
  if(input$bp_show_percentages == TRUE){
    space_val = 0
    text(x = -0.4 + 1:length(tab) * (1+space_val), y = tab, labels = sprintf("%.1f%%", tab), pos = 3)
  }
})
```

Output for Pieces 1-9



Select Brand: Cookie_Crumble

Select Perception: BP_For_Me_0_10

[✓] Show Percentages

Piece 10: Engagement Plots

- In previous lectures, we identified a trend: in that data set, the Savvy Samantha subgroup had low awareness but high rates of other states of engagement.

- In the real project, we originally discovered this trend by exploring the results in a reporting engine.

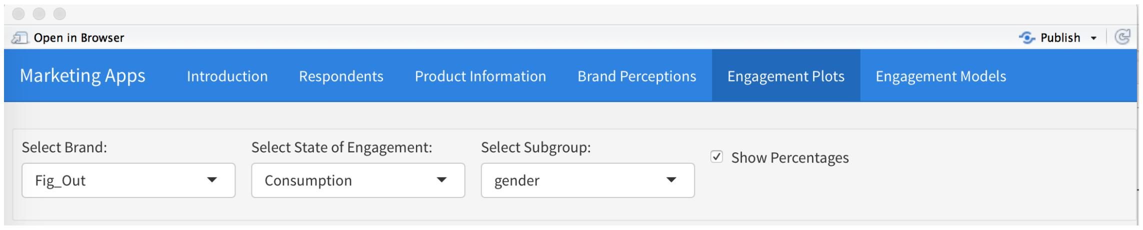
- We want to easily explore the rates of engagement:

    1. *the progression of engagement in each product;*
    2. *the rate of a state of engagement across products;*
    3. *how these rates differ across the subgroups of another variable.*

Engagement Plots’ Input Panel

```
inputPanel(
  selectInput(inputId="ep_product", label = "Select Brand:", choices = unique.products, selected = unique.products[1]),
  selectInput(inputId="ep_state", label = "Select State of Engagement:", choices = states.of.engagement, selected =
 states.of.engagement[1]),
  selectInput(inputId="ep_subgroup", label = "Select Subgroup:", choices = c("All", respondent.variables), selected = "All"),
  checkboxInput(inputId = "ep_show_percentages", label = "Show Percentages", value = TRUE)
)
```

Engagement Plots' Menus



Select Brand: Fig_Out

Select State of Engagement: Consumption

Select Subgroup: gender

✓ Show Percentages

Reactive Content for Engagement Plots

Variables:

- Product: **input$ep_product**
- Engagement State: **input$ep_state**
- Subgrouping Variable: **input$ep_subgroup**
- For the selected brand, state of engagement, and subgrouping variable, compute the **rate of engagement** of each subgroup with the product.
- Then **input$ep_show_percentages**, if TRUE, will prompt us to write in the percentages over the barplot.

Subgrouping or Not

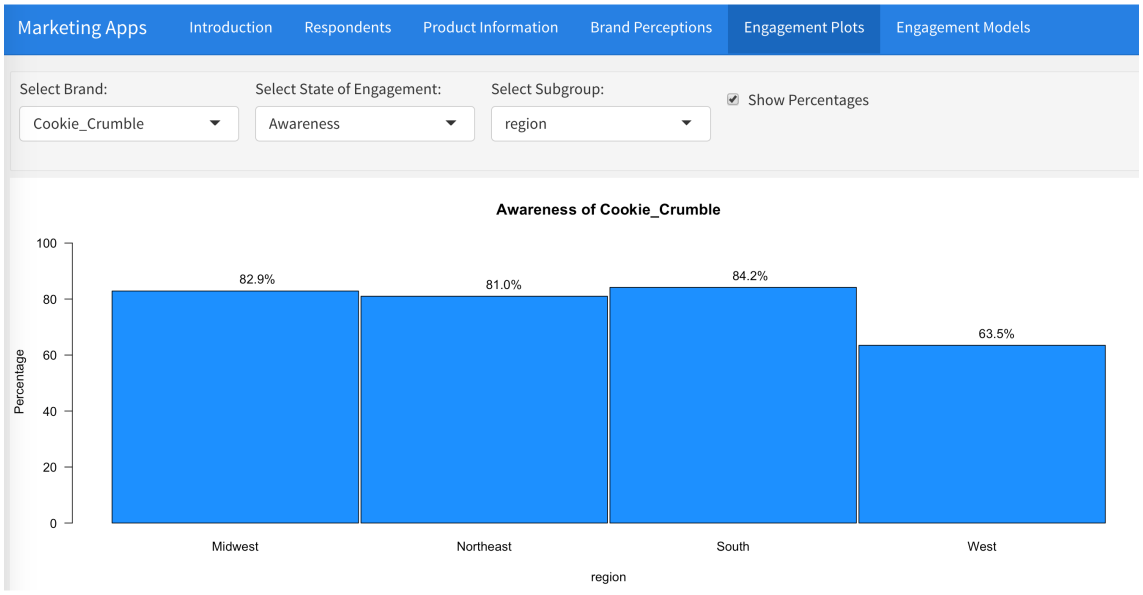
- It would be nice to start with the *overall* rate of engagement.
- Then we can drill down into the subgroups over a number of variables.
- Therefore, we need an if statement:

```
  if(input$ep_subgroup == "All"){
    tab <- dat[Product == input$ep_product, .(Mean = 100*mean(get(input$ep_state),
na.rm=TRUE))]
    tab[, All := "All respondents"]
  }
  else{
    tab <- dat[Product == input$ep_product, .(Mean = 100*mean(get(input$ep_state),
na.rm=TRUE)), keyby = eval(input$ep_subgroup)]
  }
```

Engagement Plots’ Reactive Content

```
renderPlot({
  if(input$ep_subgroup == "All"){
    tab <- dat[get(product.name) == input$ep_product, .(Mean = 100*mean(get(input$ep_state), na.rm=TRUE))]
    tab[, All := "All respondents"]
  }
  else{
    tab <- dat[get(product.name) == input$ep_product, .(Mean = 100*mean(get(input$ep_state), na.rm=TRUE)), keyby =
  eval(input$ep_subgroup)]
  }

  barplot(height = tab[, Mean], names.arg = tab[, get(input$ep_subgroup)], space=0.01, las = 1, main = sprintf("%s of %s",
  input$ep_state, input$ep_product), ylab = "Percentage", xlab = input$ep_subgroup, ylim = c(0, 1.2 * max(tab[, Mean], na.rm = TRUE)),
  col = "dodgerblue")

  if(input$ep_show_percentages == TRUE){
    space_val = 0
    text(x = -0.4 + 1:tab[, .N] * (1+space_val), y = tab[, Mean], labels = sprintf("%.1f%%", tab[, Mean]), pos = 3)
  }
})
```

Output for Engagement Plots



Select Brand: Cookie_Crumble

Select State of Engagement: Awareness

Select Subgroup: region

☑ Show Percentages

Piece 11: Engagement Models Menus

Now we are ready for the most important piece of content. Not coincidentally, it also has the most features.

In the past lecture, we used regression techniques to constructure models for each product's states of engagement. These multivariable models could take all of the respondent-specific and brand-specific variables into account.

However, now we have a **twist**: the client is asking for the flexibility to fit these models **in any combination of products** and **any combination of subgroups of the respondent-specific variables**.

Combinations of Subgroups

- Model the awareness of Fig Out! for females earning more than $100,000 in the Midwest.
- Display a model of Satisfaction that aggregates all of the responses for Tiramisoup, Browniemint Bark, and Mousse Malt Magic together.
- Make any selections of:
    1. *Products*
    2. *Age Group*
    3. *Gender*
    4. *Income Group*
    5. *Region*
    6. *Persona*
    7. *Brand Perception Variables*
- However, at any one time, you must select one single state of engagement.

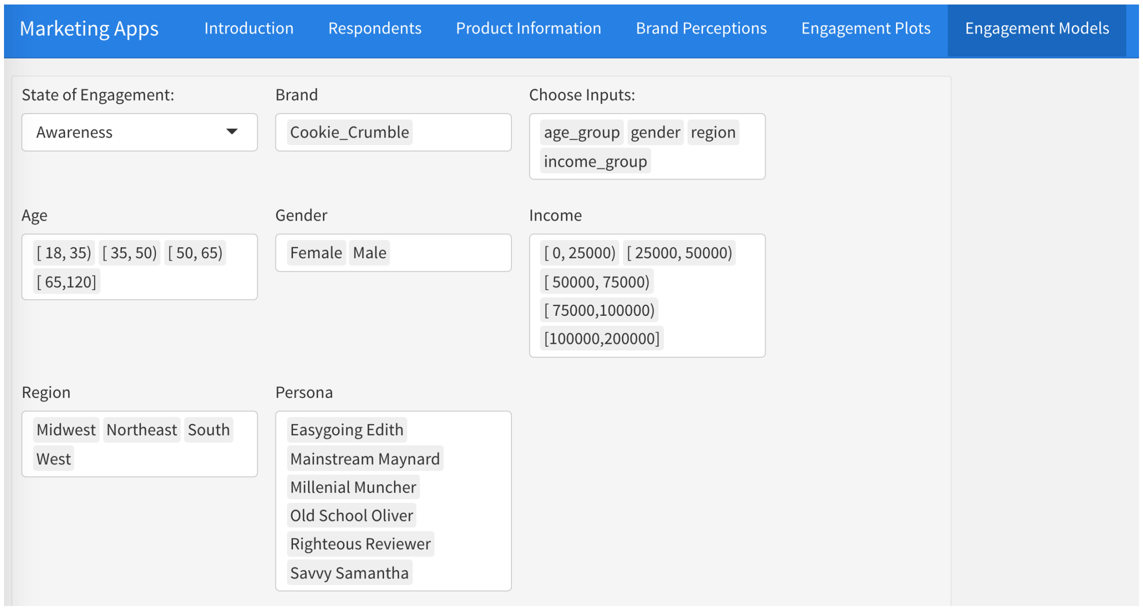
Muliple selections in a Dropdown Menu

- The **selectInput** function has a parameter called **multiple**.
- When **multiple = TRUE**, the user can select any combination of the **choices**.
- Likewise, the value of **selected** can be a character vector including any combination of the **choices*.

Engagement Model’s Input Panel

```
inputPanel(
  selectInput(inputId="em_state", label = "State of Engagement:", choices = states.of.engagement, selected = states.of.engagement[1]),
  selectInput(inputId="em_product", label = "Brand", choices = unique.products, selected = unique.products[1], multiple = TRUE),
  selectInput(inputId="em_inputs", label = "Choose Inputs:", choices = c(age.group.name, gender.name, region.name, income.group.name,
 persona.name, bp.traits), selected = c(age.group.name, gender.name, region.name, income.group.name), multiple = TRUE),
  selectInput(inputId="em_age_group", label = "Age", choices = unique.age.groups, selected = unique.age.groups, multiple = TRUE),
  selectInput(inputId = "em_gender", label = "Gender", choices = unique.genders, selected = unique.genders, multiple = TRUE),
  selectInput(inputId = "em_income_group", label = "Income", choices = unique.income.groups, selected = unique.income.groups, multiple
  = TRUE),
  selectInput(inputId = "em_region", label = "Region", choices = unique.regions, selected = unique.regions, multiple = TRUE),
  selectInput(inputId = "em_persona", label = "Persona", choices = unique.personas, selected = unique.personas, multiple = TRUE)
)
```

Output for Engagement Models Input Panel



State of Engagement: Awareness

Brand: Cookie_Crumble

Choose Inputs: age_group gender region income_group

Age: [18, 35) [35, 50) [50, 65) [65,120]

Gender: Female Male

Income: [0, 25000) [25000, 50000) [50000, 75000) [75000,100000) [100000,200000]

Region: Midwest Northeast South West

Persona: Easygoing Edith Mainstream Maynard Millenial Muncher Old School Oliver Righteous Reviewer Savvy Samantha

Piece 12: Reactive Engagement Models

- We are nearly ready to put together the engagement models.
- The user's selections will determine which subgroups to include.
- However, it's never quite as simple as that…

A Lot of Models

- We can count up the number of possible models:
- For the kth subgrouping variable with n_k possible choices, there are $2^{n_k} - 1$ possible combinations of subgroups for that variable.
- The overall formula for the number of models is:

$$\text{Number of Models} = \text{Number of States of Engagement} * \prod_{k=1}^{K} (2^{n_k} - 1).$$

Counting the Number of Models

```
num_subgroups <- function(x){
  return(2^(length(x)) -1)
}
num_models <- length(states.of.engagement) * num_subgroups(unique.products) * num_subgroups(unique.age.groups) *
  num_subgroups(unique.genders) * num_subgroups(unique.income.groups) * num_subgroups(unique.regions) * num_subgroups(unique.personas)
  *
num_subgroups(bp.traits)


print(sprintf("There are %e possible models.", num_models))
```

```
[1] "There are 1.131837e+17 possible models."
```

That is something like 113.2 quadrillion potential models. Yikes!

113.2 Quadrillion? Really?

- Just because we can fit 113.2 quadrillion models doesn't mean that they all have enough data to support them.
- Many subgroups might have few – or perhaps zero – respondents.
- Moreover, a model can only be estimated when each variable has *variation* within the data.
- A Note: These numbers are a back of the envelope approximation. There are a few subtletites, but even if we're off by a factor of a million, the number of configurations is likely to be vast.

A Broader Problem

- With a unified approach, we'll have one formula for the model of each state of engagement.
- However, if we want the user to **select the variables and the subgroups**, some of these variables will then have a **lack of contrast**. This may be true for structural reasons or by chance when the sample size is small.
- We need to find a way of dynamically altering the model's formula to exclude any variable that lacks contrasts.

Dynamic Formulae

- Fortunately, we already worked out a solution to this problem in Lecture 5.
- The **create.formula** function can be used to combine the user's selected inputs and outputs.
- The **reduce.formula** function can be used to automatically detect and remove the variables that would generate errors in the formulation of the model.

The Alternatives

- Without these functions, the logic of constructing a unique formula for each setting could be very complicated.
- Every model might need a variety of **pre-specified formulae** that would make sense for the setting.
- Even then, issues like **small sample sizes, missing data, and segmentation** might still lead to numerous problems.
- The sheer volume of possible models makes it unlikely that anyone would be able to anticipate the problems that might occur.
- **Dynamic formulae** allow you to build a more adaptible system.

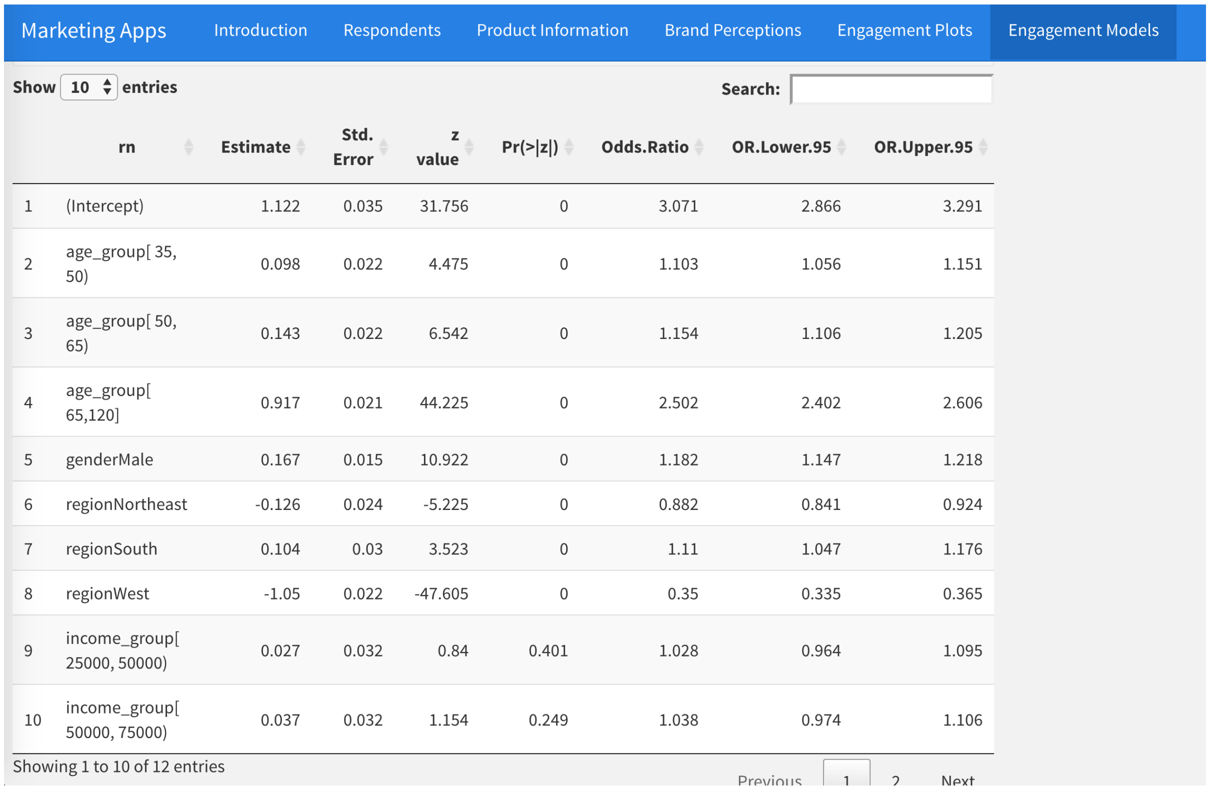
A Plan for Reactive Content

- Use a subset of the data defined by the user's selections.
- Let the user select which variables to include in the model.
- Create a general model for each state of engagement.
- Reduce the model's formula to remove any variable with a lack of contrasts for the data's subset.
- Fit the customized model to the subset of data and report on the results.

Engagement Models' Reactive Content

```
renderDataTable({
  subdat <- dat[get(product.name) %in% input$em_product & get(age.group.name) %in% input$em_age_group & get(gender.name) %in%
  input$em_gender & get(income.group.name) %in% input$em_income_group & get(region.name) %in% input$em_region & get(persona.name) %in%
  input$em_persona]

  if(input$em_state == satisfaction.name){
    model.type <- "linear"
  }
  if(input$em_state != satisfaction.name){
    model.type <- "logistic"
  }
          ??? what is the function
  res <- fit.model(dt = subdat, outcome.name = input$em_state, input.names = input$em_inputs, model.type = model.type)

  datatable(data = res)
})
```

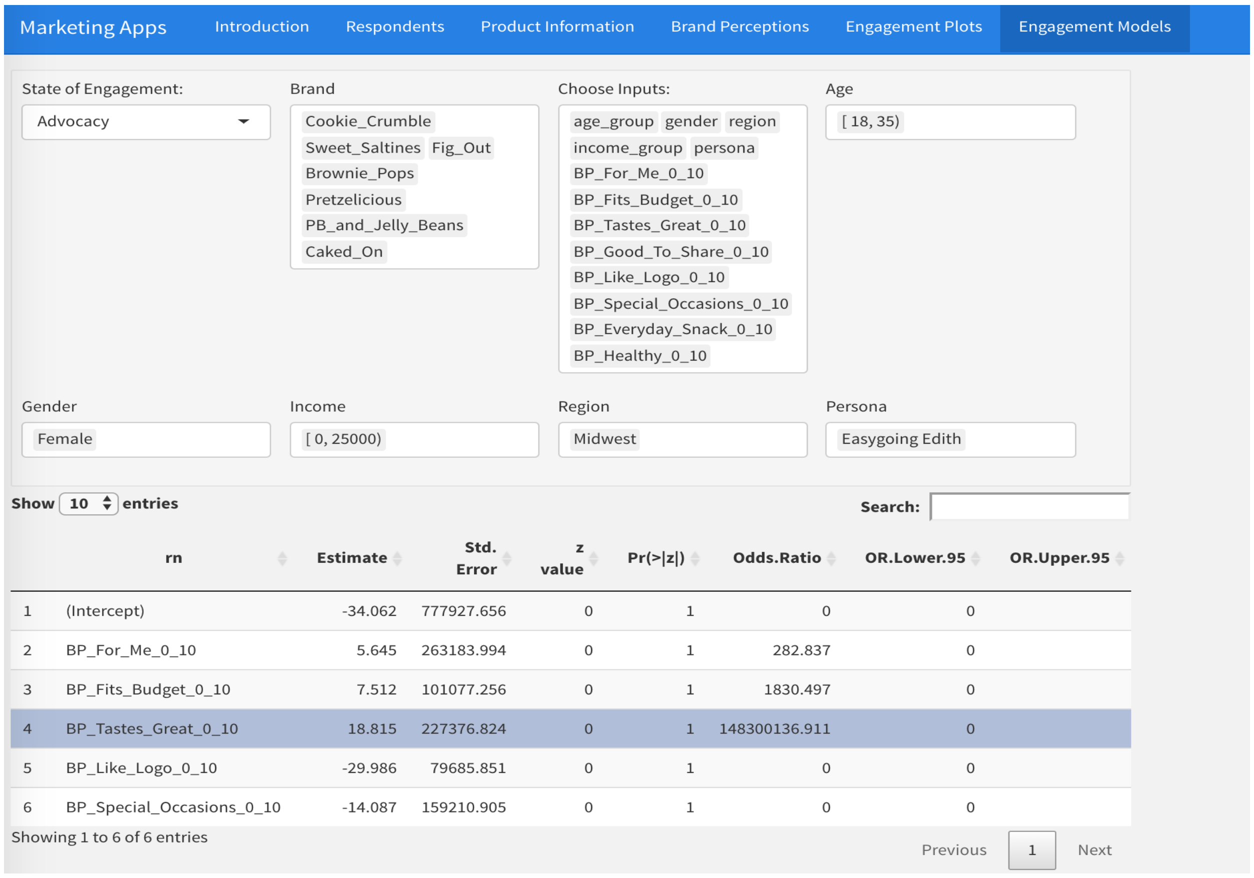
Pieces 1-12's Output



	rn	Estimate	Std. Error	z value	Pr(>|z|)	Odds.Ratio	OR.Lower.95	OR.Upper.95
1	(Intercept)	1.122	0.035	31.756	0	3.071	2.866	3.291
2	age_group[35, 50)	0.098	0.022	4.475	0	1.103	1.056	1.151
3	age_group[50, 65)	0.143	0.022	6.542	0	1.154	1.106	1.205
4	age_group[65,120]	0.917	0.021	44.225	0	2.502	2.402	2.606
5	genderMale	0.167	0.015	10.922	0	1.182	1.147	1.218
6	regionNortheast	-0.126	0.024	-5.225	0	0.882	0.841	0.924
7	regionSouth	0.104	0.03	3.523	0	1.11	1.047	1.176
8	regionWest	-1.05	0.022	-47.605	0	0.35	0.335	0.365
9	income_group[25000, 50000)	0.027	0.032	0.84	0.401	1.028	0.964	1.095
10	income_group[50000, 75000)	0.037	0.032	1.154	0.249	1.038	0.974	1.106

Showing 1 to 10 of 12 entries

What About Those 113.2 Quadrillion Models?



State of Engagement: Advocacy

Brand: Cookie_Crumble, Sweet_Saltines, Fig_Out, Brownie_Pops, Pretzelicious, PB_and_Jelly_Beans, Caked_On

Choose Inputs: age_group, gender, region, income_group, persona, BP_For_Me_0_10, BP_Fits_Budget_0_10, BP_Tastes_Great_0_10, BP_Good_To_Share_0_10, BP_Like_Logo_0_10, BP_Special_Occasions_0_10, BP_Everyday_Snack_0_10, BP_Healthy_0_10

Age: [18, 35)

Gender: Female

Income: [0, 25000)

Region: Midwest

Persona: Easygoing Edith

Show 10 entries

Search:

	rn	Estimate	Std. Error	z value	Pr(>|z|)	Odds.Ratio	OR.Lower.95	OR.Upper.95
1	(Intercept)	-34.062	777927.656	0	1	0	0	
2	BP_For_Me_0_10	5.645	263183.994	0	1	282.837	0	
3	BP_Fits_Budget_0_10	7.512	101077.256	0	1	1830.497	0	
4	BP_Tastes_Great_0_10	18.815	227376.824	0	1	148300136.911	0	
5	BP_Like_Logo_0_10	-29.986	79685.851	0	1	0	0	
6	BP_Special_Occasions_0_10	-14.087	159210.905	0	1	0	0	

Showing 1 to 6 of 6 entries

Previous 1 Next

Coaching Your Users

- Narrow subgroups with small sample sizes will eventually create ridiculous output in multivariable models.
- Odds ratios of 100 – to say nothing of a trillion – are not remotely believable.
- If your marketing team wants 113.2 Quadrillion Models, we can provide them. However, the team will likely require some coaching about which models are reasonable and when the results should be taken with a grain of salt.
- More sophisticated applications might provide more warnings in the output – or refuse to fit a model at all if the sample size is not sufficient.

Lo and Behold, a Reporting Engine

- Now we have the tools to build web interfaces in R.
- You can generate a wide range of dynamic content.
- Reporting engines are powerful tools that can provide an unprecedented degree of information to those who need it.

Simple Extensions of Reporting Engines

- **Checkboxes** to write graphs or tables to files.
- **Monitoring applications** to track business results over time (weekly, monthly, yearly, etc.)
- **Quality Investigations** to identify groups or individual records that are worthy of further investigation.

Training Your Team

- Nearly everyone can open RStudio and knit a file.
- With some simple instructions, your team members can independently explore your results.
- You may no longer need to answer the typical ad-hoc questions that can take a few hours of your time.

Getting Feedback

- Your reporting engine likely displays more content than you can reasonably check.
- There will most likely be errors, special cases, or data quality issues that you could not have foreseen.
- Over time, the application will become more robust. Take each piece of feedback as an opportunity to deliver a higher quality product.

Tremendous Skills

- Building shiny applications can greatly expand upon your ability of your team to utilize information.
- Relative to simple analyses or even a normal RMarkdown report, these applications give you an unprecedented ability to examine your data in great detail.
- Building such an application can be a great demonstration of your skills, which might lead to more and better opportunities.