Empirical Reasoning Center
R Workshop (Summer 2017)
Session 2

This guide reviews the examples we will cover in today's workshop. It should be a helpful introduction to R, but for more details, you can find additional workshop materials at the ERC website.

1 Manipulating data

Start a new script...

```
## clear the workspace
rm(list=ls(all=TRUE))

## load packages
library(foreign)
library(dplyr)
library(car)
library(ggplot2)
library(DataCombine)
```

```
## set working directory
setwd("/your/file/path")   ## add your file path
```

Load a dataset— this one includes municipal finance data from the US Census of Governments.

```
## load dataset
load("muni_25K_finance_data.RData")
```

```
## some basic info about data
dim(COG.muni.25K)

head(COG.muni.25K)

names(COG.muni.25K)
```

```
## TIP: one way to change the name of a data frame (optional)
COG <- COG.muni.25K

## TIP: one way to remove the original copy (optional)
# COG.muni.25K <- NULL
```

1.1 Subsetting data

There are many different ways to keep or drop certain variables or rows. Below you will find several examples. You can save subsets of your data to separate data frames, or many of these functions can be used within other functions to perform operations on a subset without saving it to an object.

- You can keep or drop variables (columns) using the `select()` function in the `dplyr` package. Just specify the data frame and variables to keep or drop (add a minus sign in front of variables you want to drop)

```
## keep only specified columns
COG.subset <- COG[, c(1:4, 6:8, 9, 10, 25:554)]

## drop variables using select() from the dplyr package
## note leave out the - to select variables to keep
COG.subset <- select(COG.subset, -SortCode, -SurveyYr, -County, -Type.Code)
```

- You can also specify a vector of variables that you want to keep or drop and use the vector to extract the variables you want to keep

```
## an alternative for selecting variables to keep or drop
COGvars <- names(COG) %in%  c("Year4", "ID", "State.Code", "Name",
             "Census.Region", "Population", "Total.Revenue", "Total.Taxes",
             "Property.Tax", "Total.Gen.Sales.Tax", "Total.General.Charges",
             "Total.Expenditure", "Total.Debt.Outstanding",
             "Total.Long.Term.Debt.Out", "ST.Debt.End.of.Year")

## keep the variables in COGvars
COG.subset <- COG[COGvars]

## keep the variables not in COGvars
COG.subset <- COG[!COGvars]

## keep the variables in COGvars
## these are the data we want- note we can easily overwrite the data frame
COG.subset <- COG[COGvars]
```

- The `subset()` function allows you to keep observations (rows) that meet one or more condition(s)

```
## subset based on one or more condition(s)

## use the subset() function
COG.50K <- subset(COG.subset, Population >= 50000)

COG.50K.NE <- subset(COG.subset, Population >= 50000 & Census.Region == 1)

COG.large <- subset(COG.subset, Population > median(Population))
```

- The `filter()` function in the `dplyr` package also allows you to select rows based on one or more condition(s)

```
## or use the filter() function from the dplyr package
COG.25K <- filter(COG.subset, Population < 50000)

COG.small <- filter(COG.subset, Population < median(Population))

COG.25K.WMW <- filter(COG.subset, Population < 50000 &
                          (Census.Region == 2 | Census.Region == 4))
```

- You can nest the `filter()` and `select()` functions to drop or keep rows and columns at the same time

```
## nest dplyr functions to select rows & columns at the same time
cities.100K <- select(filter(COG, Population <= 100000),
                      Year4, ID, State.Code, Name, Census.Region, Population)
```

1.2 Appending data

You can append, or add, rows of data using the `rbind()` function.

```
## append rows of data

GOG.subset2 <- rbind(COG.50K, COG.25K)
```

1.3 Merging data

There are multiple packages that offer functions to merge data, and a couple of options are included here. Recall that you can have more than one data frame open at any given time. As a result, you can have multiple datasets in memory, and you can make changes to these datasets prior to (or after) you merge them.

In this example, we will merge a crosswalk that combines the GOVS IDs used in the municipal finance data and the FIPS codes that are used to identify municipalities in most datasets.

```
## read in a new dataset
fips <- read.dta("GOVS to FIPS Crosswalk.dta")
```

Prepare the data for merge. You can drop variables from either data frame, and you can rename variables in either data frame as necessary.

```
## you can select variables to keep or drop (optional)
fips <- select(fips, -areaname)
```

```
## need to rename some variables to merge on

## one option is rename() in the dplyr package
## there are other options--- e.g., rename() in the reshape package or names()
## note that to rename variables with the names() option,
## you must re-enter names for all variables in order
COG.subset <- rename(COG.subset, govsid = ID, govsstate = State.Code)
fips <- rename(fips, Name = name)
```

The `dplyr` package has several merge options— `left_join()` is probably most commonly used (`left_join(x, y)` merges 2 data frames x and y, returning all rows from x and all columns from x and y). The `dplyr` package supports the following merge functions: `inner_join()`, `left_join()`, `right_join()`, `semi_join()`, `anti_join()`, and `full_join()`. If you do not specify which variables to use for the merge, the `left_join()` function will determine which variable(s) are common to both data frames for use in the merge.

```
COG.fips <- left_join(COG.subset, fips)

## Joining, by = c("govsid", "govsstate", "Name")
## Warning in left_join_impl(x, y, by$x, by$y, suffix$x, suffix$y):  joining character
vector and factor, coercing into character vector

## note that we're left with 427 observations in COG.subset without matches in fips
length(COG.fips$fipsplace[is.na(COG.fips$fipsplace)])

## [1] 427
```

You can also specify which variable(s) to merge on. In the example above, `left_join()` automatically merged on the variables govsid, govsstate, and Name. However, it looks like there were some issues matching the string variable Name, leaving an incomplete merge. In this case, specifying merge variables and excluding the string variable Name, leads to a better outcome.

```
## you can (but do not have) to specify variables to use for merge
COG.fips <- left_join(COG.subset, fips, by=c("govsid", "govsstate"))

length(COG.fips$fipsplace[is.na(COG.fips$fipsplace)])

## [1] 25
```

Another option is the `merge()` function– be sure to specify if you want to keep all of the observations in one or both of your data frames.

```
COG.fips <- merge(COG.subset, fips, by=c("govsid", "govsstate"), all.x = TRUE)
```

1.4 Cleaning and coding data

There are many functions for cleaning and coding data with R. This section presents a few useful functions.

Formatting variables
A few examples:

- The `sprintf()` function can format variables. In this example, an existing variable is formatted to be numeric with 5 digits before the decimal point and 0 digits after (the 0 before the 5 indicates that where necessary, leading 0s should be added).

```
COG.fips$fipsplace <- sprintf("%05.0f", as.numeric(COG.fips$fipsplace))
```

- The `paste()` and `paste0()` functions concatenate strings. With `paste()`, you will need to designate a separator to join the strings, and `paste0()` leaves no spaces. In this example, both variables are numeric, but `paste0()` will automatically turn them into strings.

```
## combining 2 variables
COG.fips$fipsid <- paste0(COG.fips$fipsstate, COG.fips$fipsplace)
```

- To change the variable to a numeric variable, simply use the `as.numeric()` function. Note that when you change a string variable to a numeric variable, non-numeric values will be missing. There are similar functions for other types of variables: `as.integer()`, `as.character()`, and `as.factor()`. However, changing a factor variable to a numeric variable requires an extra step (see note on factor variables).

```
COG.fips$fipsid <- as.numeric(COG.fips$fipsid)

## Warning:  NAs introduced by coercion
```

Note: Factor Variables are categorical variables that can be either string or numeric and ordered or unordered. Factors can be useful for designating categories or groups of interest in both statistical analyses and graphics. Sometimes, when you create a data frame or read in data, a numeric or integer variable may stored as a factor variable. Even if you see a number when you examine a factor variable, R recognizes factors as categorical– i.e., not numeric values. You can change factor variables to numeric values, but an extra step is required to ensure you get the results you expect. You must start by transforming a factor into a character variable using the `as.character()` function, and then you can turn the character (string) variable into a numeric variable with `as.numeric()`. You can code this process in two steps, or you can nest the functions (e.g., `numeric_variable <- as.numeric(as.character(factor_variable))`)

Coding and/or recoding variables
This example creates a variable and assigns values based on certain conditions. Here, we extract values from the `summary()` function to test conditions. You can also use numbers or other variables. The `recode()` function in the `car` package also provides a helpful option for recoding existing variables. However, note that both the `dplyr` and the `car` packages include `recode()` functions which operate differently and can create a conflict.

Shortcut: Note the use of the `within()` function in the code below. This function is helpful when you need to make multiple changes in one data frame because it allows you to refer to the data frame only once rather than linking every variable to the data frame.

```
COG.fips <- within(COG.fips, {

    ## create and code a variable for population categories
    Population.Category <- NULL
    Population.Category[Population < summary(COG.fips$Population)[2]] <- "Small"
    Population.Category[Population > summary(COG.fips$Population)[2] &
                        Population < summary(COG.fips$Population)[5]] <- "Medium"
    Population.Category[Population > summary(COG.fips$Population)[5]] <- "Large"

    ## recode to replace region number with region name
    ## census regions: 1 = Northeast / 2 = Midwest / 3 = South / 4 = West
    Census.Region <- recode(Census.Region, recodes = "1='Northeast';
                            2='Midwest'; 3='South'; 4='West'")
})
```

```
COG.fips <- within(COG.fips, {
    Total.Revenue.PC <- Total.Revenue/Population
    Total.Taxes.PC <- Total.Taxes/Population
    Total.Expenditure.PC <- Total.Expenditure/Population

    log.Total.Revenue.PC <- log(Total.Revenue.PC)
    log.Total.Taxes.PC <- log(Total.Taxes.PC)
    log.Total.Expenditure <- log(Total.Expenditure.PC)
})
```

Lag & lead variables

Multiple R packages include functions for generating lag and lead variables. You will find two approaches here, but you may find other options online or via R Help.

First is the `slide()` function from the `DataCombine` package. The syntax and arguments for the `slide()` function are quite straightforward. Specify the data frame, the variable to lag or lead, the time variable, and a group variable (if applicable). Then, provide a name for the new lag or lead variable and specify the number of units to lag or lead. Note, however, that `slide()` will lag or lead to the next observation of the TimeVar even if there is a gap and even if that gap only occurs for some observations. This example demonstrates just such a result. You will also see how to sort a data frame by one or more variable(s).

```
# first, sort by city and year
COG.fips <- COG.fips[order(COG.fips$fipsid, COG.fips$Year4),]

# one easy option is the slide function in the DataCombine package, which
# also creates lead variables
#
# note that slide() will lag or lead to the next observation of the
# TimeVar even if there is a gap

COG.fips <- slide(COG.fips, Var = "Total.Revenue.PC",
```

```
                  TimeVar = "Year4",
                  GroupVar = "fipsid",
                  NewVar = "lag.Total.Revenue.PC",
                  slideBy = -1)

##
## Lagging Total.Revenue.PC by 1 time units.
```

A loop is a less efficient but effective option for creating lag or lead variables. This approach requires more lines of code and may take more processing time, but you can include detailed specifications. When using a loop to create lag or lead variables, be sure to sort your data frame first. Note that the code also defines empty vectors (in this case, variables) to hold the output from the loop.

```
# first, sort by city and year
COG.fips <- COG.fips[order(COG.fips$fipsid, COG.fips$Year4),]

COG.fips <- within(filter(COG.fips, !is.na(fipsid)), {
    ## make the new variables with null values
    lag.Total.Taxes.PC <- NULL
    lag.Total.Expenditure.PC <- NULL


    ## loop over rows to fill in lagged values
    for(i in 2:length(Year4)) {
            if(fipsid[i] == fipsid[i-1] & Year4[i] == Year4[i-1] + 1){
                lag.Total.Taxes.PC[i] <- Total.Taxes.PC[i-1]
                lag.Total.Expenditure.PC[i] <- Total.Expenditure.PC[i-1]
            }}
})
```

Because there are missing years for some cities in this example, the `slide()` function created a few incorrect lagged values. In this case, the `ifelse()` function is one way to fix the problem. The `ifelse()` function takes 3 arguments, first is the "if" condition, second is the value of the variable if the condition is true, and third the value if the condition is false. You can nest `ifelse()` functions.

```
COG.fips$lag.Total.Revenue.PC <- ifelse(is.na(COG.fips$lag.Total.Taxes.PC) &
                                        is.na(COG.fips$lag.Total.Expenditure.PC),
                                        NA, ## if true
                                        COG.fips$lag.Total.Revenue.PC)  ## else
```

When you finish cleaning your data, you can save it in a .RData file. You can also write another type of file if you prefer.

```
save(COG.fips, file = "muni_finance_data_cleaned.RData")
```

2 Creating plots & figures

Figures and plots can help you present your data and results visually. The `base` package of R includes several data visualization options, including the `plot()` and `hist()` functions. However, the `ggplot2` package is popular with R users across disciplines because of its extensive options and high quality output. In this section, you will find a variety of `ggplot2` graphs and figures.

The `ggplot2` package supports both the `qplot()` and `ggplot()` functions. You can obtain similar output from either function, but `ggplot()` does include a wider variety of formatting choices. The syntax is different, so you will find sample code for both below. You can assign output from `qplot()` and `ggplot()` to objects, which allows you to recall plots without re-running the code, and you also can use these objects to export and save plots. You can also make changes to an existing saved plot.

Before making plots, you can define factor variables for groups or categories that you might want to incorporate into your figures. This step is optional, and you may already have such variables in your dataset. The examples below uses municipality-level data, so you might want to differentiate by region and by city size.

```
## create factors-- factors designate groups or categories
## (this is optional, depending on the figures you need)
COG.fips$Population.Category <- factor(COG.fips$Population.Category,
                                       levels = c("Small", "Medium", "Large"))

COG.fips$Census.Region <- factor(COG.fips$Census.Region,
                          levels = c("Northeast", "Midwest", "South", "West"))
```

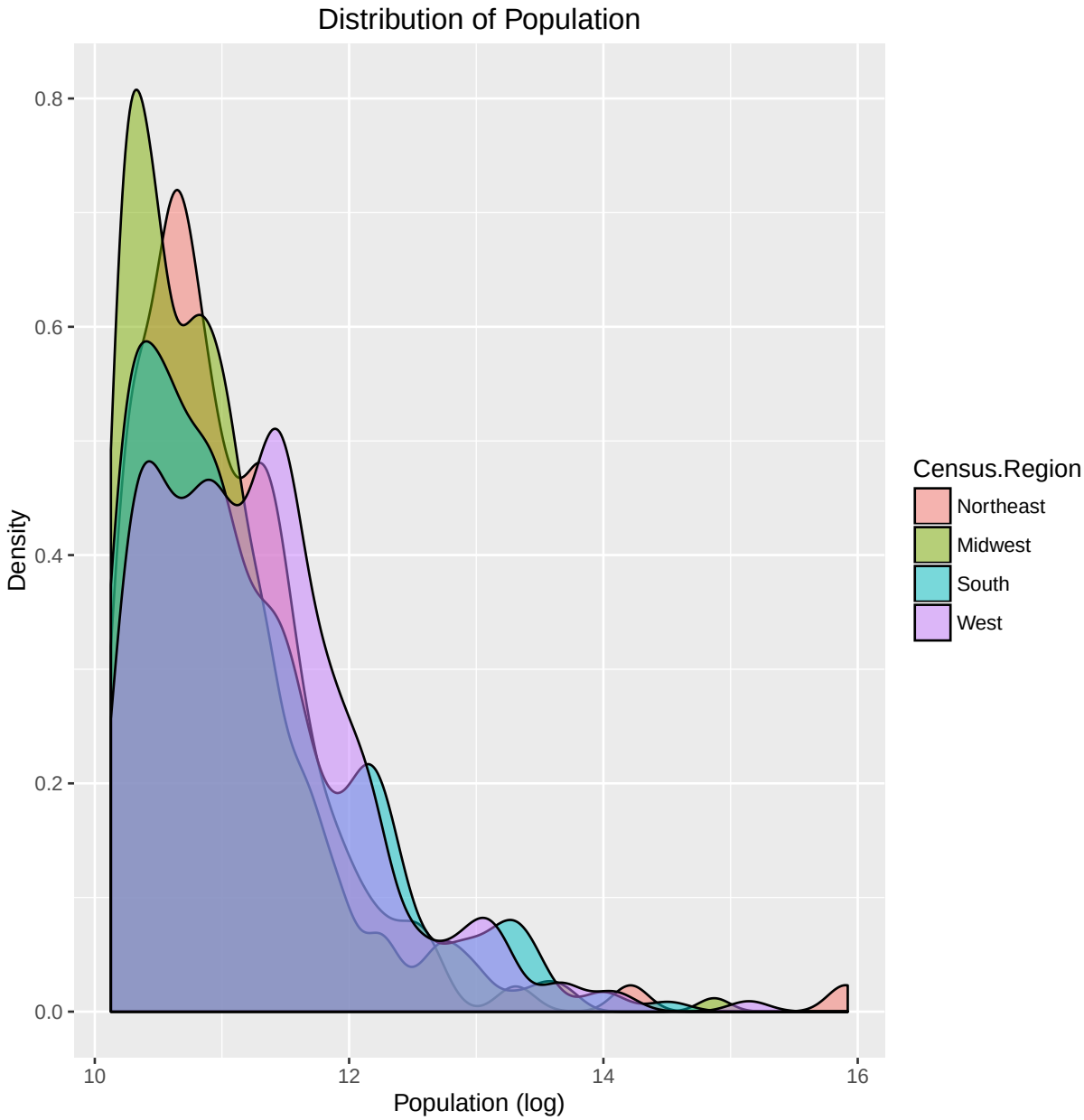
2.1 Using `qplot()`

You can make a variety of figures with the `qplot()` function. Key arguments are the variable(s) you want to plot, type of plot to produce (`geom=`), and the data to use (`data=`). You can add additional features based on your needs and preferences.

Kernel density plot
The code below uses `qplot()` to create a kernel density plot. This example plots the density of the log of the Population variable by Census.Region. Note that you can nest other functions within `qplot`. Using the `fill=` argument with a factor variable produces plots of different colors for the groups. The `alpha=` argument specifies the transparency of the fill (higher values are more opaque). If you leave out the `fill=` argument, you would generate a plot of the density for the variable for your entire dataset.

```
library(ggplot2)

# grouped by region (indicated by color)
qplot(log(Population), data=COG.fips, geom="density",
      fill=Census.Region,
      alpha=I(.5),
```

```
      main="Distribution of Population",
      xlab="Population (log)",
      ylab="Density")
```

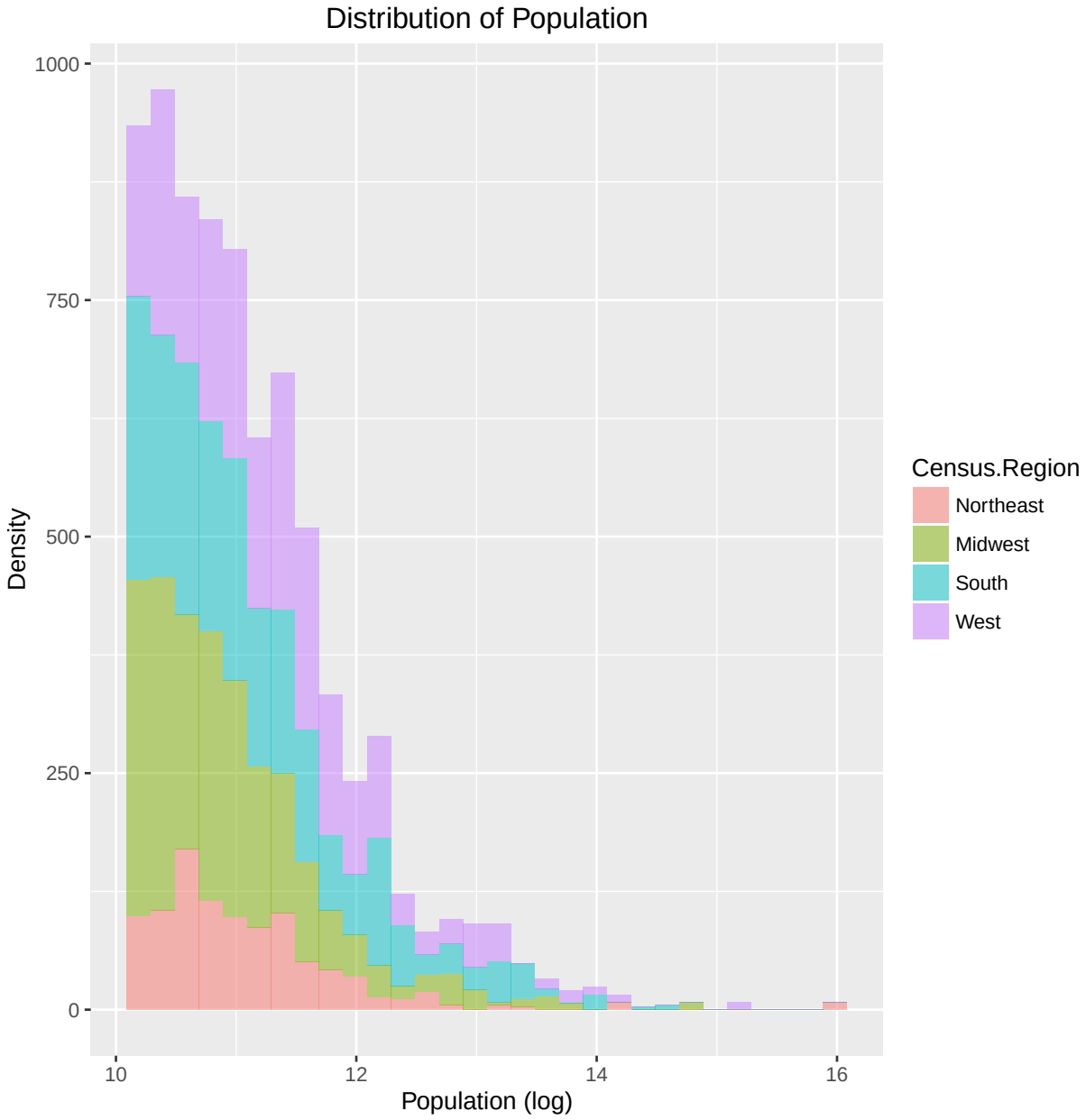


Histogram

The example below plots the distribution of log Population by Census.Region. Note that the code and arguments are similar, but here `geom="histogram"`. Again, if you leave out the `fill=` argument, you would produce a histogram for all (non-missing) observations of the Population variable.

```
# grouped by region (indicated by color)
qplot(log(Population), data=COG.fips, geom="histogram",
      fill=Census.Region,
      alpha=I(.5),
```

```
      main="Distribution of Population",
      xlab="Population (log)",
      ylab="Density")

## `stat_bin()` using `bins = 30`.  Pick better value with `binwidth`.
```

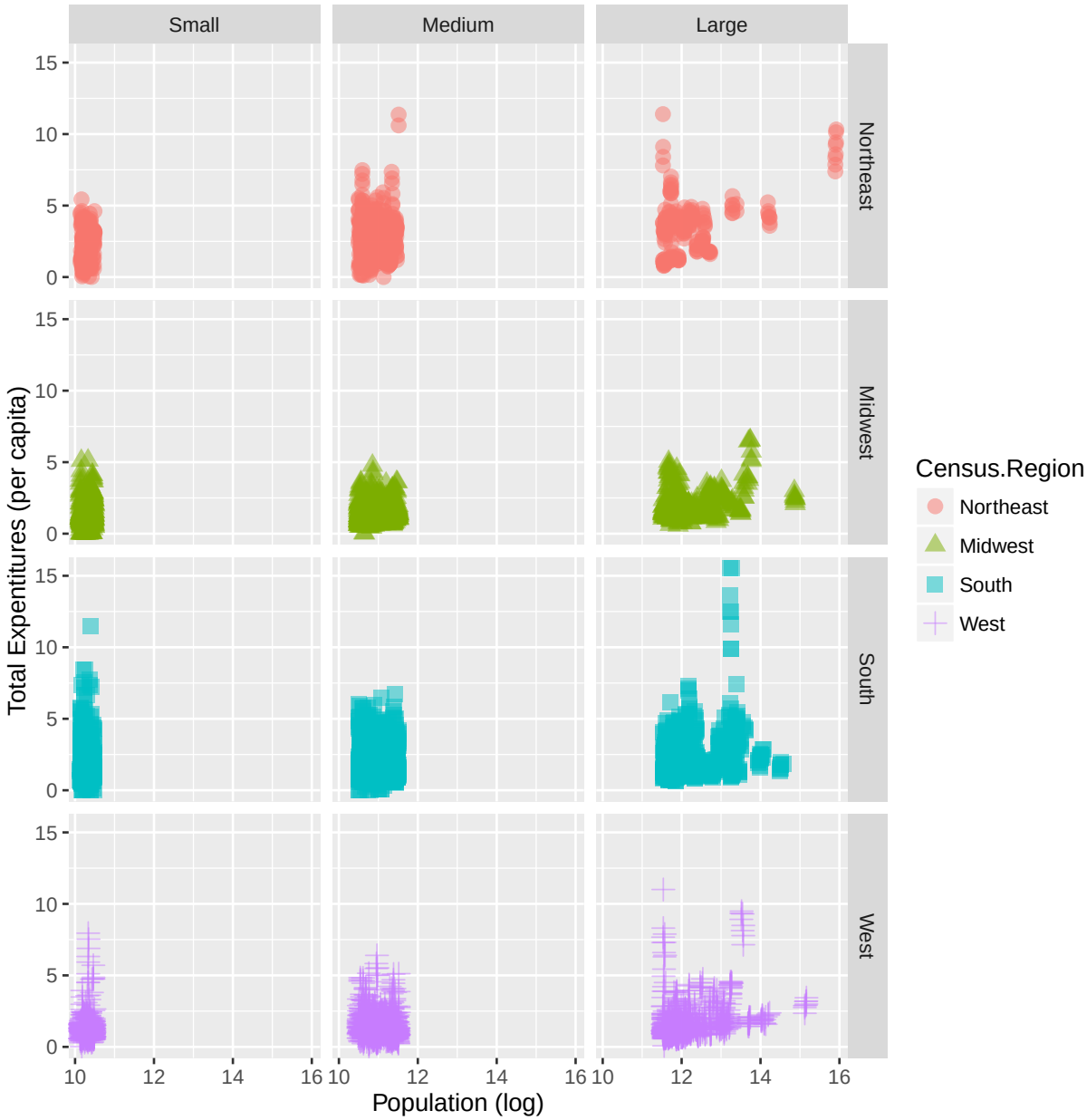


Scatter plots

To produce scatter plots, you can use the argument `geom="point"`. Note that this argument is not included in the example below. The `geom="point"` is a default for the `qplot` function, so if you do not specify the type of plot, `qplot` will produce a scatter plot (with two variables; or a histogram with only one variable). Here, the shape and the color of the points vary with Census.Region.

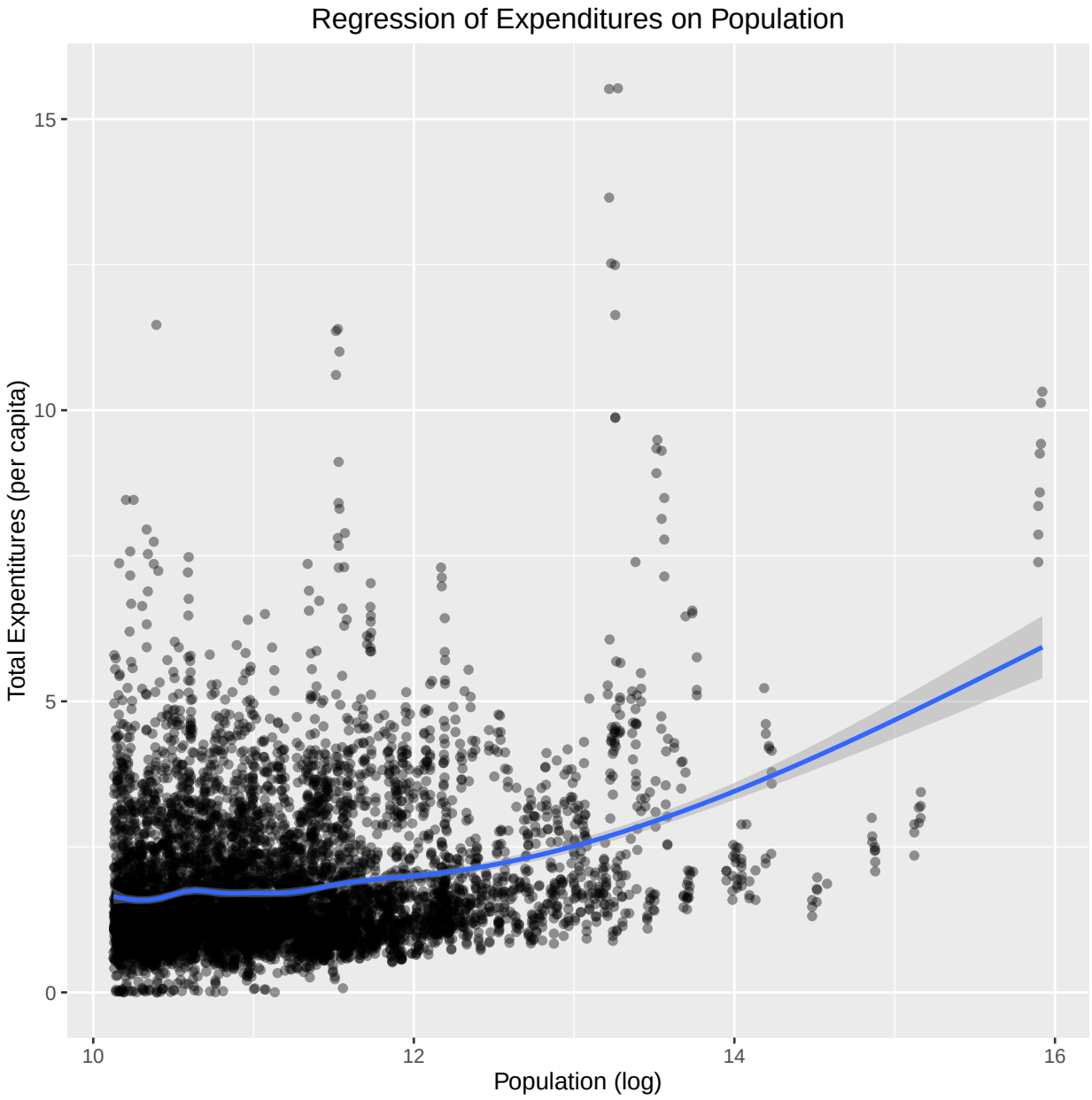
This example also includes the `facet=` argument. In the context of `ggplot2`, facets allow you to produce one figure that includes multiple plots displaying subsets of your data. The `facet=` argument specifies a factor variable to use to create rows and/or a variable to create columns. The variables are separated by a tilde (`row_variable ~ column_variable`). To produce only rows or only columns, replace the other variable with a period (.). Note that the row variable always goes before the tilde and the column variable always goes after even when you use only rows or only columns.

```
# in each facet, region is represented by shape and color
qplot(log(Population), Total.Expenditure.PC, data=COG.fips,
      shape=Census.Region,
      color=Census.Region,
      facets=Census.Region~Population.Category,
      alpha=I(.5),
      size=I(3),
      xlab="Population (log)",
      ylab="Total Expentitures (per capita)")
```



You can also add a trend line to your scatter plot. You can include the argument `geom="smooth"`, and `qplot` will choose a method to produce a trend line based on the data. The example below includes a loess line.

```
# add a trend line
qplot(log(Population), Total.Expenditure.PC, data=COG.fips,
      geom=c("point", "smooth"),
      alpha=I(.4),
      main="Regression of Expenditures on Population",
      xlab="Population (log)", ylab="Total Expentitures (per capita)")
```

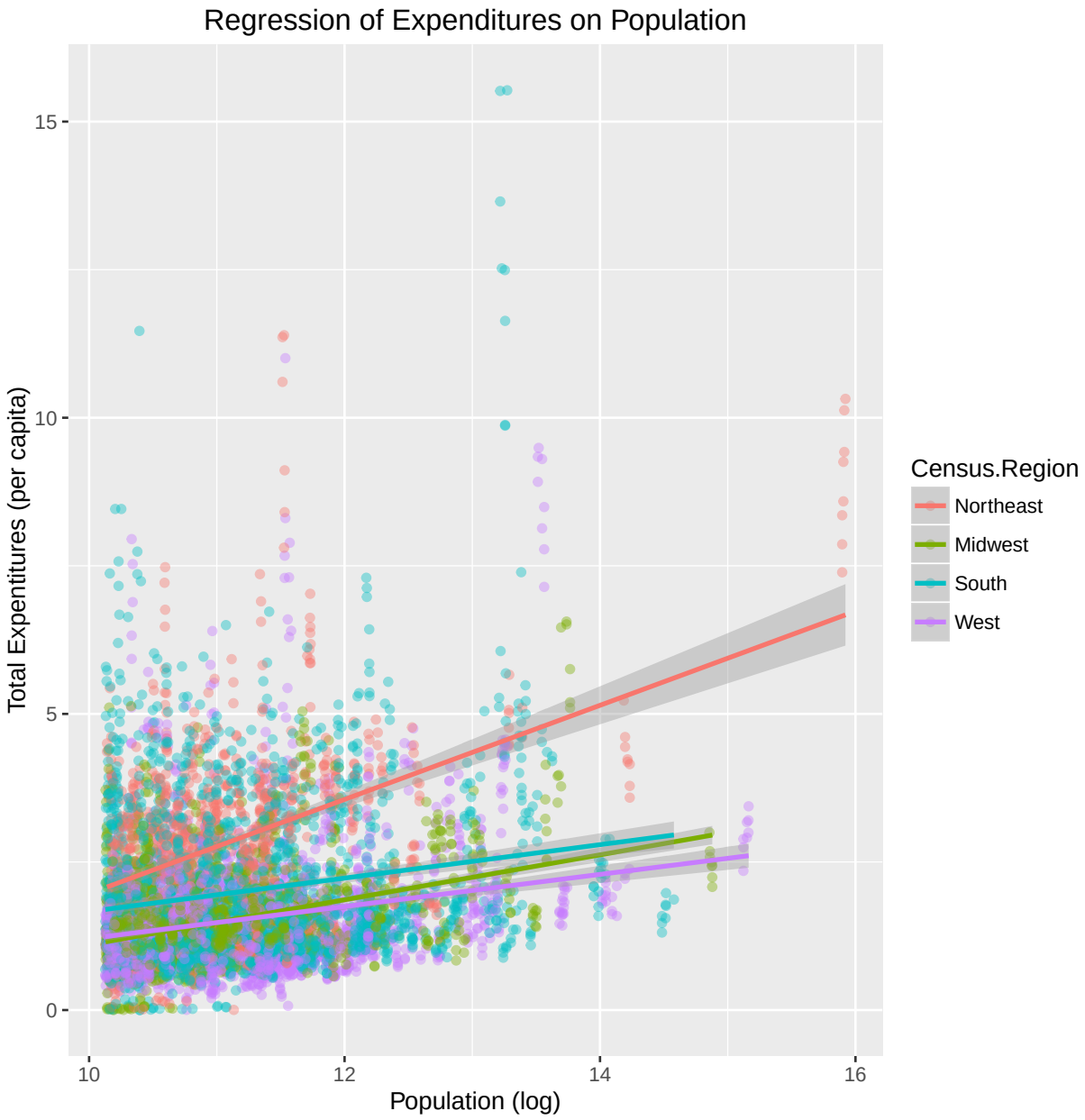


If you want to use a specific method to produce a trend line, you can also add the `stat_smooth` function. Note that `stat_smooth` is a separate function— i.e., it is not an argument to the `qplot()` function. After you end the `qplot()` function, you will add the additional function to generate the line (`+ stat_smooth(method="lm", formula=y~x)`).

This example produces linear regression lines for each Census.Region. For one line estimated with all available data, you can leave out the `color=` argument.

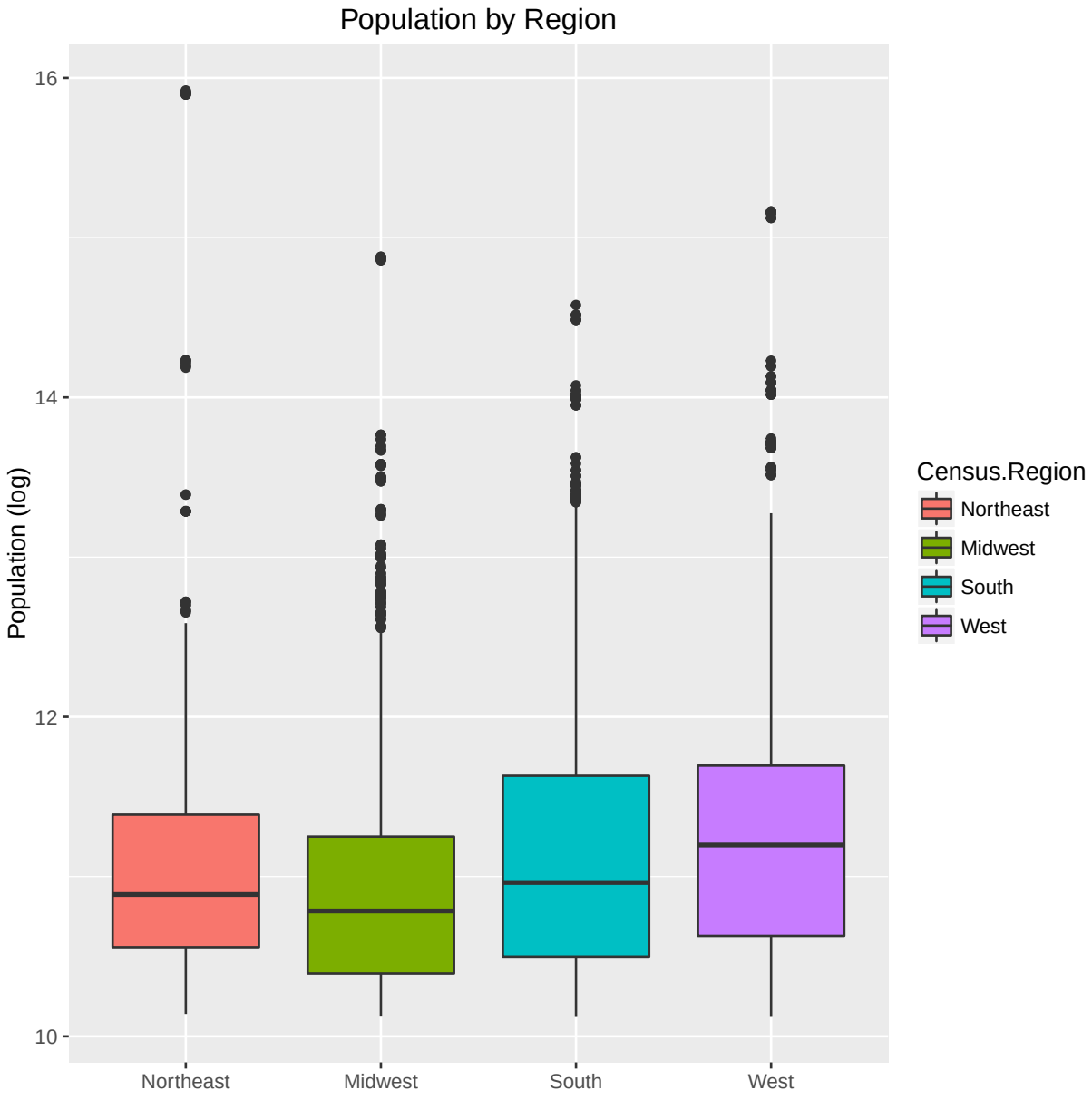
```
# linear regression lines by region
qplot(log(Population), Total.Expenditure.PC, data=COG.fips, geom=c("point"),
      color=Census.Region, alpha=I(.4),
      main="Regression of Expenditures on Population",
      xlab="Population (log)", ylab="Total Expentitures (per capita)") +
```

```
      stat_smooth(method="lm", formula=y~x)
```



Boxplot

```
# boxplot
qplot(Census.Region, log(Population), data=COG.fips, geom="boxplot",
      fill=Census.Region, main="Population by Region",
      xlab="", ylab="Population (log)")
```



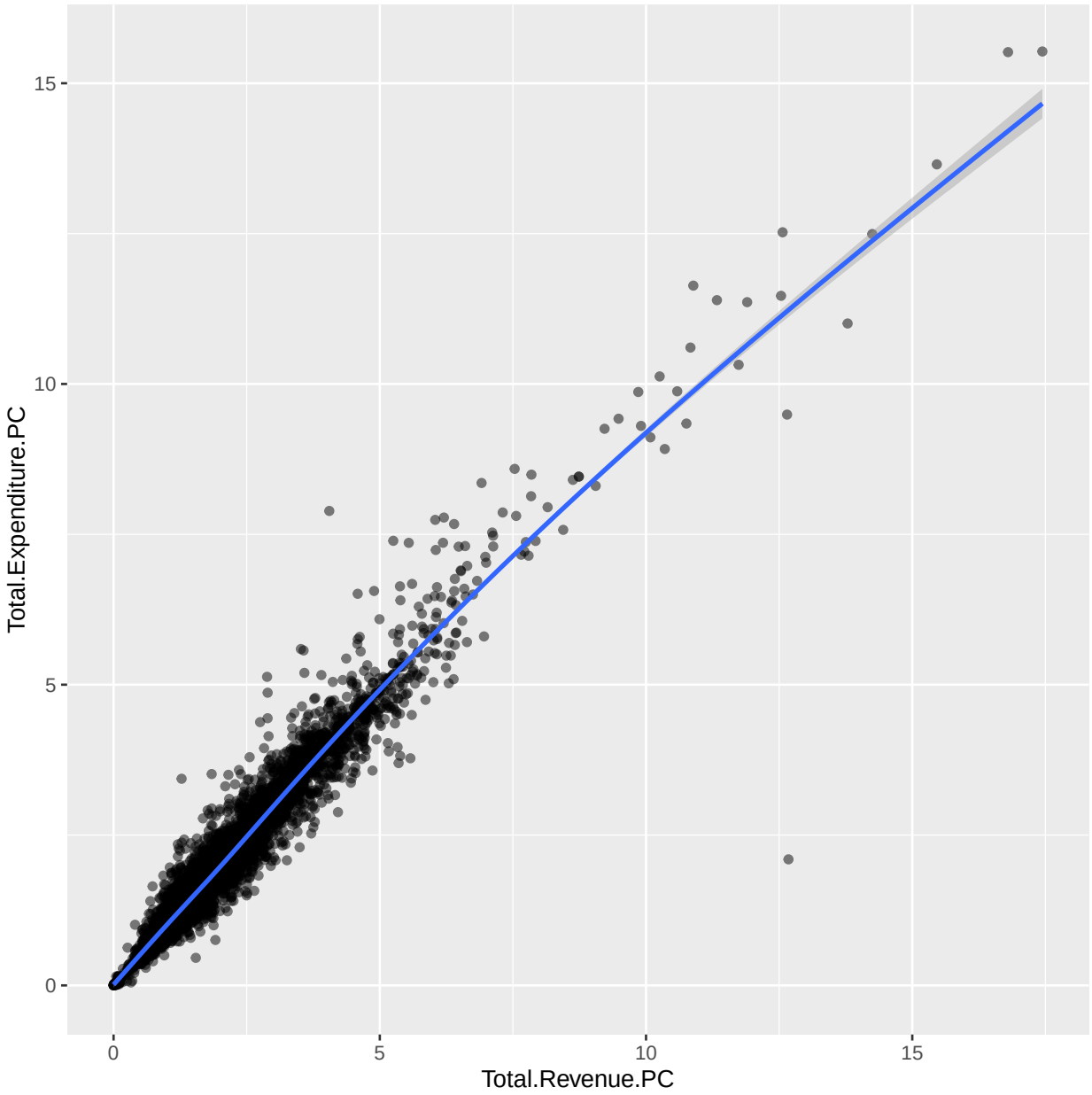
2.2 Using `ggplot()`

The `ggplot2` package builds plots in layers. With the `ggplot()` function, you first supply the data frame, and the aesthetics (aes)—i.e., the data the figure will present. Then, you can add multiple layers to the figure. These layers can include specifications of the figure as well as formatting details. You can save the plot to an object, and once you store a plot, you can add additional layers or make changes to the plot. You will find several examples here, but there are many more options you can discover online.

As you can see, the `ggplot()` function works a bit differently and uses different syntax than `qplot()`. In the initial `ggplot()` function, you specify the data and the aesthetics of the plot, i.e., which variable(s) to plot and arguments to specify groups if applicable. Then, you add a series of additional

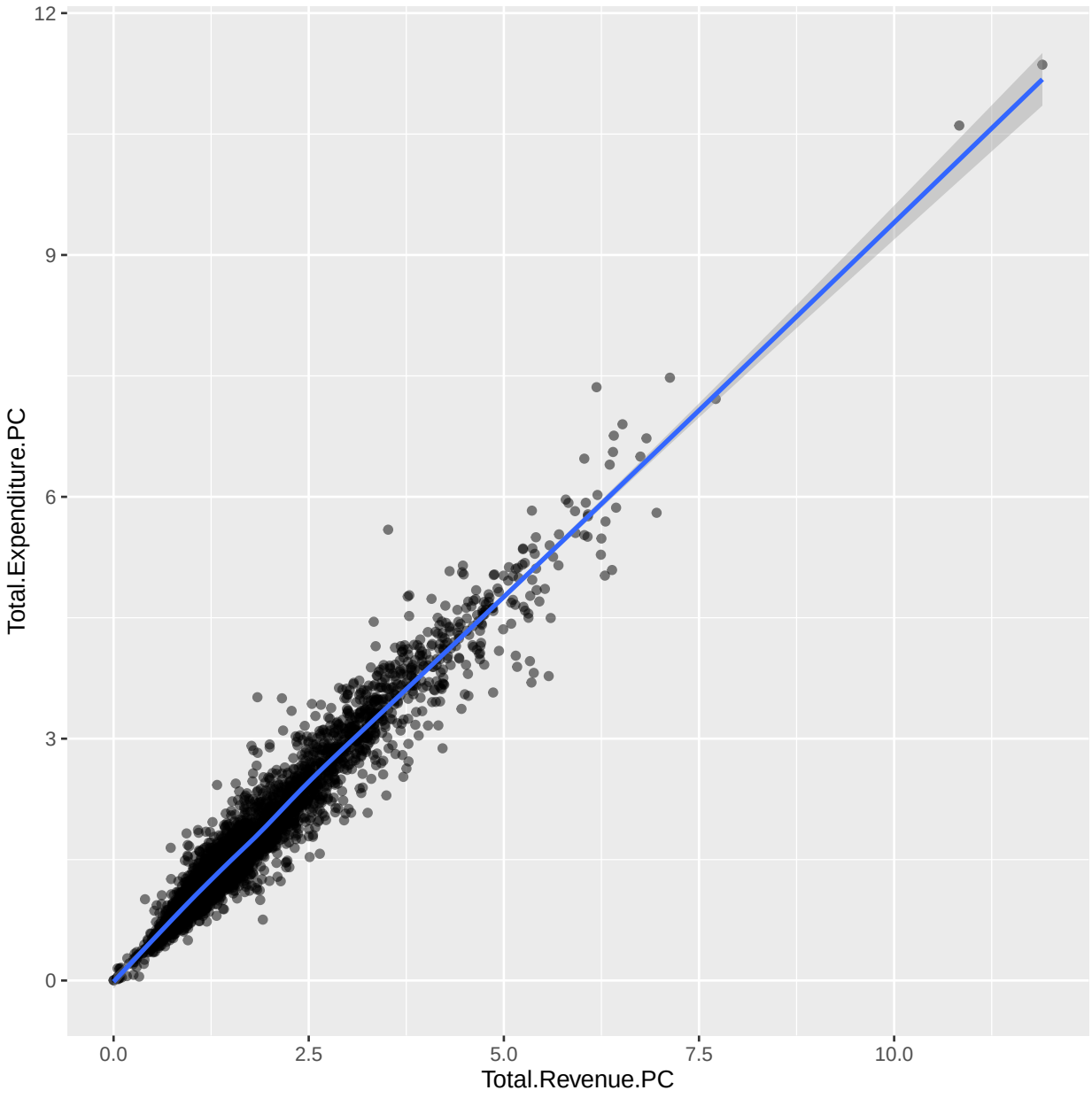
functions. These functions add features or layers to your plot. You add additional functions with an addition sign (+). Your code can be spread over multiple lines, but when you are adding functions, be sure to end each line with + to indicate that the code continues onto the next line.

```
plot <- ggplot(COG.fips, aes(x=Total.Revenue.PC, y=Total.Expenditure.PC)) +
    geom_point(alpha=.5) +
    geom_smooth()

plot
```



Note that you can use functions like `filter()` or `subset()` inside the `ggplot()` function to plot a subset of your data.

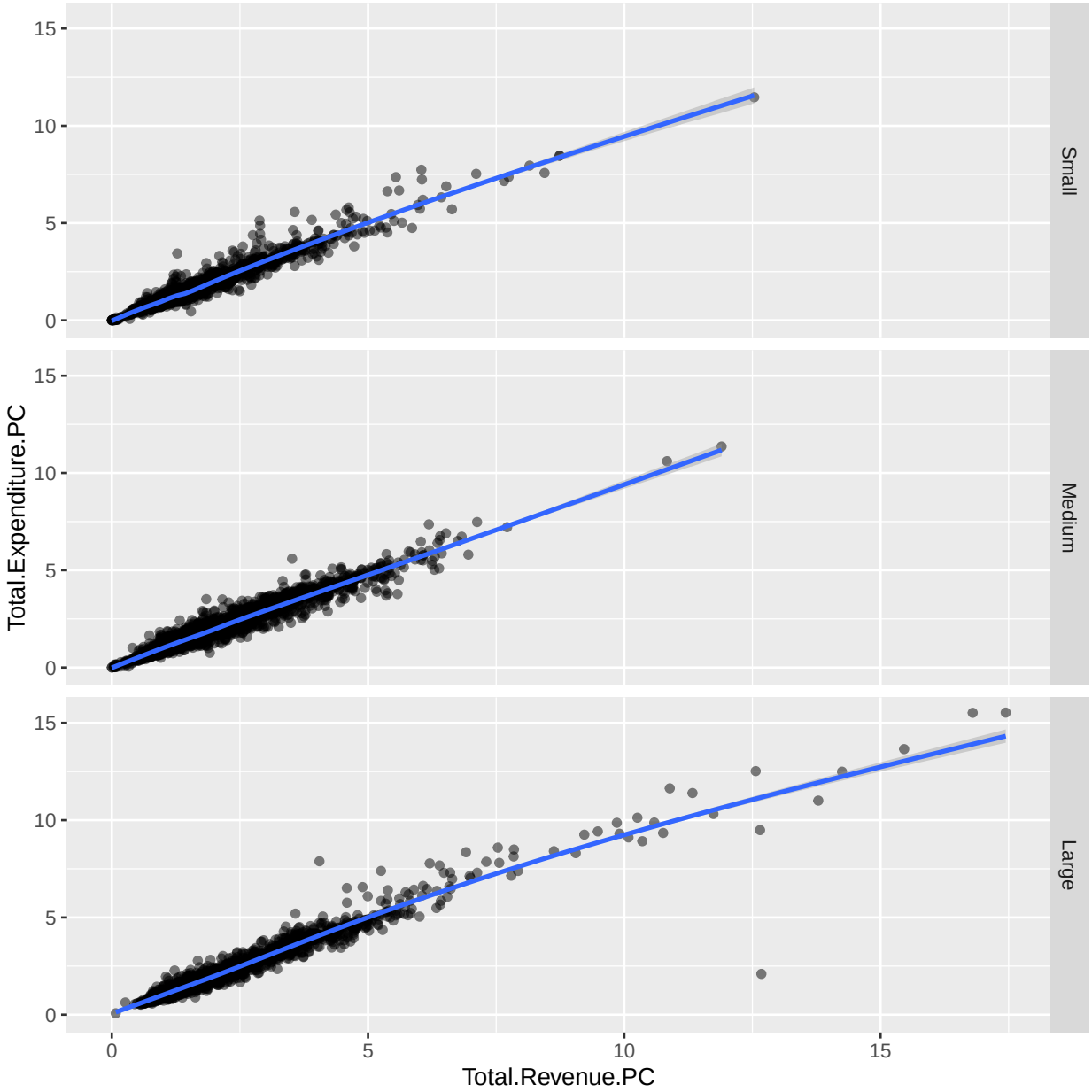
```
plot_2 <- ggplot(filter(COG.fips, Population.Category == "Medium"),
                 aes(x=Total.Revenue.PC, y=Total.Expenditure.PC)) +
    geom_point(alpha=.5) +
    geom_smooth()

plot_2
```



To incorporate facets, use the `facet_grid()` function, specifying rows and columns as with `qplot()`.

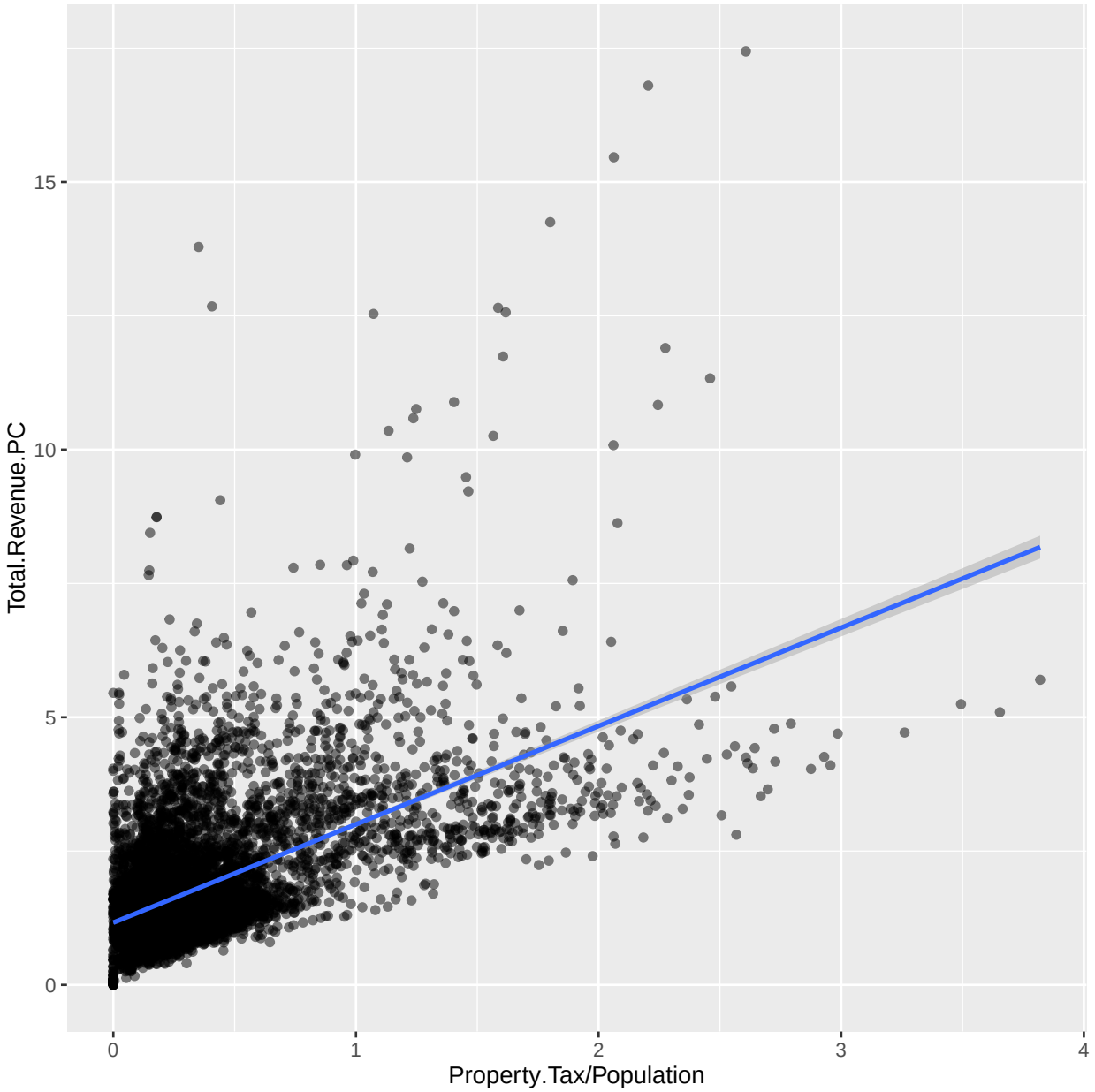
```
plot_3 <- ggplot(COG.fips, aes(x=Total.Revenue.PC, y=Total.Expenditure.PC)) +
    geom_point(alpha=.5) +
    geom_smooth() +
```

```
    facet_grid(Population.Category ~ .)

plot_3
```



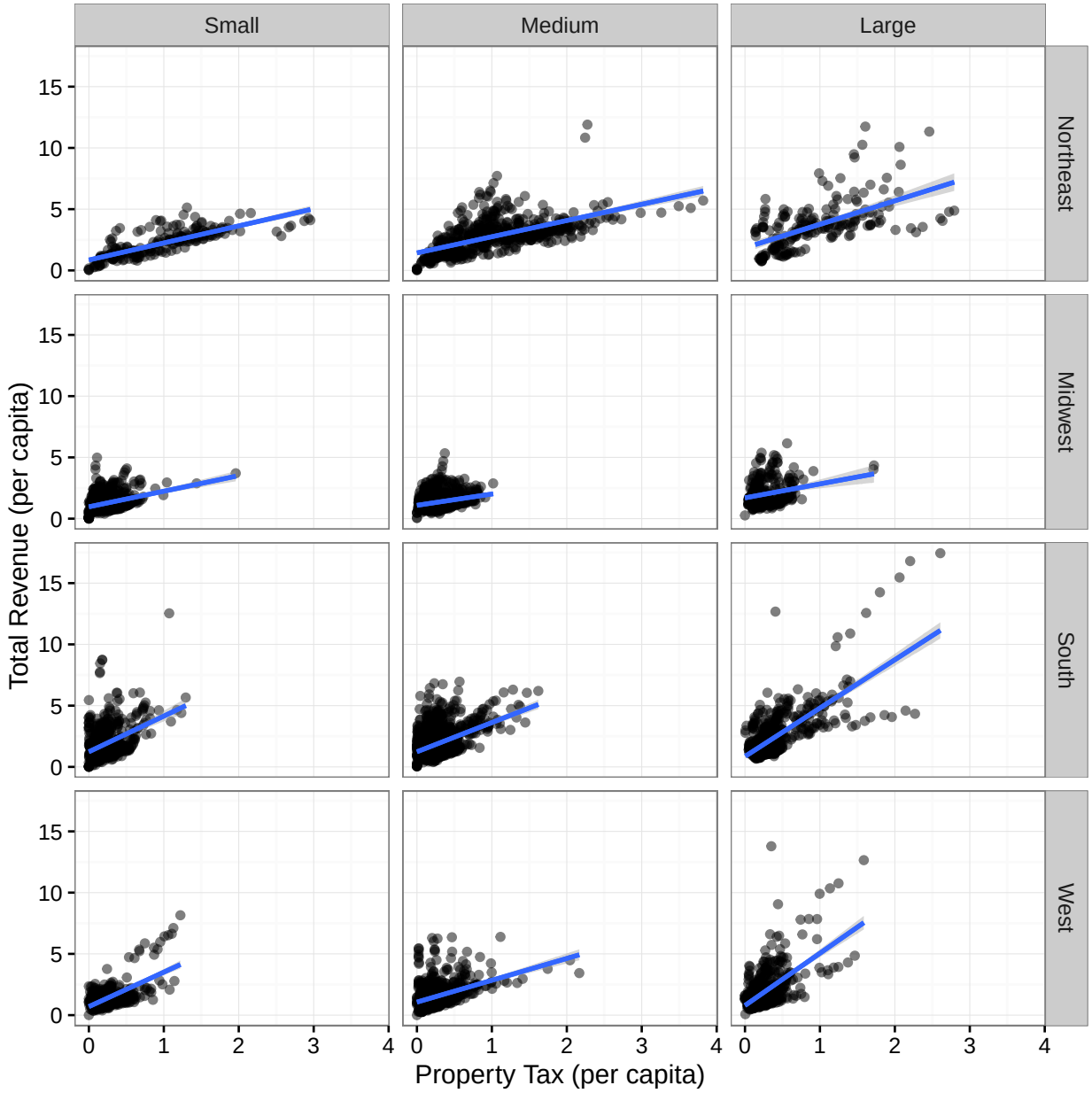
You can include a fitted line by adding the `method=` and `formula=` arguments to the `geom_smooth()` function.

```
plot_4 <- ggplot(COG.fips, aes(x=Property.Tax/Population, y=Total.Revenue.PC)) +
    geom_point(alpha=.5) +
    geom_smooth(method="lm", formula=y~x)

plot_4
```



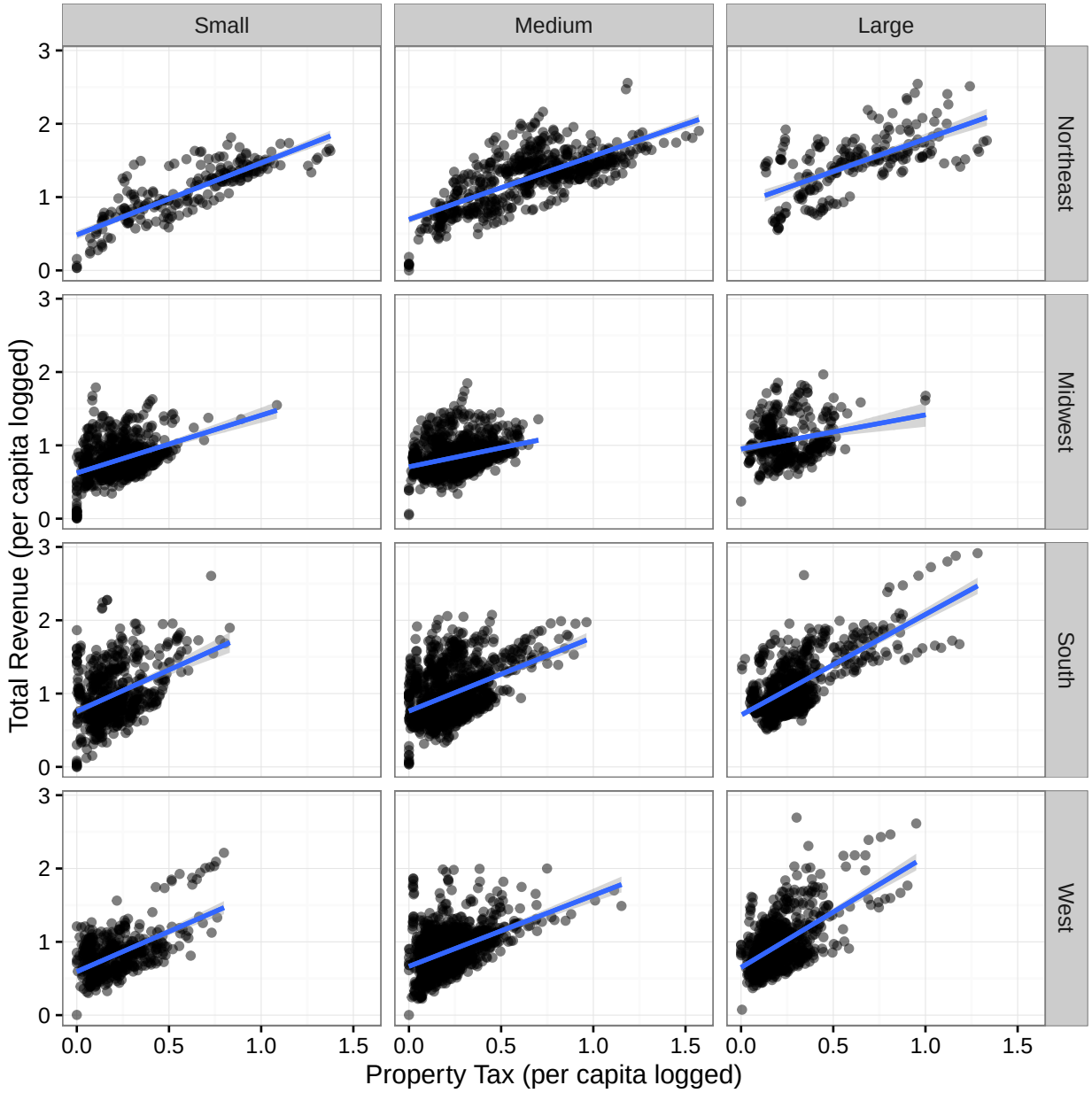
In this example, a saved plot is modified by adding and replacing features, such as facets. Note the `se=FALSE` argument to the `geom_smooth()` function omits the confidence interval around the fitted line. The `theme_bw()` function eliminates the grey shading that appears by default.

```
plot_5 <- plot_4 + facet_grid(Census.Region ~ Population.Category) +
    geom_smooth(method="lm", formula=y~x, se=FALSE) +
    theme_bw() +
    xlab("Property Tax (per capita)") +
    ylab("Total Revenue (per capita)")

plot_5
```



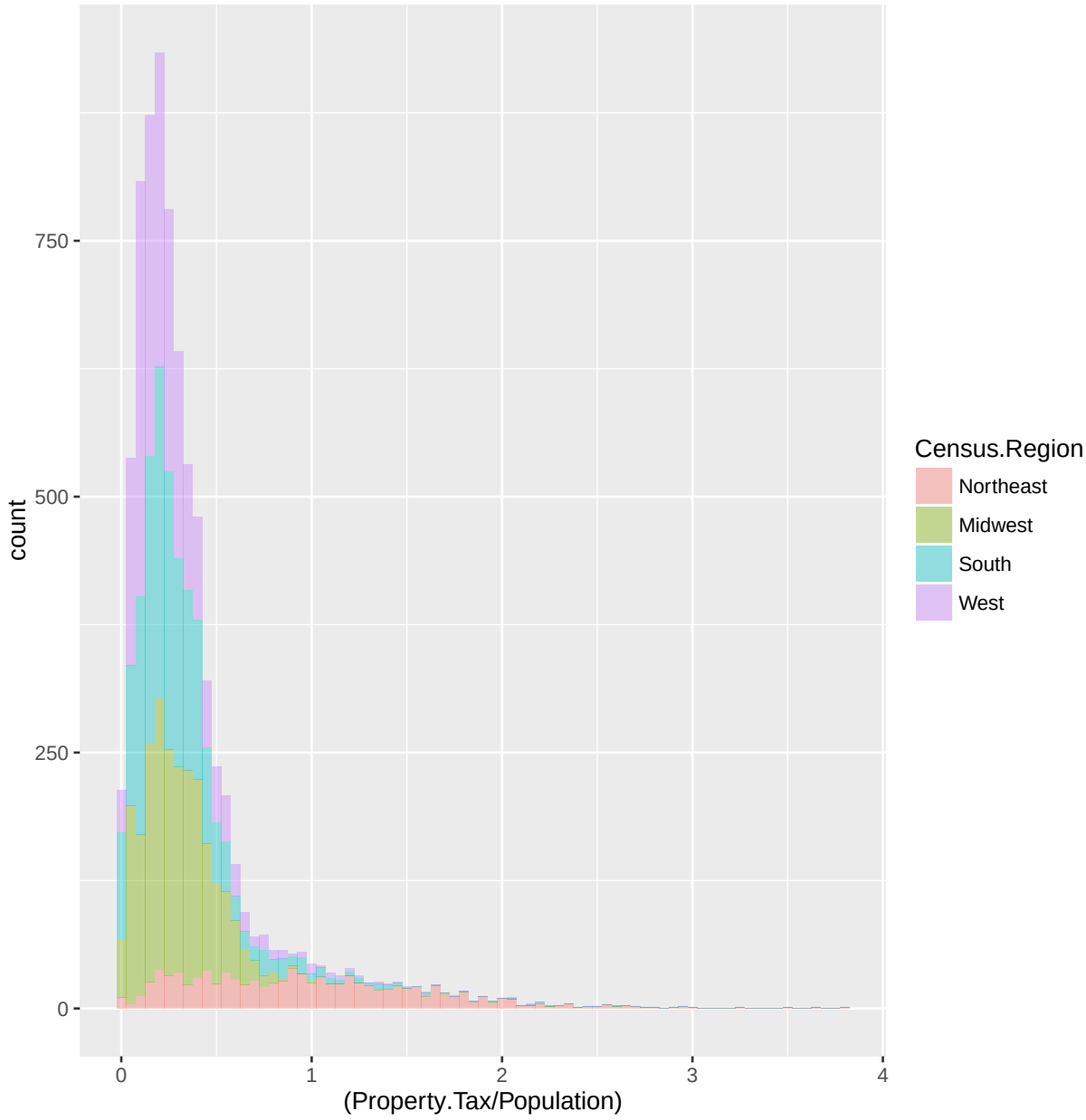
One of the most convenient features of `ggplot2` is that you can actually replace the data and/or other parameters of an existing saved plot. This code produces a plot similar to the saved plot (`plot_5`) but replaces the variables with logged values and changes the axis labels accordingly.

```
plot_6 <- plot_5 %+% aes(x=log((Property.Tax/Population)+1),
                         y=log(Total.Revenue.PC+1)) +
    xlab("Property Tax (per capita logged)") +
    ylab("Total Revenue (per capita logged)")

plot_6
```



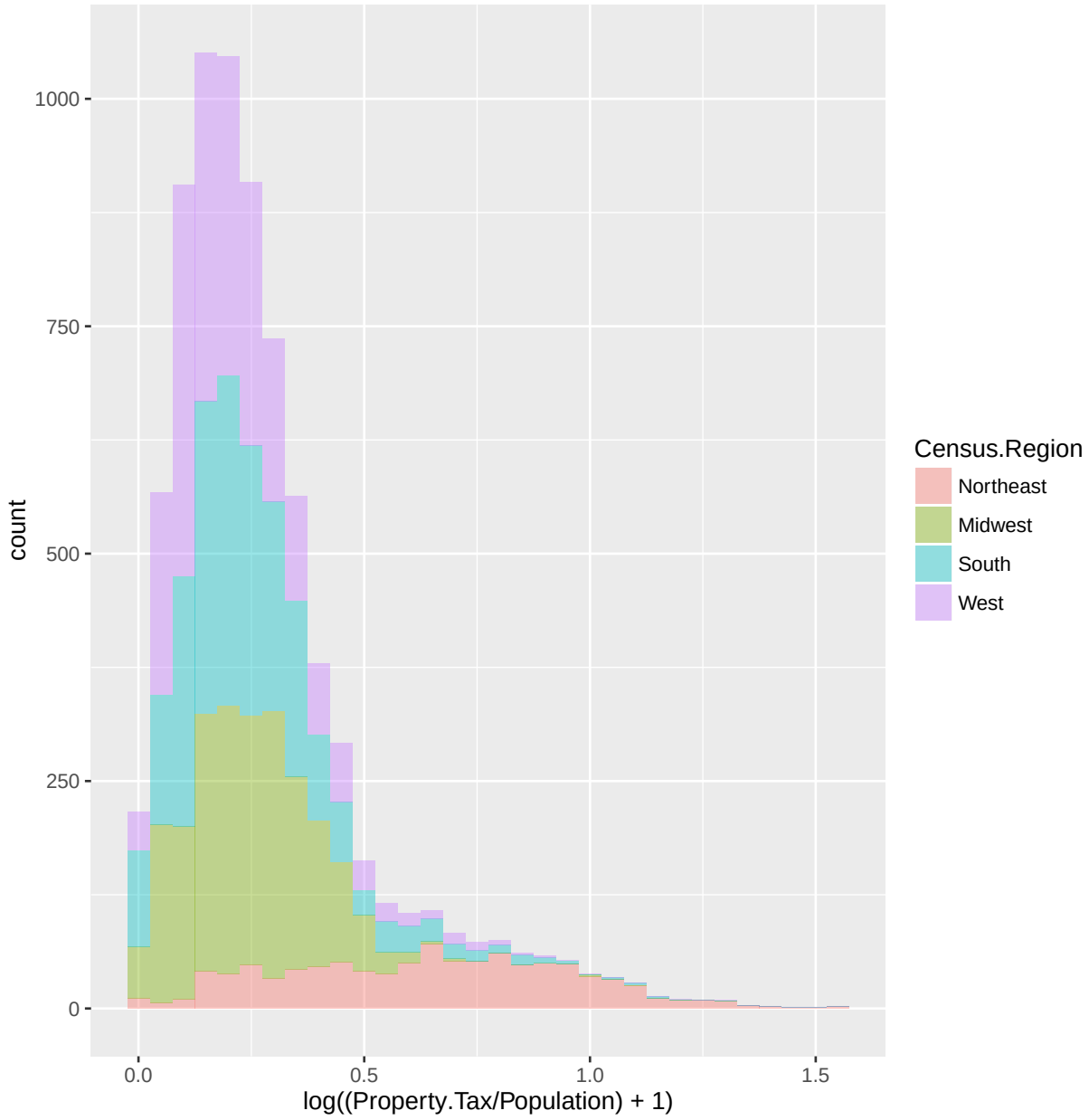
An example of histograms by Census.Region.

```
plot_7 <- ggplot(COG.fips, aes(x=(Property.Tax/Population), fill=Census.Region)) +
    geom_histogram(binwidth=.05, alpha=.4)

plot_7
```



Another example that includes code to replace part of an existing plot.

```
plot_8 <- plot_7 %+% aes(x=log((Property.Tax/Population)+1), fill=Census.Region)

plot_8
```



2.3 Exporting & saving plots

There are multiple ways to export and save a plot for use outside of R. Below, you will find code for two options. You can also export plots using the *Export* icon on the *Plots* tab in RStudio. However, if you include code to export a plot in your script, you can easily update a plot if something in your data or analysis changes. In fact, any time you re-run the code to export a plot, R will overwrite any existing file of the same name. If you prefer, you can avoid overwriting a file every time you run the code by commenting out the code after you export the plot.

You can print it to file. This method works with `ggplot` as well as plotting functions in the `base` package (e.g., `hist()`, `plot()`). You will need to run all three functions.

```
## name the file
pdf("plot_prop_tax.pdf")
## print the object (plot)
print(plot_4)
## close the figure file
dev.off()
```

The `ggsave()` function allows you to save objects made with the `ggplot2` package. Name the file and specify the data frame. You can add additional specifications such as height and/or width.

```
## ggsave
ggsave(file="revenue_plot.pdf", plot_6)
```

3 Exporting Tables

There are a variety of packages for exporting tables (`xtable` is a well-known option), and some packages are discipline-specific. Especially among social scientists, the `stargazer` package is a popular option for making tables of all kinds. The `stargazer()` function requires at least one argument—the object that contains your model, data, or table. However, you can add many, many additional arguments to customize your table. The example below includes several options, but you can find detailed documentation online.

The `stargazer()` function also allows you to export tables in multiple file formats, including LaTeX and HTML (HTML files can be easily opened with or added to MS Word documents). The appearance of the table will be quite similar in either format, but once the HTML file is opened in MS Word, the table can be edited in MS Word. Whenever you run the code to export a table, the existing file will be overwritten, which can make it easier to update documents, assignments, etc. If you prefer not to overwrite the file every time you run your code, simply comment it out.

Shortcut: Notice the `with()` function, which allows you to specify a data frame so that you do not have to link each variable to the data frame.

```
## summary stats
with(COG.fips, summary(Total.Expenditure.PC))

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   1.044   1.423   1.818   2.207  15.530

with(COG.fips, summary(Total.Revenue.PC))

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   1.064   1.436   1.836   2.233  17.440

with(COG.fips, summary(Total.Taxes.PC))

##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.0000  0.3965  0.5330  0.6513  0.7515  8.9280
```

An example of a summary statistics table with code to produce an HTML file.

```
library(stargazer)

stargazer(select(COG.fips, Total.Expenditure.PC, Total.Revenue.PC, Total.Taxes.PC),
          type = "html", title = "Summary Statistics",
          out = "summary_statistics_table.htm")
```

The code below produces a LaTeX file, and you can see the table it generates below.

```
stargazer(select(COG.fips, Total.Expenditure.PC, Total.Revenue.PC, Total.Taxes.PC),
          title = "Summary Statistics",
          out = "summary_statistics_table.tex")
```

Table 1: Summary Statistics

Statistic	N	Mean	St. Dev.	Min	Max
Total.Expenditure.PC	7,713	1.818	1.246	0.000	15.528
Total.Revenue.PC	7,713	1.836	1.267	0.000	17.443
Total.Taxes.PC	7,713	0.651	0.464	0.000	8.928

An example of a table presenting regression results:

```
## some regression models

fit_1 <- lm(Total.Expenditure.PC ~ Total.Taxes.PC, data=COG.fips)
summary(fit_1)

##
## Call:
## lm(formula = Total.Expenditure.PC ~ Total.Taxes.PC, data = COG.fips)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.9687 -0.5328 -0.2566  0.2250  8.3748
##
## Coefficients:
##                Estimate Std. Error t value Pr(>|t|)
## (Intercept)     0.64158    0.01806   35.52   <2e-16 ***
## Total.Taxes.PC  1.80639    0.02258   79.99   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9211 on 7711 degrees of freedom
## Multiple R-squared:  0.4535,Adjusted R-squared:  0.4534
## F-statistic:  6398 on 1 and 7711 DF,  p-value: < 2.2e-16
```

```
fit_2 <- lm(Total.Expenditure.PC ~ Total.Taxes.PC
            + Census.Region
            , data=COG.fips)
summary(fit_2)
```

```
##
## Call:
## lm(formula = Total.Expenditure.PC ~ Total.Taxes.PC + Census.Region,
##     data = COG.fips)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.0581 -0.5079 -0.2205  0.2529  8.6386
##
## Coefficients:
##                       Estimate Std. Error t value Pr(>|t|)
## (Intercept)            0.95633    0.03861  24.767   <2e-16 ***
## Total.Taxes.PC         1.71953    0.02378  72.304   <2e-16 ***
## Census.RegionMidwest -0.36116    0.03730  -9.682   <2e-16 ***
## Census.RegionSouth   -0.04813    0.03603  -1.336    0.182
## Census.RegionWest    -0.48286    0.03612 -13.368   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.9005 on 7708 degrees of freedom
## Multiple R-squared:  0.4778,Adjusted R-squared:  0.4775
## F-statistic:  1763 on 4 and 7708 DF,  p-value: < 2.2e-16
```

```
fit_3 <- lm(Total.Expenditure.PC ~ Total.Taxes.PC
            + Population
            + Census.Region
            , data=COG.fips)
summary(fit_3)
```

```
##
## Call:
## lm(formula = Total.Expenditure.PC ~ Total.Taxes.PC + Population +
##     Census.Region, data = COG.fips)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.8993 -0.5034 -0.2145  0.2300  8.6835
##
## Coefficients:
##                        Estimate Std. Error t value Pr(>|t|)
## (Intercept)           9.678e-01  3.818e-02  25.349   <2e-16 ***
## Total.Taxes.PC        1.650e+00  2.407e-02  68.530   <2e-16 ***
## Population            4.173e-07  3.096e-08  13.478   <2e-16 ***
```

```
## Census.RegionMidwest -3.747e-01  3.688e-02 -10.158   <2e-16 ***
## Census.RegionSouth   -6.528e-02  3.563e-02  -1.832    0.067 .
## Census.RegionWest    -5.050e-01  3.574e-02 -14.128   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.8901 on 7707 degrees of freedom
## Multiple R-squared:  0.4898,Adjusted R-squared:  0.4895
## F-statistic:  1480 on 5 and 7707 DF,  p-value: < 2.2e-16

fit_4 <- lm(Total.Expenditure.PC ~ Total.Taxes.PC
            + Population
            + Census.Region
            + Population.Category
            , data=COG.fips)
summary(fit_4)

##
## Call:
## lm(formula = Total.Expenditure.PC ~ Total.Taxes.PC + Population +
##     Census.Region + Population.Category, data = COG.fips)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.8292 -0.5018 -0.2026  0.2192  8.5027
##
## Coefficients:
##                              Estimate Std. Error t value Pr(>|t|)
## (Intercept)                 9.289e-01  4.161e-02  22.325   <2e-16 ***
## Total.Taxes.PC              1.639e+00  2.392e-02  68.515   <2e-16 ***
## Population                  2.973e-07  3.264e-08   9.110   <2e-16 ***
## Census.RegionMidwest       -3.747e-01  3.666e-02 -10.222   <2e-16 ***
## Census.RegionSouth         -9.476e-02  3.549e-02  -2.670   0.0076 **
## Census.RegionWest          -5.474e-01  3.569e-02 -15.338   <2e-16 ***
## Population.CategoryMedium   1.867e-02  2.478e-02   0.753   0.4513
## Population.CategoryLarge    2.863e-01  3.035e-02   9.433   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.8834 on 7705 degrees of freedom
## Multiple R-squared:  0.4976,Adjusted R-squared:  0.4971
## F-statistic:  1090 on 7 and 7705 DF,  p-value: < 2.2e-16
```

Code to produce an HTML file.

```
stargazer(fit_1, fit_2, fit_3, fit_4,
          omit = c("Census.Region", "Population.Category"),
          add.lines = list(c("Fixed Effects", "N/A", "Region",
```

```
                "Region", "Region, Size")),
          title = "Results",
          type = "html",
          out = "regression_table_2.htm")
```

This example uses the `stargazer()` function to produce LaTeX code, and you can see the resulting table.

```
stargazer(fit_1, fit_2, fit_3, fit_4,
          omit = c("Census.Region", "Population.Category"),
          omit.stat = c("ser", "f"),
          add.lines = list(c("Fixed Effects", "N/A", "Region",
                             "Region", "Region, Size")),
          title = "Results",
          out = "regression_table_2.tex")
```

Table 2: Results

	Dependent variable:			
	Total.Expenditure.PC			
	(1)	(2)	(3)	(4)
Total.Taxes.PC	1.806***	1.720***	1.650***	1.639***
	(0.023)	(0.024)	(0.024)	(0.024)
Population			0.00000***	0.00000***
			(0.00000)	(0.00000)
Constant	0.642***	0.956***	0.968***	0.929***
	(0.018)	(0.039)	(0.038)	(0.042)
Fixed Effects	N/A	Region	Region	Region, Size
Observations	7,713	7,713	7,713	7,713
R^2	0.453	0.478	0.490	0.498
Adjusted R^2	0.453	0.478	0.489	0.497
Note:				*p<0.1; **p<0.05; ***p<0.01