Intro

Will Doyle

2023-09-06

Introduction: Tidyverse and Tidymodels

In this introduction, we'll get up and running with using `.Rmd` files, `tidyverse` and `tidymodels`.

The Data

Definition: The Civil Rights Data Collection (CRDC) is a survey conducted every other school year by the U.S. Department of Education's Office for Civil Rights (OCR). It has been in operation since 1968.

Coverage: The 2017–18 CRDC, like its predecessors from 2011–12, 2013–14, and 2015–16, gathers data from all public local educational agencies (LEA) and schools. This includes juvenile justice facilities, charter schools, alternative schools, and schools catering to students with disabilities.

Purpose:

The CRDC gathers data on primary civil rights indicators that show access and barriers to educational opportunities from early childhood to grade 12.

It is a crucial tool for the OCR to ensure that entities receiving Federal financial assistance from the Department don't discriminate based on race, color, national origin, sex, and disability. The OCR uses the CRDC data to:

- Investigate complaints alleging discrimination.
- Determine violations of Federal civil rights laws.
- Initiate proactive compliance reviews targeting specific or widespread civil rights compliance issues.
- Offer policy guidance and technical assistance to various stakeholders including educational institutions, parents, and students.

The CRDC serves as a significant resource for other Department offices, Federal agencies, policymakers, researchers, educators, school officials, parents, students, and the general public who are interested in data related to student equity and opportunity.

Libraries

In R, there are collections of functions and methods that allow you to perform many actions without writing your code. These collections are called libraries or packages. Below I "activate" libraries for use in this session, called "loading."

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages ------------------------ tidyverse 2.0.0 --
## v dplyr     1.1.2     v readr     2.1.4
## v forcats   1.0.0     v stringr   1.5.0
## v ggplot2   3.4.2     v tibble    3.2.1
## v lubridate 1.9.2     v tidyr     1.3.0
## v purrr     1.0.1
```

```
## -- Conflicts ---------------------------------------- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to become errors
```

```
library(tidymodels)
```

```
## -- Attaching packages --------------------------------------- tidymodels 1.1.0 --
## v broom        1.0.5     v rsample      1.1.1
## v dials        1.2.0     v tune         1.1.1
## v infer        1.0.4     v workflows    1.1.3
## v modeldata    1.1.0     v workflowsets 1.0.1
## v parsnip      1.1.0     v yardstick    1.2.0
## v recipes      1.0.6
## -- Conflicts ------------------------------------------ tidymodels_conflicts() --
## x scales::discard() masks purrr::discard()
## x dplyr::filter()   masks stats::filter()
## x recipes::fixed()  masks stringr::fixed()
## x dplyr::lag()      masks stats::lag()
## x yardstick::spec() masks readr::spec()
## x recipes::step()   masks stats::step()
## * Learn how to get started at https://www.tidymodels.org/start/
```

```
library(boxr)
```

```
## boxr: see `vignette("boxr")` on how to authorize to your Box account.
```

```
library(here)
```

```
## here() starts at /Users/doylewr/ml_notes
```

In R, the concepts of installing and loading libraries are fundamental, and while they might seem similar, they serve different purposes. Let's break down the differences:

Installing Libraries:

What it means: Installing a library means downloading the package (which contains the library) from a repository (like CRAN or Bioconductor) and saving it on your local machine.

How often you do it: You only need to install a library once on your machine (unless you need to update it to a newer version later on).

Command: The typical command to install a package (and its library) is install.packages("package_name"). For example, to install the dplyr package, you'd use install.packages("dplyr").

Storage: Once installed, the package is stored on your computer's disk, and you don't need an internet connection to use it in the future (unless you're updating it).

Loading Libraries:

What it means: Loading a library means making the functions and datasets of a previously installed package available in your current R session. It's like opening a toolbox you have in your garage so that you can use the tools inside.

How often you do it: You need to load a library every time you start a new R session and want to use the functions or datasets from that library.

Command: The typical command to load a library is library(package_name). For example, to load the dplyr library, you'd use library(dplyr).

Session-specific: Loading a library affects only your current R session. If you close R and then reopen it, you'll need to load the library again if you want to use its functions.

In Simple Terms:

Think of installing a library in R as buying a physical book and putting it on your bookshelf. You only buy the book once. Loading a library, on the other hand, is like taking that book off the shelf and opening it to read. Every time you want to read the book, you need to take it off the shelf and open it, even if you've read it before.

Reading in Data

The CRDC dataset we'll be working with includes disciplinary rates for in and out of school suspensions, with separate rates for students by race and ethnicity. For this example, we'll focus just on in-school suspension rates, the variabl `iss`.

Let's start by opening the `read_csv` function.

```
cr<-read_csv(here("data","crdc.csv"))
```

```
## Rows: 13270 Columns: 68
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (6): lea_name, county_name, city_name, state_name, region, urbanicity
## dbl (57): year, leaid, stu_per_frl, stu_per_ell, stu_per_speced, stu_per_bla...
## lgl  (5): ngh_ppexp, ngh_percharter, ngh_inc, ngh_crimerate, ngh_enrollment
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

Libraries: In R, there are collections of functions and methods that allow you to perform many actions without writing your code. These collections are called libraries or packages. In your code, you're using functions from the "here" library and another library for reading CSV files.

Code Breakdown: - `cr <-`: This is an assignment operation. It means that the result of the operation on the right side of the `<-` will be stored in a variable named `cr`. Think of `cr` as a container where you're saving the data you're about to read in.

- `read_csv()`: This is a function that reads in data from a CSV (Comma-Separated Values) file and turns it into a format that R can understand and manipulate. CSV files are a type of file that represents data in a tabular form, where each row is a new line and columns are separated by commas.
- `here("data","crdc.csv")`: The `here()` function is used to create file paths. It's particularly useful because it makes your code more portable (i.e., it can be run on different computers without changing the file path). In this case, `here("data","crdc.csv")` is creating a path to the "crdc.csv" file located inside a "data" folder. So, instead of writing out the full path to the file, you're just specifying the folder and file name, and the `here()` function figures out the rest.

Calculating Conditional Means

We'll do a couple of simple conditional means, using state and percent free/reduced lunch as predictors.

ISS rates by state

```
cr%>%
  group_by(state_name)%>%
  summarize(avg_iss=mean(iss,na.rm=TRUE))%>%
  arrange(-avg_iss)%>%
  print(n=52)
```

```
## # A tibble: 51 x 2
##    state_name           avg_iss
##    <chr>                  <dbl>
##  1 Mississippi            11.7
##  2 Georgia                11.6
##  3 Arkansas               11.3
##  4 South Carolina         11.2
##  5 Texas                  10.1
##  6 Florida                 9.46
##  7 Missouri                9.09
##  8 Louisiana               8.93
##  9 Alabama                 8.68
## 10 North Carolina          8.38
## 11 Kentucky                8.24
## 12 Delaware                8.10
## 13 Tennessee               7.56
## 14 West Virginia           7.41
## 15 Virginia                7.05
## 16 Oklahoma                6.74
## 17 Wyoming                 5.04
## 18 New York                4.91
## 19 Arizona                 4.78
## 20 New Mexico              4.55
## 21 South Dakota            4.52
## 22 Illinois                4.47
## 23 Indiana                 4.40
## 24 Kansas                  4.26
## 25 Alaska                  4.13
## 26 Montana                 4.10
## 27 Colorado                4.08
## 28 Minnesota               4.06
## 29 Nevada                  4.05
## 30 Oregon                  3.92
## 31 Pennsylvania            3.90
## 32 Idaho                   3.84
## 33 Nebraska                3.84
## 34 Connecticut             3.71
## 35 Michigan                3.70
## 36 Ohio                    3.70
## 37 New Hampshire           3.68
## 38 Washington              3.50
## 39 Iowa                    3.33
## 40 Vermont                 3.15
## 41 Wisconsin               2.91
## 42 North Dakota            2.80
## 43 Maryland                2.71
## 44 Maine                   2.66
## 45 Rhode Island            2.60
## 46 New Jersey              2.39
## 47 California              2.14
## 48 Massachusetts           1.62
## 49 Hawaii                  1.50
## 50 Utah                    1.48
## 51 District of Columbia    1.08
```

The code uses functions from the `tidyverse` set of libraries, particularly `dplyr`. The tidyverse is a collection of R packages designed for data science. `dplyr` is one of these packages, and it provides a set of tools for efficiently manipulating datasets.

The code is taking the school discipline data, organizing it by state, calculating the average number of in-school suspensions for each state, and then sorting and displaying the states based on these averages, from highest to lowest.

Code Breakdown:

`cr %>%`: This is the starting point. The %>% is called a "pipe" operator. It takes the output from one function and feeds it as input to the next function. Think of it as a conveyor belt moving your data from one step to the next.

`group_by(state_name)`: This groups the data by the state_name variable. In other words, it's organizing the data so that all the records for each state are lumped together. This is done so that subsequent operations can be performed on each state's data separately.

`summarize(avg_iss=mean(iss,na.rm=TRUE))`: This step calculates the average in-school suspensions (`iss`) for each state. The `mean()` function calculates the average, and `na.rm=TRUE` ensures that any missing values (represented as `NA` in R) are ignored in the calculation. The result is stored in a new variable called `avg_iss`.

`arrange(-avg_iss)`: This arranges (or sorts) the states based on the average in-school suspensions (`avg_iss`). The `-` sign before `avg_iss` means that the sorting is done in descending order, so states with the highest averages will be at the top.

`print(n=52)`: This displays the results. The `n=52` argument tells R to print the results for all 52 states (assuming there are 52 states in the dataset, which might include territories or special districts).

ISS rates by FRL

Let's use that exact same logic, but this time use a continuous indepnednent variable, percent of students eligible for free or reduced price lunch.

```
cr%>%
  mutate(frl_q=ntile(stu_per_frl,n=4))%>%
  group_by(frl_q)%>%
  summarize(avg_iss=mean(iss,na.rm=TRUE))%>%
  arrange(-avg_iss)
```

```
## # A tibble: 4 x 2
##   frl_q avg_iss
##   <int>   <dbl>
## 1     4    8.07
## 2     3    6.09
## 3     2    4.12
## 4     1    2.30
```

The code is categorizing schools into 4 groups based on the percentage of students receiving free or reduced lunch. Then, for each of these groups, it calculates the average number of in-school suspensions. Finally, it sorts these groups by the average suspensions, from highest to lowest.

Building upon the previous explanation, let's break down this new code:

`mutate(frl_q=ntile(stu_per_frl,n=4))`: - **Function**: `mutate` is used to create or modify columns in a dataset. - **Purpose**: This line is creating a new column named `frl_q`. - `ntile(stu_per_frl,n=4)`: The `ntile` function divides the `stu_per_frl` column into 4 equal groups (quartiles). So, for instance, if `stu_per_frl` represents the percentage of students per school who receive free or reduced lunch, `frl_q` will categorize schools into 4 groups based on this percentage. Group 1 would have the lowest percentages, and group 4 would have the highest.

`group_by(frl_q)`: - As before, this groups the data, but now it's by the `frl_q` quartile groups we just created.

`summarize(avg_iss=mean(iss,na.rm=TRUE))`: - This calculates the average in-school suspensions (`iss`) for each of the `frl_q` quartile groups. As before, any missing values are ignored in the calculation.

`arrange(-avg_iss)`: - This arranges the quartile groups based on the average in-school suspensions (`avg_iss`) in descending order.

Plotting

`ggplot2` in R is like a versatile toolkit for creating visualizations. You start with a blank canvas, specify what data you're painting with, and then add layers (like shapes, colors, and patterns) to create your final artwork. The "grammar" in its name refers to a consistent set of rules you follow to create a wide variety of plots. `ggplot` was Developed by Hadley Wickham as part of the `tidyverse` collection of packages. It's based on the "Grammar of Graphics" concept, which provides a consistent and systematic approach to creating visualizations.

Components: - **Data**: The dataset you want to visualize. - **Aesthetics (aes)**: Mappings of variables in your data to visual properties like axes, colors, and sizes. - **Geoms**: Geometric objects that represent the data. Examples include points (`geom_point()` for scatter plots), lines (`geom_line()` for line graphs), and bars (`geom_bar()` for bar charts). - **Stats**: Statistical transformations that can be applied to the data before plotting, like binning data for histograms. - **Scales**: Control how data values are translated to visual values. For instance, converting a continuous variable to colors or changing axis labels. - **Coordinate Systems**: Define how geoms are positioned, like Cartesian coordinates or polar coordinates. - **Facets**: Allow for creating multiple similar plots split by a factor or variable.

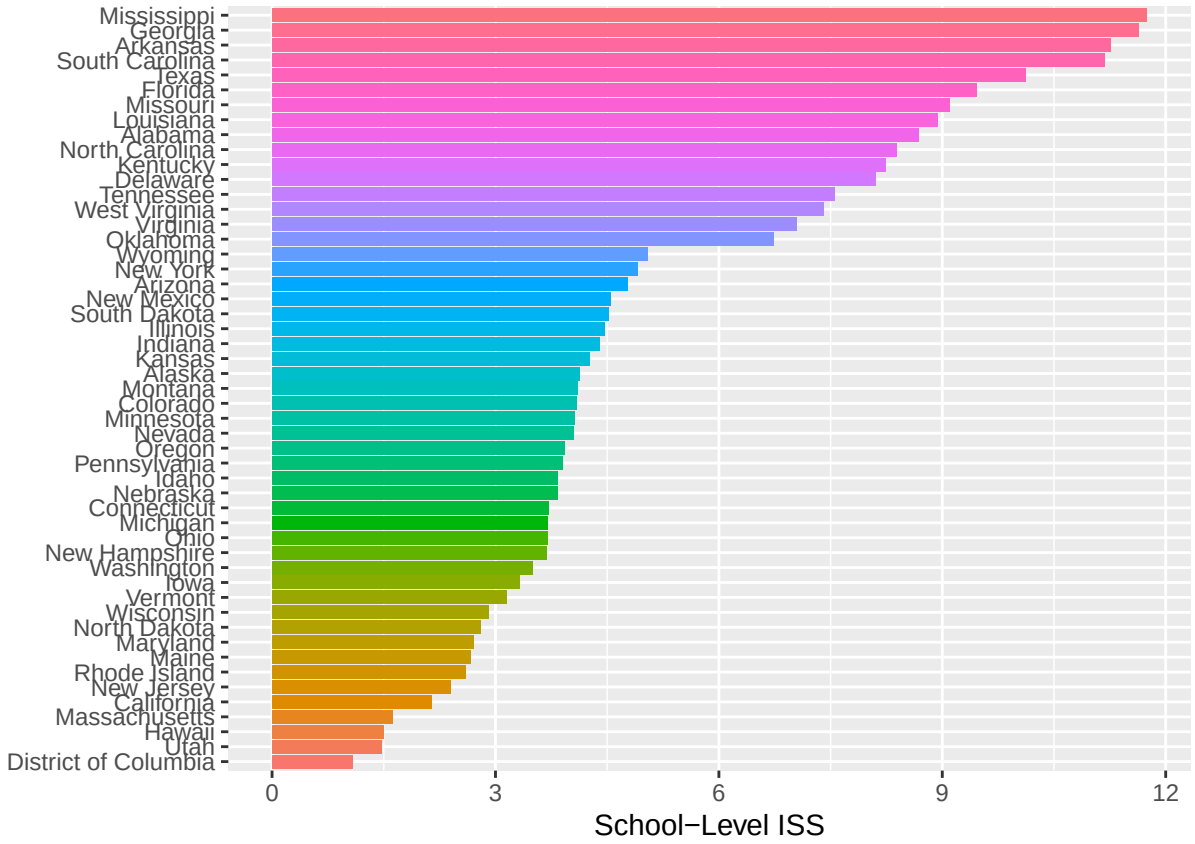
Layered Approach: - `ggplot2` uses a layering paradigm. You start with a base layer specifying the data and global aesthetics and then add layers for geoms, stats, scales, etc. - Layers are added using the `+` operator.

Customization: - `ggplot2` offers extensive customization options for themes, legends, axis labels, and more. - There are predefined themes, and users can create custom themes.

Usage: - Typically starts with the `ggplot()` function, specifying data and global aesthetics. - Additional layers (like geoms) are added using the `+` operator.

ISS rates by state

```
cr%>%
  group_by(state_name)%>%
  summarize(avg_iss=mean(iss,na.rm=TRUE))%>%
  ggplot(aes(x=fct_reorder(state_name,avg_iss),
             y=avg_iss,
             fill=fct_reorder(state_name,avg_iss)))+
  geom_col()+
  coord_flip()+
    theme(legend.position = "none")+
  xlab("")+ylab("School-Level ISS")
```



The code takes the school discipline data, calculates the average number of in-school suspensions for each state, and then creates a horizontal bar chart. Each bar represents a state, and the length of the bar indicates the average number of suspensions. The states are ordered in the chart based on these averages. The bars are colored by state, but there's no legend since the state names are already labeled on the chart.

Let's break down the code in the context of the `ggplot2` explanation:

Data Preparation: - **`group_by(state_name)`**: The data (`cr`) is being grouped by the `state_name` variable, which represents different states. - **`summarize(avg_iss=mean(iss,na.rm=TRUE))`**: For each state, the average number of in-school suspensions (`iss`) is calculated. This average is stored in a new column named `avg_iss`.

Plotting with `ggplot2`: - **`ggplot(aes(...))`**: Initiates the plot creation. The `aes()` function defines the aesthetics or visual properties of the plot. - **`x=fct_reorder(state_name,avg_iss)`**: The x-axis represents states, but they're reordered based on the average in-school suspensions (`avg_iss`). This ensures states with similar averages are grouped together. - **`y=avg_iss`**: The y-axis represents the average in-school suspensions. - **`fill=fct_reorder(state_name,avg_iss)`**: The bars will be colored based on the state, but again, the colors are determined by the reordered states based on suspensions.

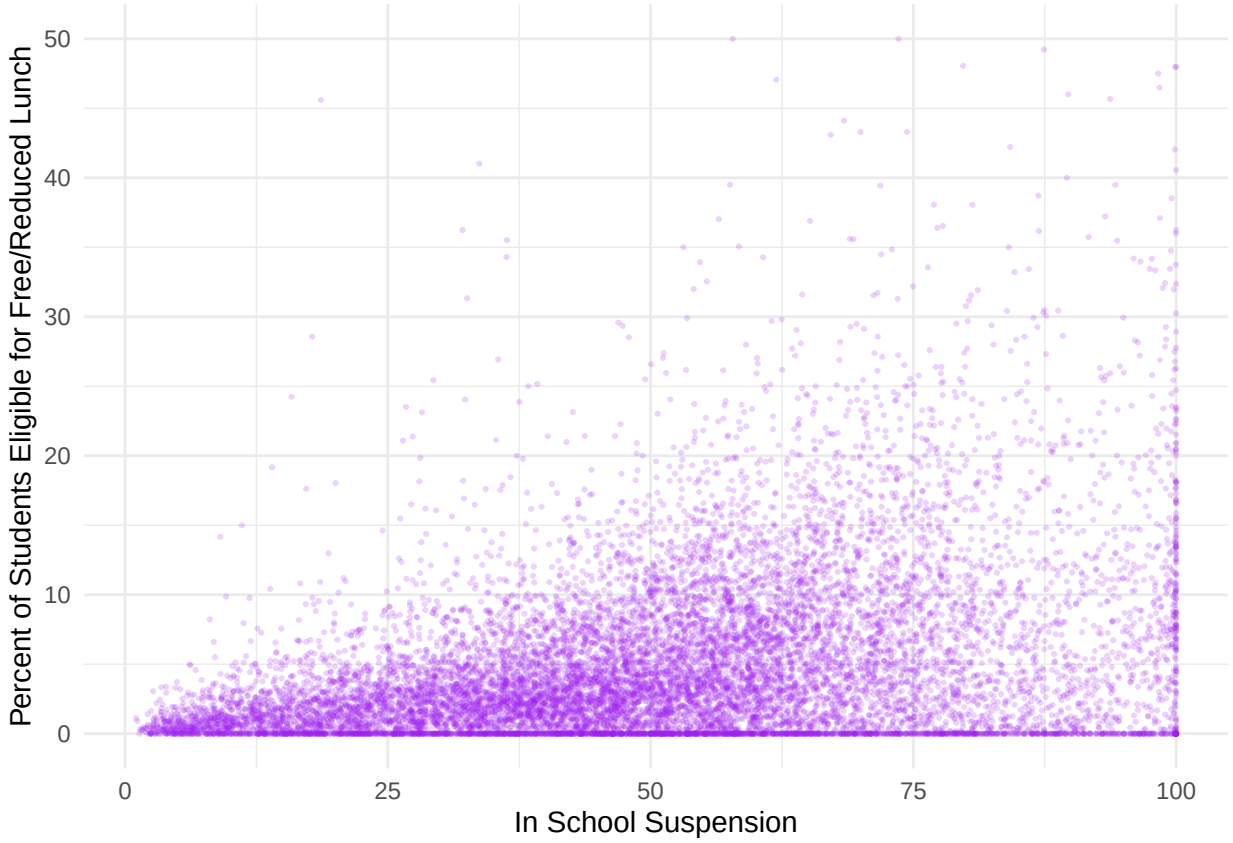
- **`geom_col()`**: Adds a layer to the plot to create a column (or bar) chart.
- **`coord_flip()`**: Flips the coordinates, turning the vertical column chart into a horizontal one. This can make it easier to read state names.
- **`theme(legend.position = "none")`**: Modifies the plot's theme to remove the legend. Since the bars are colored by state and the state names are already on the x-axis, a legend might be redundant.
- **`xlab("")`**: Removes the x-axis label.
- **`ylab("School-Level ISS")`**: Sets the y-axis label to "School-Level ISS".

Scatterplots in GGPlot

The code below is creating a scatter plot where each point represents a school (or another entity). The x-axis of the plot shows the percentage of students eligible for free or reduced lunch, while the y-axis displays the number of in-school suspensions. The points are small, semi-transparent, and purple, and the plot has a clean, minimalistic look. The y-axis is limited to show values between 0 and 50.

```
cr%>%
  ggplot(aes(x=stu_per_frl,y=iss))+
  geom_point(size=.4,alpha=.2,color="purple")+
  ylim(0,50)+
  theme_minimal()+
  xlab("In School Suspension")+
  ylab("Percent of Students Eligible for Free/Reduced Lunch")
```

```
## Warning: Removed 105 rows containing missing values (`geom_point()`).
```



Let's break down this code to understand the visualization it creates:

Starting with the Data: - `cr %>%`: The dataset `cr` is being used as the basis for the visualization.

Initiating the Plot: - `ggplot(aes(x=stu_per_frl, y=iss))`: The `ggplot()` function is used to start the plot creation. The `aes()` function inside it defines the aesthetics or visual properties of the plot. - `x=stu_per_frl`: The x-axis will represent the variable `stu_per_frl`, which might be the percentage of students per school who are eligible for free or reduced lunch. - `y=iss`: The y-axis will represent the variable `iss`, which is the number of in-school suspensions.

Adding Points to the Plot: - `geom_point(size=.4, alpha=.2, color="purple")`: This adds a layer of points to the plot. Each point represents a school (or another relevant entity from the dataset). - `size=.4`: The size of each point is set to 0.4, making them relatively small. - `alpha=.2`: The transparency of each point

is set to 0.2 (on a scale from 0 to 1), making them semi-transparent. This can help in visualizing overlapping points. - `color="purple"`: The color of each point is set to purple.

Setting the Y-axis Limits: - `ylim(0,50)`: This restricts the y-axis to show values between 0 and 50. Any points outside this range won't be displayed.

Applying a Theme: - `theme_minimal()`: This applies a minimalistic theme to the plot, which provides a clean and distraction-free background.

Setting Axis Labels: - `xlab("In School Suspension")`: Renames the x-axis label to "In School Suspension". - `ylab("Percent of Students Eligible for Free/Reduced Lunch")`: Renames the y-axis label to "Percent of Students Eligible for Free/Reduced Lunch".

The `tidymodels` workflow

`tidymodels` is a collection of R packages that provides a consistent and tidy approach to building, evaluating, and tuning statistical models. It integrates various stages of the modeling process, from data preprocessing to model evaluation, under a unified framework that's easy to use and aligns with the principles of the `tidyverse`.

Philosophy: - **Tidy Data Principles**: `tidymodels` is built on the same principles as the `tidyverse`, emphasizing clear, readable, and consistent code. It's designed to work seamlessly with other `tidyverse` packages. - **Unified Interface**: It offers a consistent interface for various modeling tasks, regardless of the underlying model type.

Components: - `recipes`: Provides tools for feature engineering and data preprocessing. It allows you to specify a series of steps to prepare your data for modeling. - `parsnip`: A unified interface for model specification. It allows you to define a model without committing to a specific computational engine. - `workflows`: Combines pre-processing steps from `recipes` and model specifications from `parsnip` into a single object to streamline the modeling process. - `tune`: Tools for model tuning, such as grid search and cross-validation. - `yardstick`: For model evaluation and metric calculation. - `dials`: Helps in creating tuning grids for different model parameters. - `rsample`: Provides infrastructure for data splitting and resampling, which is essential for model validation and testing.

Workflow: - **Data Splitting**: Using `rsample`, you can create training and testing datasets. - **Preprocessing**: With `recipes`, you define a series of preprocessing steps like normalization, encoding categorical variables, and more. - **Model Specification**: Using `parsnip`, you specify the type of model you want (e.g., linear regression, decision tree) without choosing the computational engine (e.g., lm, xgboost). - **Combining Preprocessing and Modeling**: `workflows` lets you combine your preprocessing steps and model specification into a single object. - **Model Training**: Train your model on the training data. - **Model Evaluation**: Once trained, you can evaluate your model's performance on the testing data using `yardstick`. - **Tuning (if needed)**: If your model has hyperparameters, you can use `tune` and `dials` to find the best parameters.

Advantages: - **Flexibility**: Easily switch between different model types or computational engines without drastically changing your code. - **Consistency**: Regardless of the model type, the code structure remains consistent. - **Integration**: Designed to work seamlessly with other `tidyverse` packages, making it easier to integrate modeling into a broader data analysis pipeline.

Below I'll use tidymodel

Data Wrangling

I'm going to select just certain variables from the larger dataset. Luckily this dataset makes use of consistent variable naming patterns that I can use. I'm going to grab my dependent variable of `iss` and every variable that has `stu` for student and `ngh` for neighborhood. I'll also include the categorical variables of urbanicity and state.

```
cr<-cr %>%
  select(iss,
```

```
        contains("stu"),
        contains("ngh"),
        urbanicity,
        state_name)
```

Splitting into training and testing

The code is dividing the `cr` dataset into two parts: a training set (`train`) to teach a machine learning model and a testing set (`test`) to evaluate how well the model has learned. This split ensures that the model is evaluated on data it hasn't seen before, providing a more realistic assessment of its performance.

```
cr_split<-initial_split(cr)

train<-training(cr_split)

test<-testing(cr_split)
```

`cr_split <- initial_split(cr)`: - This line uses the `initial_split` function to divide the `cr` dataset into two parts. By default, `initial_split` typically allocates 75% of the data to the training set and the remaining 25% to the testing set, though this ratio can be adjusted. - The result, `cr_split`, is a special split object that contains information about which rows of the original `cr` dataset belong to the training set and which belong to the testing set.

`train <- training(cr_split)`: - Here, the `training` function extracts the training portion of the data from the `cr_split` object and assigns it to the `train` variable. This dataset will be used to train a machine learning model.

`test <- testing(cr_split)`: - Similarly, the `testing` function extracts the testing portion of the data from the `cr_split` object and assigns it to the `test` variable. This dataset will be used to evaluate the performance of the trained model on unseen data.

Set Model

The code is setting up a linear regression model using the standard linear modeling engine in R.

```
cr_model<-linear_reg(mode="regression",engine="lm")
```

Detailed Explanation:

- **`linear_reg(mode="regression", engine="lm")`**:
  - **`linear_reg()`**: This function is from the `parsnip` package, which is part of the `tidymodels` framework. It's used to specify a linear regression model.
  - **`mode="regression"`**: This argument specifies the type of modeling. In this case, it's regression, which is used to predict a continuous outcome variable based on one or more predictor variables.
  - **`engine="lm"`**: This argument specifies the computational engine to use for the linear regression. The "lm" engine refers to R's built-in `lm()` function for linear modeling.
- **`cr_model`**: The specified linear regression model is then stored in the `cr_model` variable. This doesn't train the model yet; it merely sets up the type of model and the engine to be used. Training the model would require additional steps using the training data.

Set Recipe

In the tidymodels framework, recipes is a package that provides a streamlined way to define and preprocess data for modeling. It allows users to specify a series of steps to transform and preprocess data, such as normalization, encoding categorical variables, and handling missing values. These steps are defined in a consistent and reproducible manner, creating a "recipe" that can be applied to both training and new datasets. This ensures that data transformations are consistent across different stages of the modeling process,

facilitating more reliable model training and prediction. Essentially, recipes offers a tidy and systematic approach to data preparation in the modeling workflow.

The code below is preparing the data for modeling. It sets up a series of steps to process the data, such as handling missing values, converting categorical variables into a format suitable for modeling, and normalizing the data.

```
cr_formula<-as.formula("iss~.")

cr_rec<-recipe(cr_formula,data=train)%>%
  update_role(iss,new_role = "outcome")%>%
  step_other(all_nominal_predictors(),threshold = .01)%>%
  step_dummy(all_nominal_predictors())%>%
  step_filter_missing(all_predictors(),threshold = .1)%>%
  step_naomit(all_outcomes(),all_predictors())%>%
  step_corr(all_predictors(),threshold = .95)%>%
  step_zv(all_predictors())%>%
  step_normalize(all_predictors())
```

The code is preparing the data for modeling. It sets up a series of steps to process the data, such as handling missing values, converting categorical variables into a format suitable for modeling, and normalizing the data.

Detailed Explanation:

1. `cr_formula <- as.formula("iss~.")`:
   - This creates a formula indicating that the variable `iss` is the outcome (or dependent variable) we want to predict, and the `.` means we want to use all other variables in the dataset as predictors (or independent variables).
2. `recipe(cr_formula, data=train)`:
   - The `recipe` function starts the specification of preprocessing steps. It uses the formula and the training data (`train`) as inputs.
3. `update_role(iss, new_role = "outcome")`:
   - This explicitly sets the role of the `iss` variable as the "outcome" or the variable we're trying to predict.
4. `step_other(all_nominal_predictors(), threshold = .01)`:
   - For categorical (nominal) predictors, any categories that constitute less than 1% of the data will be lumped together into a new category, typically called "other".
5. `step_dummy(all_nominal_predictors())`:
   - Converts categorical variables into dummy variables (also known as one-hot encoding). This is necessary because many modeling algorithms require numerical input.
6. `step_filter_missing(all_predictors(), threshold = .1)`:
   - This step removes any predictor variables that have more than 10% missing values.
7. `step_naomit(all_outcomes(), all_predictors())`:
   - Removes rows (observations) from the data where either the outcome or any of the predictor variables have missing values.
8. `step_corr(all_predictors(), threshold = .95)`:
   - Identifies and removes predictor variables that have a correlation higher than 0.95 with any other predictor. This helps in addressing multicollinearity.
9. `step_zv(all_predictors())`:
   - Removes predictor variables that have a zero variance, meaning they have the same value for all observations.
10. `step_normalize(all_predictors())`:

- Normalizes all predictor variables so they have a mean of 0 and a standard deviation of 1. This can be beneficial for certain modeling algorithms.

The result of all these steps is stored in `cr_rec`, which can then be used to preprocess the training data and

any future data in a consistent manner before modeling.

Checking out the results of a recipe

The first code prepares the series of data transformation steps defined in cr_rec but doesn't apply them yet. The second code prepares and then applies these transformations to the train dataset.

```
cr_rec%>%
  prep()
```

```
## Warning in stats::cor(x, use = use, method = method): the standard deviation is
## zero
```

```
## Warning: The correlation matrix has missing values. 2 columns were excluded
## from the filter.
```

```
##
```

```
## -- Recipe ----------------------------------------------------------------
```

```
##
```

```
## -- Inputs
```

```
## Number of variables by role
```

```
## outcome:    1
## predictor: 28
```

```
##
```

```
## -- Training information
```

```
## Training data contained 9952 data points and 9952 incomplete rows.
```

```
##
```

```
## -- Operations
```

```
## * Collapsing factor levels for: urbanicity, state_name | Trained
```

```
## * Dummy variables from: urbanicity, state_name | Trained
```

```
## * Missing value column filter removed: stu_per_ell, ... | Trained
```

```
## * Removing rows with NA values in: iss, stu_per_frl, ... | Trained
```

```
## * Correlation filter on: ngh_per_hisp, ngh_per_black, ... | Trained
```

```
## * Zero variance filter removed: urbanicity_other, state_name_Colorado | Trained
```

```
## * Centering and scaling for: stu_per_frl, stu_per_speced, ... | Trained
```

```
cr_rec%>%
  prep()%>%
   bake(train)
```

```
## Warning in stats::cor(x, use = use, method = method): the standard deviation is
## zero
```

```
## Warning in stats::cor(x, use = use, method = method): The correlation matrix
## has missing values. 2 columns were excluded from the filter.
```

```
## # A tibble: 9,952 x 51
##    stu_per_frl stu_per_speced stu_per_black stu_per_hisp stu_per_white
##          <dbl>          <dbl>         <dbl>        <dbl>         <dbl>
```

```
##  1      1.38            0.532       -0.407        0.374        -0.0866
##  2     -0.486           0.544       -0.375        0.0453        0.368
##  3      0.869          -0.908       -0.217        0.491        -0.0650
##  4     -0.308           0.619        0.0932      -0.137         0.229
##  5     -0.605           0.608       -0.396       -0.601         0.738
##  6      0.0577          1.08        -0.372       -0.371         0.540
##  7      0.664          -0.464       -0.301       -0.577         0.752
##  8     -1.64            0.461       -0.373       -0.581         0.762
##  9     -0.869          -1.40        -0.407       -0.603         0.871
## 10     -1.59           -0.669       -0.314       -0.413         0.301
## # i 9,942 more rows
## # i 46 more variables: stu_per_multi <dbl>, ngh_per_asian <dbl>,
## #   ngh_per_baplus <dbl>, ngh_per_poverty <dbl>, ngh_per_unemp <dbl>,
## #   ngh_per_singlemom <dbl>, ngh_pupilteachratio <dbl>, ngh_per_multi <dbl>,
## #   iss <dbl>, urbanicity_Suburban <dbl>, urbanicity_Town <dbl>,
## #   urbanicity_Urban <dbl>, state_name_Arizona <dbl>,
## #   state_name_Arkansas <dbl>, state_name_California <dbl>, ...
```

1. **`cr_rec %>% prep()`**:
   - **`prep()`**: This function prepares the steps defined in the `cr_rec` recipe. It computes any required statistics or parameters needed for the transformations (e.g., mean and standard deviation for normalization) but doesn't apply the transformations to the data yet. Think of it as getting the recipe ready for cooking but not actually cooking.
2. **`cr_rec %>% prep() %>% bake(train)`**:
   - **`bake()`**: Once the recipe is prepared with `prep()`, the `bake()` function is used to apply the transformations to a dataset. In this case, the transformations are applied to the `train` dataset.
   - Essentially, this sequence of functions means "prepare the transformations and then apply them to the `train` dataset."

By separating the `prep()` and `bake()` steps, `tidymodels` allows for a consistent set of transformations to be easily applied to different datasets, ensuring that both training and testing data undergo the same preprocessing.

Creating a workflow and fitting the model

```
cr_wf<-workflow()%>%
  add_model(cr_model)%>%
  add_recipe(cr_rec)%>%
  fit(train)
```

```
## Warning in stats::cor(x, use = use, method = method): the standard deviation is
## zero
```

```
## Warning: The correlation matrix has missing values. 2 columns were excluded
## from the filter.
```

The code is setting up a workflow that combines both the data preprocessing steps (from `cr_rec`) and the modeling specification (from `cr_model`). It then trains the model using the `train` dataset.

1. **`workflow()`**:
   - This function initializes a workflow. In the `tidymodels` framework, a workflow is a way to bundle together preprocessing steps (like those defined in a recipe) and a model specification.
2. **`add_model(cr_model)`**:
   - This adds the model specification from `cr_model` (which was defined earlier) to the workflow. It tells the workflow what kind of model we're planning to train.
3. **`add_recipe(cr_rec)`**:

- This adds the data preprocessing steps from `cr_rec` (which were defined earlier) to the workflow. It tells the workflow how to preprocess the data before training the model.

4. `fit(train)`:
   - Once the workflow has both the model and the recipe, the `fit()` function is used to train the model using the `train` dataset. This involves applying the preprocessing steps to the training data and then training the model on the processed data.

The result is stored in `cr_wf`. This object now contains the trained model along with all the preprocessing steps, making it ready for predictions on new data.

Testing predictions in the testing split

```
test<-
  cr_wf%>%
  predict(new_data=test)%>%
  bind_cols(test)
```

In Simple Terms: The code calculates the Root Mean Squared Error (RMSE) for the predictions in the `test` dataset by comparing the predicted values (`.pred`) with the actual values (`iss`).

Detailed Explanation:

1. **`test %>%`**:
   - This takes the `test` dataset, which now contains both the actual values (`iss`) and the predicted values (`.pred`), as the starting point for the subsequent operations.
2. **`rmse(truth="iss", estimate=.pred)`**:
   - The `rmse()` function from the `yardstick` package (part of `tidymodels`) is used to calculate the RMSE.
   - **`truth="iss"`**: Specifies that the actual values are in the `iss` column of the `test` dataset.
   - **`estimate=.pred`**: Specifies that the predicted values are in the `.pred` column of the `test` dataset.

The result will be the RMSE value, which provides a measure of the differences between the predicted and actual values. A lower RMSE indicates a better fit of the model to the data, while a higher RMSE suggests a poorer fit.

Note: Ensure that the `yardstick` package is loaded to use the `rmse()` function.

```
test%>%
  rmse(truth="iss",estimate=.pred)
```

```
## # A tibble: 1 x 3
##   .metric .estimator .estimate
##   <chr>   <chr>          <dbl>
## 1 rmse    standard        4.91
```

In Simple Terms: The code calculates the Root Mean Squared Error (RMSE) for the predictions in the `test` dataset by comparing the predicted values (`.pred`) with the actual values (`iss`).

Detailed Explanation:

1. **`test %>%`**:
   - This takes the `test` dataset, which now contains both the actual values (`iss`) and the predicted values (`.pred`), as the starting point for the subsequent operations.
2. **`rmse(truth="iss", estimate=.pred)`**:
   - The `rmse()` function from the `yardstick` package (part of `tidymodels`) is used to calculate the RMSE.
   - **`truth="iss"`**: Specifies that the actual values are in the `iss` column of the `test` dataset.
   - **`estimate=.pred`**: Specifies that the predicted values are in the `.pred` column of the `test` dataset.

The result will be the RMSE value, which provides a measure of the differences between the predicted and actual values. A lower RMSE indicates a better fit of the model to the data, while a higher RMSE suggests a poorer fit.

Note: Ensure that the `yardstick` package is loaded to use the `rmse()` function.

Examining Coefficients

Coefficients? Who cares about coefficients? We've already got an rmse!

The code fits the model above to the full dataset, extracts the coefficients from the trained model from the workflow, organizes these results in a tidy format, sorts them based on the magnitude of the estimates (from highest to lowest), and then displays the top 100 results.

```
cr_wf_full<-
  cr_wf%>%
  fit(cr)
```

```
## Warning in stats::cor(x, use = use, method = method): the standard deviation is
## zero
```

```
## Warning: The correlation matrix has missing values. 2 columns were excluded
## from the filter.
```

```
cr_wf_full%>%
  extract_fit_parsnip()%>%
  tidy()%>%
  arrange(-estimate)%>%
  print(n=100)
```

```
## # A tibble: 51 x 5
##    term                    estimate std.error statistic  p.value
##    <chr>                      <dbl>     <dbl>     <dbl>    <dbl>
##  1 (Intercept)                5.15      0.0444   116.    0
##  2 stu_per_black              2.11      0.234      9.01  2.47e-19
##  3 stu_per_white              2.11      0.404      5.21  1.92e- 7
##  4 stu_per_multi              1.17      0.196      5.96  2.66e- 9
##  5 state_name_Texas           1.08      0.135      7.97  1.70e-15
##  6 stu_per_frl                1.02      0.0934    11.0   8.33e-28
##  7 stu_per_hisp               1.02      0.312      3.26  1.10e- 3
##  8 ngh_per_asian              0.464     0.0974     4.76  1.96e- 6
##  9 state_name_Missouri        0.295     0.106      2.79  5.35e- 3
## 10 state_name_Arkansas        0.287     0.0802     3.58  3.51e- 4
## 11 ngh_per_singlemom          0.242     0.0849     2.85  4.42e- 3
## 12 urbanicity_Town            0.218     0.0496     4.39  1.14e- 5
## 13 state_name_Georgia         0.215     0.0693     3.11  1.88e- 3
## 14 stu_per_speced             0.146     0.0552     2.64  8.19e- 3
## 15 state_name_Kentucky        0.124     0.0694     1.79  7.35e- 2
## 16 state_name_Arizona         0.122     0.0842     1.45  1.46e- 1
## 17 ngh_per_unemp              0.0543    0.0664     0.818 4.13e- 1
## 18 state_name_Virginia        0.0535    0.0647     0.827 4.08e- 1
## 19 state_name_Mississippi     0.0490    0.0645     0.760 4.47e- 1
## 20 ngh_per_poverty            0.0184    0.0768     0.239 8.11e- 1
## 21 state_name_Tennessee      -0.0125    0.0642    -0.194 8.46e- 1
## 22 state_name_Oregon         -0.0395    0.0804    -0.491 6.23e- 1
## 23 urbanicity_Suburban       -0.0419    0.0642    -0.653 5.14e- 1
## 24 ngh_per_multi             -0.104     0.0458    -2.28  2.27e- 2
```

```
## 25 state_name_Washington     -0.111     0.0904    -1.23  2.20e- 1
## 26 urbanicity_Urban          -0.118     0.0544    -2.16  3.05e- 2
## 27 state_name_Connecticut    -0.186     0.0726    -2.57  1.03e- 2
## 28 state_name_Indiana        -0.207     0.0824    -2.51  1.21e- 2
## 29 state_name_New.Hampshire  -0.224     0.0719    -3.12  1.84e- 3
## 30 state_name_Oklahoma       -0.230     0.105     -2.19  2.83e- 2
## 31 state_name_South.Dakota   -0.235     0.0702    -3.34  8.32e- 4
## 32 state_name_Montana        -0.272     0.0812    -3.34  8.33e- 4
## 33 state_name_other          -0.279     0.119     -2.34  1.91e- 2
## 34 state_name_Minnesota      -0.320     0.0855    -3.74  1.85e- 4
## 35 state_name_Michigan       -0.328     0.105     -3.14  1.72e- 3
## 36 state_name_Nebraska       -0.352     0.0819    -4.30  1.73e- 5
## 37 state_name_Kansas         -0.374     0.0853    -4.39  1.16e- 5
## 38 state_name_Iowa           -0.451     0.0892    -5.06  4.35e- 7
## 39 state_name_California     -0.458     0.147     -3.11  1.87e- 3
## 40 state_name_New.York       -0.470     0.123     -3.83  1.29e- 4
## 41 state_name_North.Dakota   -0.479     0.0790    -6.06  1.41e- 9
## 42 state_name_Wisconsin      -0.494     0.0934    -5.29  1.24e- 7
## 43 state_name_Massachusetts  -0.498     0.0868    -5.74  9.85e- 9
## 44 ngh_per_baplus            -0.523     0.0781    -6.70  2.23e-11
## 45 state_name_Ohio           -0.563     0.107     -5.27  1.39e- 7
## 46 state_name_Maine          -0.576     0.0770    -7.48  7.86e-14
## 47 state_name_Illinois       -0.579     0.117     -4.95  7.35e- 7
## 48 state_name_Vermont        -0.585     0.0868    -6.73  1.72e-11
## 49 state_name_Pennsylvania   -0.602     0.106     -5.69  1.33e- 8
## 50 state_name_New.Jersey     -0.760     0.116     -6.56  5.58e-11
## 51 ngh_pupilteachratio       -0.978     0.179     -5.48  4.40e- 8
```

Detailed Explanation:

1. `cr_wf %>%`:
   - This takes the trained workflow `cr_wf` as the starting point for the subsequent operations.
2. `extract_fit_parsnip()`:
   - This function extracts the results of the trained model from the workflow. In the context of a linear regression model (as indicated by previous interactions), this would typically include coefficients for each predictor variable.
3. `tidy()`:
   - This function, from the `broom` package, tidies the results of the model, converting them into a clean and standardized data frame format. For a linear regression model, this would typically result in a data frame with columns like `term` (the predictor variable name), `estimate` (the coefficient value), and others like `std.error`, `statistic`, and `p.value`.
4. `arrange(-estimate)`:
   - This arranges (or sorts) the results based on the `estimate` column in descending order (from highest to lowest coefficient value).
5. `print(n=100)`:
   - This displays the top 100 results after sorting. It's especially useful if there are many predictor variables in the model and you want to see the ones with the highest coefficient values.

Overall, this code is useful for understanding the impact and significance of different predictor variables in the trained model. The sorted list can give insights into which variables have the most substantial positive or negative effects on the outcome.

Note: Ensure that the `broom` package is loaded to use the `tidy()` function.

Moving forward

We'll next use lasso, ridge and elastic net to select the features that contribute the most to the model, resulting in a model that should be able to better predict the outcome in the testing data.