Data Analysis in R: A Basic Guide

Antonio Solorio

2024-07-15

Contents

Introduction

Welcome to "Data Analysis in R: A Basic Guide." This book aims to provide you with a basic foundation in data analysis using R, a powerful and versatile programming language. Throughout this book, you will learn various techniques and tools essential for effective data analysis.

To illustrate these concepts, we will use the Home Mortgage Disclosure Act (HMDA) data as a practical example. This real-world dataset will help you understand how to apply data analysis methods in a meaningful context.

Chapter 1

What is HMDA Data?

The Home Mortgage Disclosure Act (HMDA) was enacted by Congress in 1975 and is implemented by the Consumer Financial Protection Bureau (CFPB). The HMDA requires many financial institutions to maintain, report, and publicly disclose information about mortgages. This information is crucial for understanding and monitoring trends in housing finance, and for ensuring compliance with fair lending laws. [1]

HMDA data includes information on loan applications, loan originations, loan purchases, and denied applications. The data encompasses various aspects such as:

- **Loan Characteristics**: Information about the loan amount, type of loan, and purpose of the loan (e.g., home purchase, refinance).
- **Applicant Information**: Demographic details of the loan applicants including race, ethnicity, gender, and income.
- **Property Information**: Data about the location and type of property being financed.
- **Action Taken**: The outcome of the loan application, whether it was approved, denied, or withdrawn.

1.1 Why Use HMDA Data?

While this book is focused on teaching data analysis in R, the HMDA dataset serves as an excellent example for several reasons:

1. **Real-World Relevance**: HMDA data provides a real-world context that makes the learning process more engaging and practical.

[1] If you would like to learn more about HMDA data please see: https://www.consumerfinance.gov/data-research/hmda/

2. **Comprehensive Dataset**: The dataset includes a wide range of variables, making it suitable for demonstrating various data analysis techniques.
3. **Publicly Available**: HMDA data is publicly accessible, allowing you to follow along with the examples and practice on your own.

By the end of this book, you will not only have a solid understanding of data analysis in R but also be equipped with practical skills that can be applied to other datasets and domains.

Let's get started on this journey of exploring data analysis with R, using the HMDA data as our guide!

Chapter 2

Data Importing

In this chapter, we will explore the process of importing data into R for analysis. Data import is a crucial step in the data analysis workflow, as it allows you to load external data into R for further processing and analysis. We will focus on importing data from CSV files, which are one of the most common data formats used in data analysis. We will also discuss common issues encountered during data import and how to handle them, and how to handle the importation of large datasets in chunks.

2.1 Different Types of Data

In the realm of data analysis, you will encounter various types of data formats. Here are some common ones:

- **Text Files**: Unstructured text data that can be read line by line or in blocks, and which may be delimited by specific characters.
- **CSV (Comma-Separated Values)**: A CSV file is a type of text file that is delimited by commas. It is one of the most common data formats used for storing tabular data.
- **Excel Files**: Commonly used spreadsheets saved in formats like `.xlsx` or `.xls`.
- **JSON (JavaScript Object Notation)**: A lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate.
- **SQL Databases**: Structured data stored in relational databases, which can be queried using SQL (Structured Query Language).
- **API Data**: Data fetched from web APIs, which often come in formats like JSON or XML.

2.1.1 Why Start with CSV Files?

We will start with CSV files for several reasons:

1. **Simplicity**: CSV files are easy to understand and work with, making them ideal for beginners.
2. **Ubiquity**: CSV is one of the most common data formats, widely supported by various applications and programming languages.
3. **Ease of Use in R**: R provides straightforward functions for importing and handling CSV files, making it an excellent starting point for learning data import techniques.

By mastering the import of CSV files, you'll build a strong foundation that will make it easier to work with other data formats as you progress in your data analysis journey.

2.2 Downloading and Importing HMDA Data in CSV Format

To practice importing CSV files in R, we will use the Home Mortgage Disclosure Act (HMDA) data in CSV format. This data can be found at the Consumer Financial Protection Bureau (CFPB) website. In particular we will be working with the Snapshot National Loan Level Dataset, specifically for that in 2022 for Nevada.

2.2.1 Snapshot National Loan Level Dataset

The Snapshot files contain the national HMDA datasets as of May 1, 2023 for all HMDA reporters, as modified by the Bureau to protect applicant and borrower privacy. The snapshot files are available to download in both .csv and pipe delimited text file formats at the following link: https://ffiec.cfpb.gov/data-publication/snapshot-national-loan-level-dataset/. One of the issues with these files however is that they are quite large, so we will be working with a subset of the data for Nevada in 2022.

The subset of the data for Nevada in 2022 can be downloaded from the following link: Nevada 2022 HMDA Data.

2.3 Importing CSV Files in R

R provides several functions for importing CSV files. The most commonly used function is `read.csv()`, which is part of the base R package. Additionally, the

readr package offers the **read_csv()** function, which is optimized for faster performance and easier handling of large datasets.

2.3.1 Using read.csv()

The **read.csv()** function is straightforward to use. It is actually a special case of the more general **read.table()** function, with default parameters set for reading CSV files Here's how you can import a CSV file using this function:

```r
# Importing a CSV file using read.csv()
data <- read.csv("downloads/state_NV.csv")

# Display the first few rows of the data
head(data)
```

In this example, replace **"downloads/state_NV.csv"** with the actual path to your CSV file. The **head()** function is used to display the first few rows of the imported data.

Details on read.csv()

The **read.csv()** function is a simplified wrapper around **read.table()**, with pre-set arguments tailored for reading comma-separated files. Specifically, it sets the following default arguments:

- **sep = ","** sets the field separator to a comma.
- **header = TRUE** indicates that the first line of the file contains column names.
- **stringsAsFactors = default.stringsAsFactors()** specifies whether character vectors should be converted to factors (default behavior depends on the R version).

Here's an equivalent way to use `read.table()` to achieve the same result as `read.csv()`:

```r
# Importing a CSV file using read.table()
data <- read.table("downloads/state_NV.csv", sep = ",", header = TRUE, stringsAsFactors = FALSE)

# Display the first few rows of the data
head(data)
```

As you can see, read.csv() simplifies the process by encapsulating these common settings, making it easier and quicker to read CSV files.

2.3.2 Using `read_csv()` from the `readr` Package

The **`readr`** package provides a faster and more convenient way to import CSV files with the **`read_csv()`** function. First, you need to install and load the **`readr`** package:

```
# Install the readr package
install.packages("readr")
```

Once the package is installed, you can use the **`read_csv()`** function to import the CSV file:

```
# Load the readr package
library(readr)

# Importing a CSV file using read_csv()
data <- read_csv("downloads/state_NV.csv")

# Display the first few rows of the data
head(data,50)
```

Similar to **`read.csv()`**, replace **`"downloads/state_NV.csv"`** with the actual path to your CSV file. The **`read_csv()`** function also automatically parses the data types of the columns, which can save you time and effort, you need to be carefull as sometimes read_csv() may guess the column type wrong!

Details on `read_csv()`

The **`read_csv()`** function is a special case of the more general **`read_delim()`** function from the **`readr`** package, with default parameters set for reading comma-separated files. Specifically, it sets the following default arguments:

- **`delim = ","`** sets the field separator to a comma.
- **`col_types = cols()`** automatically detects the data types of columns unless specified otherwise.
- **`trim_ws = TRUE`** indicates that whitespace should be trimmed from the beginning and end of each field.

These defaults make **`read_csv()`** particularly convenient for reading CSV files without needing to manually specify these common options.

Here's an equivalent way to use **`read_delim()`** to achieve the same result as **`read_csv()`**:

```r
# Importing a CSV file using read_delim()
data <- read_delim(
  "downloads/state_NV.csv",
  delim = ",",
  col_types = cols(),
  trim_ws = TRUE
)

# Display the first few rows of the data
head(data, 50)
```

As you can see, **read_csv()** simplifies the process by encapsulating these common settings, making it easier and quicker to read CSV files.

2.3.2.1 Handling Parsing Issues

If you been following along, when you ran **`data <- read_csv("downloads/state_NV.csv")`** you have probably encountered a warning similar to:

```
```{r}
data <- read_csv("C:\\Users\\anton\\Desktop\\Code Projects\\Using R to Work with Data\\downloads\\state_NV.csv")
```

Warning: One or more parsing issues, call `problems()` on your data frame for details, e.g.:
  dat <- vroom(...)
  problems(dat)
Rows: 180204 Columns: 99
── Column specification ──
Delimiter: ","
chr (13): lei, state_code, conforming_loan_limit, derived_loan_product_type, derived_dwelling_category, derived_ethnicity, derived_ra...
dbl (76): activity_year, derived_msa-md, county_code, census_tract, action_taken, purchaser_type, preapproval, loan_type, loan_purpos...
lgl (10): multifamily_affordable_units, applicant_ethnicity-3, applicant_ethnicity-4, applicant_ethnicity-5, co-applicant_ethnicity-3...

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.
```

The warning is letting us know that **`read_csv()`** ran into some parsing issues, and its recommending that we run **`problems()`** to see what the issues are. Let's run **`problems()`** to see what the issues are:

```r
# Display the problems encountered during parsing
problems(data)
```

A tibble: 3,559 x 5

row <int>	col <int>	expected <chr>	actual <chr>	file <chr>
59	44	a double	>149	...downloads/state_NV.csv
60	44	a double	>149	...downloads/state_NV.csv
61	44	a double	>149	...downloads/state_NV.csv
62	44	a double	>149	...downloads/state_NV.csv
63	44	a double	>149	...downloads/state_NV.csv
226	66	1/0/T/F/TRUE/FALSE	41	...downloads/state_NV.csv
262	71	1/0/T/F/TRUE/FALSE	41	...downloads/state_NV.csv
268	52	1/0/T/F/TRUE/FALSE	14	...downloads/state_NV.csv
282	52	1/0/T/F/TRUE/FALSE	12	...downloads/state_NV.csv
282	57	1/0/T/F/TRUE/FALSE	14	...downloads/state_NV.csv

1-10 of 3,559 rows   Previous 1 2 3 4 5 6 ... 100 Next

The **`problems()`** function displays the issues encountered during parsing. The **'row'** column indicates the row number where the issue occurred, and the **'col'** column indicates the column number. The **'expected'** column shows the expected data type, and the **'actual'** column shows the actual value.

In the example above in row 59, column 44 expected a double but the cell contains **">149"**, which is a character type. A double is a numeric data type that can represent decimal numbers, while a character is a text data type. The issue here is that **`read_csv()`** expected a double but found a character. Even though you can't see it in the table above Column 44 is the **total_units** column, which should contain the total number of units for the property. The value **">149"** lets us know that the property has more than 149 units. Therefore the correct column type should be character and not double.

There are two main ways to handle parsing issues in `read_csv()`:

- **Manually Specify Column Types**: You can manually specify the column types using the `col_types` argument in **`read_csv()`**. This approach is useful when you know the data types of the columns in advance, but might be cumbersome for large datasets with many columns.
- **Increase the `guess_max` Argument**: You can increase the `guess_max` argument in `read_csv()` to allow the function to guess the column types for a larger number of rows. This approach isn't perfect, but this way you can avoid having to manually specify the column types.

Below is a code example to manually specify the column types:

```
# Manually specify the column types
data <- read_csv(
  "downloads/state_NV.csv",
  col_types = cols(
    loan_amount = col_double(),
    total_units = col_character(),
    .default = col_character(),
  ),
  na = c("", "NA") # This is to specify what is considered a missing value
  )
```

In this code, we manually specify the column types for the **`loan_amount`** and **`total_units`** columns. We also set the default column type to **`col_character()`** to ensure that all other columns are treated as character columns. The **`na`** argument specifies the values that should be treated as missing values, which in this case we have set to an empty string and **"NA"**.

It's also possible to set the **`guess_max`** argument to a higher value to allow **`read_csv()`** to guess the column types for a larger number of rows. This can be useful when you have a large dataset and want to avoid manually specifying the column types. You can even set it to **`Inf`** to allow **`read_csv()`** to guess the column types by using all the rows in the dataset.

```r
# Increase the guess_max argument
data <- read_csv("downloads/state_NV.csv", guess_max = Inf)
```

2.3.3 Handling File Paths

When specifying the path to your CSV file, it's important to ensure that the path is correct. You can use absolute paths or relative paths. Here are some examples:

- **Absolute Path**: An absolute path specifies the complete path from the root directory. For example, on Windows: `"C:/Users/YourName/Documents/data.csv"`, or on macOS/Linux: `"/Users/YourName/Documents/data.csv"`.
- **Relative Path**: A relative path specifies the path relative to your current working directory. For example, if your current working directory is `"C:/Users/YourName/Documents"`, you can use `"data.csv"`.

You can check your current working directory in R using the `getwd()` function:

```r
# Get the current working directory
getwd()
```

You can also set the working directory using the `setwd()` function:

```r
# Set the working directory
setwd("path/to/your/directory")
```

2.3.4 Common Issues and Solutions

- **File Not Found Error**: Ensure the file path is correct and the file exists at the specified location.
- **Incorrect Data Parsing**: If columns are not parsed correctly, you can specify the column types manually using the `col_types` argument in `read_csv()`.
- **Missing Values**: R automatically handles missing values as `NA`. You can customize the handling of missing values using the `na` argument.

By understanding how to import CSV files in R, you can easily load your data and start your data analysis process. In the next sections, we will explore how to clean and manipulate the imported data to prepare it for analysis.

2.4 Importing Data in Chunks

When working with large datasets, it's often necessary to import data in chunks, especially when the dataset is too large to fit into memory or cannot be opened by standard software like Excel. The readr package in R provides a solution with the **read_delim_chunked()** function, which allows for reading a delimited file in manageable chunks.

The **read_delim_chunked()** function operates similarly to **read_delim()**, but it processes data in smaller portions, making it easier to handle large datasets. A practical approach is to use a callback function to filter data as each chunk is processed.

Here's an example demonstrating how to import a delimited file in chunks and apply a callback function to filter the data for state_code "NV":

```r
# Load the readr package
library(readr)
library(dplyr)

# Define the callback function to filter data for state_code "NV"
filter_data <- function(data_chunk, pos) {
  # Filters data chunk for only rows where state_code == "NV"
  data_chunk <- data_chunk%>%filter(state_code == "NV")
}

# Import a CSV file in chunks using read_csv_chunked()
chunked_data <- read_delim_chunked(
  "downloads/2023_combined_mlar_header.txt", # specify the path to the CSV file
  callback = DataFrameCallback$new(filter_data), # specify the callback function
  chunk_size = 10000, # specify the chunk size,
  delim = "|",
  escape_double = FALSE,
  trim_ws = TRUE,
  col_names = TRUE,
  col_types = cols(.default = col_character())
)
```

In this example:

- **"downloads/2023_combined_mlar_header.txt"** should be replaced with the actual path to your delimited file.
- **delim = "|"** specifies the delimiter used in the file.
- **escape_double = FALSE** specifies whether double quotes should be escaped.

- **`trim_ws = TRUE`** indicates that whitespace should be trimmed from the beginning and end of each field.
- **`col_names = TRUE`** specifies that the file contains column names.
- **`col_types = cols(.default = col_character())`** sets all columns to be read as character data types.

The **`filter_data()`** function is used as a callback to filter the data for the state code "NV". A callback function is a function passed as an argument to another function, which is then executed within that function. Here, the **`filter_data()`** function is applied to each chunk of data read by **`read_delim_chunked()`**, enabling the filtering of data for the state code "NV" as it is read in chunks.

Chapter 3

Data Exploration and Cleaning

Once you have data loaded into your R environment, now comes one of the most important parts of the data processing stage, data exploration and cleaning.

Data Exploration

Data exploration is the initial step in data analysis, where you get a sense of the structure, contents, and characteristics of the dataset. This step involves:

- **Understanding the Dataset:** Reviewing the dataset to understand its structure, the types of data it contains, and the relationships between different variables.
- **Summary Statistics:** Calculating basic statistics such as mean, median, standard deviation, and percentiles to understand the distribution and spread of the data.
- **Visualization** Creating visual representations of the data, such as histograms, box plots, scatter plots, and correlation matrices, to identify patterns, trends, and outliers.
- **Identifying Data Types** Checking the data types of each column to ensure they are as expected (e.g., numerical, categorical, date/time).
- **Detecting Anomalies** Identifying any anomalies, such as missing values, outliers, or inconsistencies that might need to be addressed.

Data Cleaning

Data cleaning, also known as data cleansing or scrubbing, involves correcting or removing inaccuracies and inconsistencies in the data to improve its quality. Key steps include:

- **Handling Missing Values** Dealing with missing data by either removing rows/columns with missing values, imputing missing values using statistical methods, or using algorithms that can handle missing data.

- **Removing Duplicates** Identifying and removing duplicate entries to ensure each record is unique.

- **Correcting Errors** Fixing errors such as typos, incorrect data entries, and inconsistent formatting.

- **Data Transformation** Converting data into the appropriate format or structure, such as normalizing or standardizing numerical data, encoding categorical variables, and creating new derived features.

- **Outlier Treatment** Identifying and handling outliers, which may involve removing them or transforming them to reduce their impact.

- **Consistent Formatting** Ensuring consistent formatting across the dataset, such as consistent date formats, uniform case for text data, and standardized units for numerical data.

Importance of Data Exploration and Cleaning

- **Improves Data Quality:** Ensures the data is accurate, complete, and reliable, which is essential for drawing valid conclusions and making accurate predictions.
- **Enhances Analysis:** Clean and well-understood data allows for more effective and insightful analysis.
- **Reduces Errors:** Minimizes the risk of errors and biases in the data, leading to more robust and trustworthy results.
- **Facilitates Model Building:** Prepares the data in a way that is suitable for building machine learning models, improving their performance and reliability.

Overall, data exploration and cleaning are foundational steps that set the stage for successful data analysis and machine learning projects. In this chapter we will go over some of the most common ways to both explore and clean data.

3.1 Exploring and Cleaning the Data Structure

In this section we will utilize the HMDA Snapshot data for 2022 in Nevada to practice data structure exploration and cleaning. The data is available at the following link: https://ffiec.cfpb.gov/v2/data-browser-api/view/csv?states=NV&years=2022. We have downloaded the data and read it into R using the following:

When we

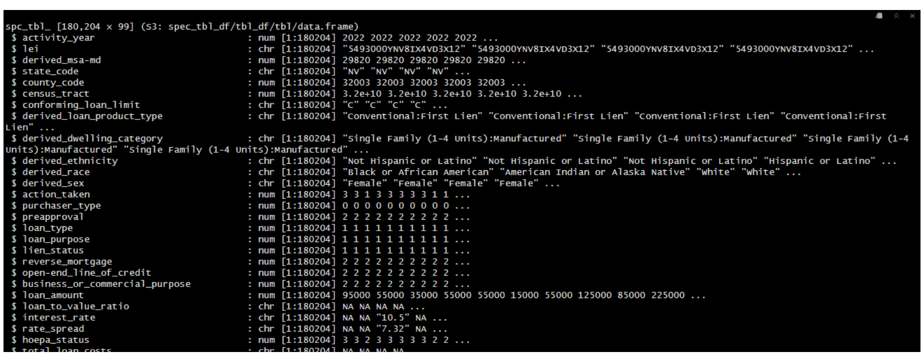
```r
# Load the data
hmda_data <- read_csv("downloads/state_NV.csv", guess_max = Inf)
```

Where `"downloads/state_NV.csv"` would be the path to the downloaded dataset.

3.1.1 Exploring Data Structure

One of the first things we should do is to take a look at the structure of the data. This will help us understand the variables and their types. We can do this using the following code:

```r
# Display the structure of the data
str(hmda_data)
```



```
spc_tbl_ [180,204 x 99] (S3: spec_tbl_df/tbl_df/tbl/data.frame)
 $ activity_year                    : num [1:180204] 2022 2022 2022 2022 2022 ...
 $ lei                              : chr [1:180204] "5493000YNV8IX4VD3X12" "5493000YNV8IX4VD3X12" "5493000YNV8IX4VD3X12" "5493000YNV8IX4VD3X12" ...
 $ derived_msa-md                   : num [1:180204] 29820 29820 29820 29820 29820 ...
 $ state_code                       : chr [1:180204] "NV" "NV" "NV" "NV" ...
 $ county_code                      : num [1:180204] 32003 32003 32003 32003 32003 ...
 $ census_tract                     : num [1:180204] 3.2e+10 3.2e+10 3.2e+10 3.2e+10 3.2e+10 ...
 $ conforming_loan_limit            : chr [1:180204] "C" "C" "C" "C" ...
 $ derived_loan_product_type        : chr [1:180204] "Conventional:First Lien" "Conventional:First Lien" "Conventional:First Lien" "Conventional:First
Lien" ...
 $ derived_dwelling_category        : chr [1:180204] "Single Family (1-4 Units):Manufactured" "Single Family (1-4 Units):Manufactured" "Single Family (1-4
Units):Manufactured" "Single Family (1-4 Units):Manufactured" ...
 $ derived_ethnicity                : chr [1:180204] "Not Hispanic or Latino" "Not Hispanic or Latino" "Not Hispanic or Latino" "Hispanic or Latino" ...
 $ derived_race                     : chr [1:180204] "Black or African American" "American Indian or Alaska Native" "White" "White" ...
 $ derived_sex                      : chr [1:180204] "Female" "Female" "Female" "Female" ...
 $ action_taken                     : num [1:180204] 3 3 1 3 3 3 3 3 1 1 ...
 $ purchaser_type                   : num [1:180204] 0 0 0 0 0 0 0 0 0 0 ...
 $ preapproval                      : num [1:180204] 2 2 2 2 2 2 2 2 2 2 ...
 $ loan_type                        : num [1:180204] 1 1 1 1 1 1 1 1 1 1 ...
 $ loan_purpose                     : num [1:180204] 1 1 1 1 1 1 1 1 1 1 ...
 $ lien_status                      : num [1:180204] 1 1 1 1 1 1 1 1 1 1 ...
 $ reverse_mortgage                 : num [1:180204] 2 2 2 2 2 2 2 2 2 2 ...
 $ open-end_line_of_credit          : num [1:180204] 2 2 2 2 2 2 2 2 2 2 ...
 $ business_or_commercial_purpose   : num [1:180204] 2 2 2 2 2 2 2 2 2 2 ...
 $ loan_amount                      : num [1:180204] 95000 55000 35000 55000 55000 15000 55000 125000 85000 225000 ...
 $ loan_to_value_ratio              : chr [1:180204] NA NA NA NA ...
 $ interest_rate                    : chr [1:180204] NA NA "10.5" NA ...
 $ rate_spread                      : chr [1:180204] NA NA "7.32" NA ...
 $ hoepa_status                     : num [1:180204] 3 3 2 3 3 3 3 3 2 2 ...
```

The **str()** function provides a summary of the data frame, including the number of observations and variables, the names of the variables, and the type of each variable. This information is useful for understanding the structure of the data and planning the analysis. In the attached image of the output above, we can see that the data frame has 180204 observations and 99 variables. We can also see that a couple of the columns got assigned incorrect data types by `read_csv()`, one of these being **county_code** which represents the Federal Information Processing Standards (FIPS) code for the county.

3.1.2 Changing Data Types

As we saw in the previous section, some of the columns were assigned incorrect data types by **read_csv()**. We can fix this by changing the data types of the columns using the **mutate()** function from the **dplyr** package. The **dplyr** package provides a set of functions for data manipulation, and the **mutate()** function is used to create new columns or modify existing columns.[1] Below we utilize the **mutate()** function to change the data type of the **county_code** column to character:

```
# Change the data types of the columns
hmda_data <- hmda_data %>%
  mutate(county_code = as.character(county_code))
```

In the code above, we used the **mutate()** function to change the data type of the **county_code** column to character. **as.character()** is a function that converts the input to a character type, there are other functions like **as.numeric()** and **as.factor()** that can be used to convert the input to numeric and factor types respectively.

3.1.3 Using across() to Change Data Types for Multiple Columns

When you need to change the data types of multiple columns simultaneously, the **across()** function in **dplyr** can be particularly useful. The **across()** function allows you to apply a function to multiple columns in a **mutate()** call.

For example, if you want to change the data types of the **census_tract**, **action_taken**, **loan_type**, and loan_purpose columns to character, you can use the **across()** function as follows:

```
# Change the data types of multiple columns to character
hmda_data <- hmda_data %>%
  mutate(across(c(census_tract, action_taken, loan_type, loan_purpose), as.character))
```

In this code:

- **across(c(census_tract, action_taken, loan_type, loan_purpose), as.character)** applies the **as.character()** function to each of the columns listed inside the **across()** function.
- This approach makes the code more concise and easier to read, especially when dealing with multiple columns.

[1] You can learn more about the **dplyr** package at: https://dplyr.tidyverse.org/

By using `across()`, you can efficiently change the data types of multiple columns in one step, ensuring that your data is properly formatted for subsequent analysis.

3.2 Spotting and Handling NA Values

Missing values, represented as `NA` in R, are a common occurrence in datasets and can significantly impact the results of your data analysis. Therefore, identifying and understanding the extent of missing values is a crucial step in data exploration.

Understanding NA Values

In R, `NA` (Not Available) is used to represent missing or undefined data. Missing values can arise due to various reasons such as data entry errors, data collection issues, or intentional omissions.

In the HMDA data that we have been working with, all of the reasons above apply. Sometimes financial institutions make errors when submitting the data, they are unable to collect the data for one reason or another, or certain data fields don't apply to certain loan applications.

Why Spotting NA Values is Important

- **Data Integrity:** Missing values can lead to incorrect conclusions if not handled properly.
- **Analysis Readiness:** Many statistical and machine learning methods cannot handle missing values directly.
- **Decision Making:** Identifying the pattern and extent of missing values can inform your strategy for handling them (e.g., imputation, removal).

3.2.1 Common Ways to Spot NA Values

Identifying missing values is a critical part of data exploration. Here are some common ways to spot NA values using the `hmda_data` dataset that we loaded in.

Checking for Any NA Values

You can use the `anyNA()` function to check if there are any NA values in the entire dataset.

```r
# Check if there are any NA values in the entire dataset
any_na <- anyNA(hmda_data)
print(any_na) # Returns TRUE if there are any NA values, otherwise FALSE
```

Counting NA Values Per Column

To understand which columns contain NA values and how many, you can use **`colSums(is.na())`**.

```r
# Count the number of NA values in each column
na_per_column <- colSums(is.na(hmda_data)) %>%
  as.data.frame()
```

Description: df [99 x 1]

	. <dbl>
activity_year	0
lei	0
derived_msa-md	0
state_code	0
county_code	1060
census_tract	1505
conforming_loan_limit	272
derived_loan_product_type	0
derived_dwelling_category	0
derived_ethnicity	0

1-10 of 99 rows Previous 1 2 3 4 5 6 ... 10 Next

In the script above, the following steps are performed:

`is.na(hmda_data)`: This function checks each element of the **`hmda_data`** dataset to see if it is an NA value. It returns a logical matrix of the same dimensions as **`hmda_data`**, where each element is **`TRUE`** if the corresponding element in **`hmda_data`** is NA, and **`FALSE`** otherwise.

`colSums(is.na(hmda_data))`: This function calculates the sum of **`TRUE`** values (which are treated as 1) for each column in the logical matrix. As a result, it provides a named vector where each name corresponds to a column in **`hmda_data`**, and each value represents the count of NA values in that column.

`as.data.frame()`: This function converts the named vector into a data frame. This step is useful for better readability and further manipulation of the results. The resulting data frame has two columns: one for the column names from the original dataset and one for the corresponding counts of NA values.

By running this script, you will obtain a data frame (**`na_per_column`**) that lists each column in **`hmda_data`** along with the number of NA values it contains. This information is crucial for understanding the extent of missing data in each column, which can guide your decisions on how to handle these missing values in subsequent analysis steps.

3.2.2 Visualizing NA Values Using `visdat` Package

Visualizing missing values can provide a quick and intuitive understanding of the extent and distribution of NA values in your dataset. The **`visdat`** package in R offers a suite of tools for this purpose. In this section, we'll demonstrate how to visualize NA values using a subsample of the hmda_data dataset, focusing specifically on the "property_value" and "loan_amount" columns, we do this because the hmda_data is quite large and **`visdat`** can't handle large dataframes well.

Installing and Loading the `visdat` Package

First, ensure that the **`visdat`** package is installed and loaded. If you haven't installed it yet, you can do so with the following command:

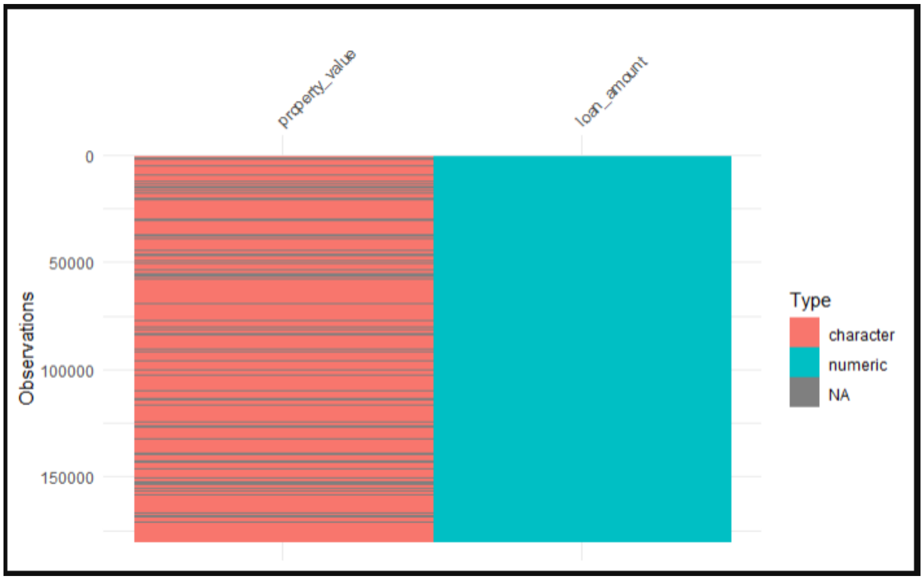
```r
install.packages("visdat")
```

Visualizing NA Values

To visualize the NA values in the **"property_value"** and **"loan_amount"** columns of the **hmda_data** dataset, you can use the **`vis_dat()`** function. Here's the script to create the visualization:

```r
# Loading libraries
library(visdat)
library(dplyr)

# Visualizing NA values
vis_dat(hmda_data %>%
          select(loan_amount, property_value), warn_large_data = FALSE)
```



In this script:

- **`hmda_data %>% select(loan_amount, property_value)`**: This line uses the `select()` function from the `dplyr` package to create a subsample of the `hmda_data` dataset, containing only the "loan_amount" and "property_value" columns.
- **`vis_dat()`**: This function from the `visdat` package generates a visualization of the dataset, highlighting the NA values.
- **`warn_large_data = FALSE`**: This argument suppresses warnings related to large datasets, which is useful when working with large data frames.

Understanding the Visualization

The visualization generated by **`vis_dat()`** provides a color-coded barchart where each bar represents a value in the dataset. The colors indicate different data types or the presence of NA values:

- **Gray cells**: Represent NA values.
- **Other colors**: Represent different data types (e.g., numeric, character, etc.).

This visual representation makes it easy to spot patterns and concentrations of missing data. For example, you can quickly see if NA values are clustered in certain rows or columns, which might suggest specific reasons for the missing data.

Chapter 4

Data Visualization

4.1 Data Visualization Using `ggplot2`

4.1.1 Introduction

Data visualization is a crucial step in data analysis as it helps in understanding the underlying patterns, trends, and relationships in the data. In this chapter, we will explore how to create various types of visualizations using the ggplot2 package in R, focusing on HMDA (Home Mortgage Disclosure Act) data.

4.1.2 Getting Started with ggplot2

First, ensure you have ggplot2 installed. If not, you can install it using:

```
install.packages("ggplot2")
```

This will download the latest version of `ggplot2` from the CRAN repository.

Load the package along with with the HMDA data:

```
library(ggplot2)
library(dplyr)  # For data manipulation
options(scipen = 999) # To prevent R from printing in scientific notation

# Load HMDA data
hmda_data <- read_csv("downloads/state_NV.csv", guess_max = Inf)
```

Once we have the data loaded, we proceed to do some preliminary prep and cleanup. The schema for the different data fields avaialable can be found at: https://ffiec.cfpb.gov/documentation/publications/modified-lar/modified-lar-schema

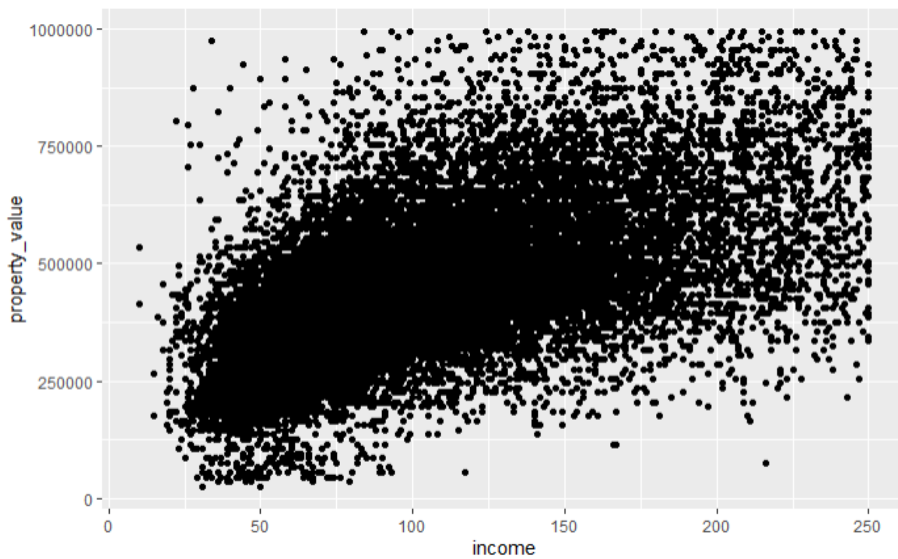
```
# Filter and prep HMDA data for plotting
filtered_hmda_data <- hmda_data%>%
  filter(
    # Filter for only originated transactions
    action_taken == 1,
    # Filter for only for home purchases
    loan_purpose == 1,
    # Filter for only primary homss
    occupancy_type == 1,
    # Filter for primary liens
    lien_status == 1,
    # Filter for single unit homes
    total_units == "1",
    # Filter propery value
    !property_value %in% c("Exempt", NA),
    # Filter income for values below 250 but above 0
    income <=250 & income>0,
    # Filter for Clark County
    county_code == "32003"

  )%>%
  mutate(
    property_value = as.numeric(property_value),
    loan_type = as.character(loan_type))%>%
    # Only keep property values under $1 million
  filter(property_value<1000000)
```

4.1.3 Basic Plot Structure

The structure of a ggplot2 plot is built around the **`ggplot()`** function and the **`+`** operator to add layers. Here's a simple example of a scatter pplot:

```
ggplot(data = filtered_hmda_data, aes(x = income, y = property_value))+
  geom_point()
```



In this example, `ggplot()` is the initial function call to create a new plot. The function takes the following primary arguments:

- **`data`**: This argument specifies the dataset to be used in the plot. In this case, **`filtered_hmda_data`** is the dataset containing the HMDA data.
- **`aes()`**: Short for aesthetics, this function defines the mapping of variables in your data to visual properties (aesthetics) such as x and y axes, colors, shapes, and sizes of points or lines. In the example, **`x = income`** maps the **`income`** variable to the x-axis, and **`y = property_value`** maps the **`property_value`** variable to the y-axis.

After the initial **`ggplot()`** function, we add layers to the plot using the **`+`** operator. Each layer represents a specific component of the plot, such as points, lines, bars, etc.

- **`geom_point()`**: This is a geometric object (geom) layer that adds a scatter plot layer to the plot. Each point represents an observation in the dataset.