WI ADA 2023 Class

Coleridge Initiative

Last Updated on 14 December, 2023

Table of contents

Preface

This contains the class notebooks for the Coleridge Initiative 2023 Wisconsin Applied Data Analytics class Examining Unemployment to Reemployment Trajectories of Unemployment Insurance Claimants.

1 Module 2: Workbook 1A

Exploratory Data Analysis

2 Introduction

Welcome to the first workbook for Module 2 of this course, covering **Exploratory Data Analysis**, or EDA.

In class, we learned that EDA is the process of examining your data to:

- Gain familiarity with its layout
- Discover patterns
- Spot possible errors
- Develop hypotheses

EDA helps you ensure that the provided data suits your desired analysis and that you have the necessary understanding to make informed analytic decisions as you work through your project.

In the Foundations Module, we learned that EDA itself is part of the larger process of **data discovery**. In this workbook, we're going to be `discovering` the two main datasets we will use in this course: the **PROMIS Unemployment Insurance (UI) claims data** and the **Wisconsin UI wage records**. We will work to answer key data discovery questions about each of these datasets, including

- **What is the structure of the dataset?** What variables are available? What are their types? How do we interpret the meaning of a single row?
- **What is the coverage of the dataset?** What years, geographic regions, and other subgroups are covered by the data?
- **What is the distribution of the variables in the dataset?** Are there missing observations, outliers, and other aspects we need to adjust for later in analysis?

2.1 The Purpose of These Workbooks

As we've discussed in class over the past few weeks, you will now apply the skills we learned in both modules thus far to restricted use Wisconsin data. With your team, you'll carry out a descriptive analysis of this data and prepare a final project showcasing the results of this analysis.

These workbooks are here to help you along in this project development by showcasing how to apply the techniques that we'll discuss in class to the Wisconsin microdata. Part of this workbook will be technical - providing basic code snippets your team can **modify** to begin parsing through the data. But, as always, there will also be an applied data literacy component of these workbooks, and it should help you develop a better understanding of the structure and use of the underlying data even if you never write a line of code.

The timeline for completing these workbooks will be given on the training website and communicated to you in class. Unlike the Foundations Module workbooks, these workbooks should be completed as homework after we have discussed the material in class.

3 Technical Setup

This workbook will cover both SQL and R coding concepts, so we need to set up our environment to connect to the proper database and run R code only accessible in packages external to the basic R environment. Typically, throughout these workbooks, we use SQL for the majority of data exploration and creation of the analytic frame, and then read that analytic frame into R for the descriptive analysis and visualization.

Note: If you aren't concerned with the technical setup of this workbook, please feel free to skip ahead to the next section, WI PROMIS data: `ds_wi_dwd.promis`.

3.1 Load Libraries

Just like we did in the Foundations Module, in running SQL and R code together through R, we need to load the `RJDBC` package. In addition, we will load the `tidyverse` suite of packages, as they will help us implement some of our fundamental data operations while maintaining a consistent syntax.

> Every time you create a new R file, you should copy and run the following code snippet.

```
options(scipen = 999) # avoid scientific notation
library(RJDBC)
library(tidyverse)
```

3.2 Establish Database Connection

To load data from the Redshift server into R, we need to first set up a connection to the database. The following set of commands accomplish this:

```
dbusr=Sys.getenv("DBUSER")
dbpswd=Sys.getenv("DBPASSWD")

url <- "jdbc:redshift:iam://adrf-redshift11.cdy8ch2udktk.us-gov-west-1.redshift.amazonaws.
```

```
driver <- JDBC(
  "com.amazon.redshift.jdbc42.Driver",
  classPath = "C:\\drivers\\redshift_withsdk\\redshift-jdbc42-2.1.0.12\\redshift-jdbc42-2.
)

con <- dbConnect(driver, url, dbusr, dbpswd)
```

As a reminder, don't worry too much about the details of this connection - **you can simply copy and paste this code each time you want to connect your R script to the Redshift database**.

3.3 New `.Renviron`

For this code to work, you need to create a new `.Renviron` file in your user folder (i.e. U:\John.Doe.P00002) that contains the following:

```
DBUSER='adrf\John.Doe.P00002'
DBPASSWD='xxxxxxxxxxxx'
```

where `John.Doe.P00002` is replaced with your username and `xxxxxxxxxx` is replaced with your password (both still in quotes!) The setup of this code is nearly identical to that required in the Foundations Module workspace - however, `DBUSER` should now end with `.T00111` instead of `.T00112`.

4 WI PROMIS data: `ds_wi_dwd.promis`

The primary dataset we will use in this class is the **Wisconsin PROMIS data.** The PROMIS (Program for Measuring Insured Unemployed Statistics) data, stored on Redshift as `ds_wi_dwd.promis`, provides information on unemployment insurance claimants in Wisconsin. Specifically, according to the LAUS Extraction Guide, the data includes "initial claims, additional initial claims, and continued claims that were either new or updated."

4.1 Structure of the PROMIS data

Let's begin answering our key data discovery questions. First - what is the structure of the PROMIS data? There are two dimensions to this question:

- What variables are available in the PROMIS data?
- What does each row in the PROMIS data represent?

To start, just like in the Foundations Module, let's glance at a few rows of the PROMIS data:

> Note: To avoid the consistent column-wise scrolling required, you can run the code stored in `qry` in DBeaver, as it is only written in SQL.

```
qry <- "SELECT * FROM ds_wi_dwd.promis LIMIT 5"

dbGetQuery(con, qry)
```

We can see that there are 43 columns available in the PROMIS data. The PROMIS data dictionary (Right-click on link to open) on the P: drive contains detailed descriptions of each variable.

The data dictionary descriptions, while helpful, do not provide the entire context for these variables. We also need to have a clear definition of what each observation - or row - in the PROMIS dataset represents. That is, we want to understand which variable or combination of variables **uniquely define** each row of the dataset.

By uniquely define, we mean that if you kept only this set of variables in the dataset, and counted all unique rows, it would return the same number of rows as the total dataset. In that case, we would know that those variables (and only those variables) are essential in defining that observation, or data point. If we concerned any smaller set of variables, then we would have multiple rows that look identical - not helping us uniquely identify each observation.

To know what we're aiming for, we can find the total number of rows in the PROMIS data:

```
qry <- "SELECT COUNT(*) FROM ds_wi_dwd.promis"

dbGetQuery(con, qry)
```

Let's think about the three categories of variables that might appear in our dataset to uniquely define our rows:

4.1.1 The Unit of Observation

The first category to consider is *variables that describe the unit of observation*. The unit of observation refers to the type of entity or object about which data is collected. This could be a person, a organization, a state, or any other entity that is the focus of data collection.

In the case of the PROMIS data, our unit of observation is the person - the individual claimants who have their claims reported in the dataset. Besides leveraging background information on the nature of the data, we can identify this because `ssn` is a person level variable in our dataset representing the smallest unit identified in the data.

Importantly, we need to note that the unit of observation alone does not necessarily define a row for our dataset. We might guess that each row of the PROMIS data defines a single individual, and `ssn` alone defines a row. But we can test this, using the SQL keyword `DISTINCT` to count the number of unique hashed social security numbers in our data:

```
qry <- "
SELECT COUNT(DISTINCT(ssn))
FROM ds_wi_dwd.promis
"

dbGetQuery(con, qry)
```

We see that there are far fewer unique `ssn` values than total rows in the table. This indicates that `ssn` alone does not define a row of our data - some individuals must appear in the data more than once. Why might there be multiple observations for the same individual? The most common reason is the one we'll talk about next: time.

4.1.2 Period of Observation

As we saw above, a given dataset might have multiple rows for each unit of observation. There are many reasons why this could be, but the most common one is that data about each unit of observation is observed at multiple points in time.

This introduces a separate category of variables in our dataset: those that help describe the *period of observation*, or the time interval of data collection for the unit of observation. It represents how often the data is captured or updated. This could also be referred to as the *time dimension* of the data.

Note that not every dataset will have variables representing the period of observation. For example, consider a dataset containing a list of all workers who have ever been employed in Wisconsin - each individual should only appear in the dataset once.

In the PROMIS data, the period of observation is represented by the variable `week_ending_date`, which indicates the end date of the week in which the claimant was eligible and claimed. Note that this is a **date** variable, which often can encode information about the period of observation.

Let's look at the amount of unique combinations of `ssn` and `week_ending_date` combined:

```
qry <- "
--CONCAT COMBINES THE VALUES IN week_ending_date AND ssn INTO A SINGLE VARIABLE

SELECT COUNT(DISTINCT CONCAT(week_ending_date, ssn))
FROM ds_wi_dwd.promis
"

dbGetQuery(con, qry)
```

This is exactly the number of total rows in our dataset! Therefore, we can say that the claims data is collected at the *person-week* level, because we see (at most) one observation per individual per week. Because the row numbers match, we know we don't have individuals who are marked as having been eligible and claimed during the same week.

4.1.3 Attributes

What about the other 41 variables described in the data dictionary?

These remaining variables represent the information that the PROMIS table contains about our person-week observations. There are several different types of attributes that we can see in our data dictionary:

- Individual attributes: `birth_date`, `commuter`, `education`, `gender`, `race` and other similar variables
- Last employer attributes: `last_employer_naics` and `last_employer_name` describe the employer with maximum last day of work provided by the claimant
- Claim attributes: `ic_claim_date`, `montetraily_eligible`, `program_type`, and other similar variables describe information about the UI claim the individual is making

Continuing this process through the remaining variables can help you get a better understanding of your data, and will greatly aid in gauging the potential of your team's research ideas.

4.2 Coverage of the PROMIS data

The next step of our exploratory data analysis is to determine the data coverage: what time span, region, and other important subgroups are captured in the data? Luckily, this is pretty straightforward for the PROMIS data: we know that the geographical area is the entire state of Wisconsin, and that the subgroup covered is UI claimants. To determine the time periods covered, we can look group our data by week:

```
qry <-  "
SELECT week_ending_date, COUNT(*)
FROM ds_wi_dwd.promis
GROUP BY week_ending_date
ORDER BY week_ending_date
"

counts_by_week <- dbGetQuery(con, qry)

counts_by_week
```

We can flip through the table above and begin to get a feeling for the number of individuals from each week, but you can also use this same summary dataset to quickly calculate our coverage:

```
min(counts_by_week$week)
max(counts_by_week$week)
```

From this, we see that the PROMIS data begins in the week ending on REDACTED, and ends after the week ending REDACTED.

We can also note that the week with the highest number of claimants was the week ending REDACTED:

```
counts_by_week %>%
  filter(
    count == max(counts_by_week$count)
  )
```

4.3 Distribution of the PROMIS data

The final part of data discovery, is in looking at the distribution of key variables in our dataset, and documenting any irregularities such as missingness, outliers, invalid values, or other issues that need to be addressed in an analysis leveraging these data components. We'll look at a few of the variables from the PROMIS data now, but this type of exploration is more of an art than a science, and the continued exploration of these variables will be an essential component of your projects going forwards in the class. As a reminder, you are encouraged to use this code as inspiration in your own exploration.

4.3.1 `week_ending_date`

Let's begin by taking the same data pull we just used to examine the data coverage, and plot the data using a simple line plot:

```
qry <-  "
SELECT week_ending_date, COUNT(*)
FROM ds_wi_dwd.promis
GROUP BY week_ending_date
ORDER BY week_ending_date
"

counts_by_week <- dbGetQuery(con, qry)

ggplot(counts_by_week, aes(x = week_ending_date, y = count)) +
  geom_line()
```

Immediately, we can see a massive jump in records in REDACTED, which should make sense given Wisconsin's economic situation at the time. Does anything else catch your eye? Take some time to examine this graph.

4.3.2 `birth_date`

Let's now look at the distribution of claimants by their year of birth, reported in the PROMIS data as part of the variable `birth_date`. We will use the SQL function `EXTRACT` to isolate the year.

> Note: We have to use `COUNT(DISTINCT(SSN))` instead of `COUNT(*)` because an individual may be represented in more than one row.

```
qry <- "
SELECT extract(year from birth_date) as year, COUNT(DISTINCT(SSN))
FROM ds_wi_dwd.promis
GROUP BY year
ORDER BY year DESC
"

counts_by_birthday <- dbGetQuery(con, qry)

counts_by_birthday
```

Looking at just the first page of this table is surprising! We see that there are individuals with no reported birthday or with birthdays either in the future or so recently as to be improbable. To see this laid out further, we can plot the above data:

```
ggplot(counts_by_birthday, aes(x = year, y = count)) +
  geom_bar(stat = "identity")
```

This confirms our first concerns from the table - there seems to be a weird island of implausible birthdays at the far right side of the graph. For now, we will proceed with our EDA, but these are the kinds of underlying data quality issues that are essential to flag and handle properly later in your eventual analysis. Think about the potential root cause(s) of this issue and the potential impact on your project work.

4.3.3 `ethnicity` and `race`

Finally, we'll look at the distribution of two additional individual characteristic variables, starting with `ethnicity`.

```
qry <- "
SELECT ethnicity, COUNT(DISTINCT(SSN))
FROM ds_wi_dwd.promis
GROUP BY ethnicity
```

```
"
dbGetQuery(con, qry)
```

And now `race`:

```
qry <- "
SELECT race, COUNT(DISTINCT(SSN))
FROM ds_wi_dwd.promis
GROUP BY race
ORDER BY race
"

dbGetQuery(con, qry)
```

For each of these variables, we encourage you to refer back to the data dictionary and check what each value above refers to, and whether these are the values we would expect. For both variables, we do see missing observations, as well as an asterisk in the ethnicity variable, which also refers to missing values. What might these mean? For whom might race be unreported? We will discuss the implications of missing data in the Missingness and Inference Errors lecture, but it's helpful to think about these issues as you continue scoping your project.

As a note, the data dictionary is incomplete, but we believe that `race = 8` represents Hispanic claimants. In addition, `NA` and `0` values indicate unknown missing race here, rather `9` as indicated in the dictionary.

4.4 Checkpoint

Using the data dictionary, identify one or more further variables from the PROMIS data that might be relevant to your group's analysis. Think through what these variables "should" look like, as well as what issues might arise. Working individually or with your group, examine the distribution of these variables. Document any EDA-related concerns and findings in your team's project template. Brainstorm as to what the cause of these issues might be, and how it could impact your analysis.

5 WI UI wage records: `ds_wi_dsd.ui_wage`

We're now going to apply these same EDA concepts to a second dataset, Wisconsin's UI wage records, which are stored on Redshift as `ds_wi_dwd.ui_wage`.

We will keep the narrative of our exploration far briefer in this section. You are encouraged to read through the following output and think about how it pertains to the discussions that we had above.

5.1 Structure of the UI wage records

Let's preview the wage data as before:

```
qry <- "
SELECT *
FROM ds_wi_dwd.ui_wage
LIMIT 5
"

dbGetQuery(con, qry)
```

We see there are thirteen variables in our dataset. Detailed descriptions of these columns can be found in the UI wage data dictionary (Right-click on link to open) on the P drive.

Let's analyze these:

- Unit of observation: The "reference worker". Part of this, as before, is the individual, defined by `ssn`, `name_first`, `name_middle`, `name_last`. All of these variables are hashed. But, since we are specifically concerned about the worker here, the definition of the reference worker also includes the employer identifier, `ui_account`. In other words, our unit of observation is the person-employer combination.
- Period of observation: Calendar quarter, defined by `year` and `quarter`
- Attributes:
  - Employer attributes: `fips_state_code`, `ui_account`, `sein_unit`, and `ein` all hold information on the reference worker's employer

- Labor attributes: `wage`, `hours`, and `weeks` are all measures of how much the reference worker worked and was paid in a given week. Both `hours` and `weeks` are not populated in the table.

As before, our rows are defined by the combination of our unit and period of observation variables - the UI wage records are available at the **person-employer-quarter** level. We can confirm this fact:

```
qry <- "
SELECT COUNT(*)
FROM ds_wi_dwd.ui_wage
"

dbGetQuery(con, qry)
```

```
qry <- "
SELECT COUNT(DISTINCT CONCAT(CONCAT(CONCAT(ssn, year), quarter), ui_account))
FROM ds_wi_dwd.ui_wage
"

dbGetQuery(con, qry)
```

You might note that there is still a slight discrepancy in the counts. While the majority of the data is at the person-employer-quarter level, there are some observations that are not unique up to these three variables. Specific causes vary: some cases are due to changing names, some cases represent just actual duplicated rows of the data. These duplicates don't indicate that our definition of a row is wrong, but rather that there is remaining underlying messiness in the data that your team will need to address for your sample of the data as part of your data cleaning.

5.2 Coverage of the UI wage records

We know our wage data covers most earners in Wisconsin, but over what time period? We can answer this by looking at observations by quarter:

```
qry <- "
SELECT year, quarter, COUNT(*)
FROM ds_wi_dwd.ui_wage
GROUP BY year, quarter
ORDER BY year, quarter
"
```

```
dbGetQuery(con, qry)
```

The coverage of our data is from REDACTED to REDACTED, with no gaps in years or quarters.

5.3 Distribution of the UI wage records

5.3.1 `year`

Let's look at the trends by year for the number of observations in the wage data:

```
qry <- "
SELECT year, COUNT(*)
FROM ds_wi_dwd.ui_wage
GROUP BY year
ORDER BY year
"

count_per_year <- dbGetQuery(con, qry)

ggplot(count_per_year, aes(x = year, y = count)) +
  geom_line()
```

This graph looks as expected, except the first and last years are much lower than all the others. For the first year, REDACTED, this is to be expected: our coverage query revealed that there was only one quarter of that year available.

But REDACTED still seems exceptionally low. Looking back at our table of counts by quarter, we can see that beginning in REDACTED, the total number of observations drops by around 2 million per quarter. We are still working on understanding the cause of this drop, so stay tuned for a further update and explanation!

5.3.2 `wage`

Next, let's explore the distribution of wages. An easy place to start is the range of observations: what are the minimum, median, and maximum wages recorded?

```
qry <- "
SELECT MIN(wage), MEDIAN(wage), MAX(wage)
FROM ds_wi_dwd.ui_wage
```

```
"

dbGetQuery(con, qry)
```

Note, here we use the median because, as compared to the mean, it will be less impacted by large outliers. We can also look at these summary statistics by year:

```
qry <- "
SELECT year, MIN(wage), MEDIAN(wage), MAX(wage)
FROM ds_wi_dwd.ui_wage
GROUP BY year
ORDER BY year
"

dbGetQuery(con, qry)
```

From this table, we can see that:

- The maximum is usually many order of magnitude larger than the median, indicating that it, as well as some other potential values, are likely outliers
- From REDACTED-REDACTED, the maximum wage is REDACTED. This is likely due to a decision in the database during that period to "top-code" or cut off the highest earners.

To explore this further, let's use a bar plot to look at wages in 4 income bands in a single quarter:

```
qry <- "
SELECT wage
FROM ds_wi_dwd.ui_wage
WHERE year = 2019 AND quarter = 4
"

wages_2019_q4 <- dbGetQuery(con, qry) %>%
  mutate(
    # bin the data into 4 groups for plotting
    bin = cut_interval(wage, n = 4, dig.lab = 7)
  ) %>%
  group_by(bin) %>%
  summarize(
    count = n()
  )
```

```r
ggplot(wages_2019_q4, aes(x = bin, y = count)) +
  geom_bar(stat = "identity")
```

This graph shows us that almost all wage records are at the very bottom of the distribution, with a few shifting the x-axis rightward - even though there aren't even enough observations in those higher groups to appear on our graph! This is a common feature of wage distribution, and something to be aware of moving forward, especially when reporting certain statistics that may be influenced by outliers, such as an average (again, as opposed to the median).

5.4 Checkpoint

As with the PROMIS data, use the data dictionary to identify one or more further variables from the UI wage records that might be relevant to your group's analysis. Think through what these variables "should" look like, as well as what issues might arise. Working individually or with your group, examine the distribution of these variables. Document any EDA-related concerns and findings in your project template. Brainstorm as to what the cause of these issues might be, and how it could impact your analysis.

6 Next Steps: Applying the workbook to your project

The workbook provides a structure for you to start your EDA process on the data within the scope of your project. The data coverage and row definition for the two primary datasets in this training is available, allowing you to focus on evaluating the distribution of variables potentially relevant to your analysis. The data coverage is particularly essential for project ideas linking the two datasets, as you will want to select a set of years, quarters, and weeks that are available in potentially both datasets.

As you evaluate variable distributions, you can start by repurposing the code in these sections. There are code snippets for distributions of numeric, time-based, and categorical variables that may be appropriate depending on the type of column you are interested in exploring.

In doing so, as recommended in the checkpoints, note your findings in your team's project template. As your project progresses, it will be helpful to look back at these notes, especially in thinking through how to most accurately and best communicate your team's final product to an external audience. Ultimately, the EDA process is an essential step in the project development lifecycle, as it provides helpful contextual information on the variables you may choose to use (or not use) in your analysis.

For all of these steps, remember not to take notes or discuss exact results outside the ADRF. Instead, create notes or output inside the ADRF, and store them either in your U: drive or in your team's folder on the P: drive. When discussing results with your team, remember to speak broadly, and instead direct them to look at specific findings within the ADRF. And, as always, feel free to reach out to the Coleridge team if you have any questions as you get used to this workflow!

7 Citation

LAUS EXTRACTION GUIDE

AR EDA Notebook (link to come)