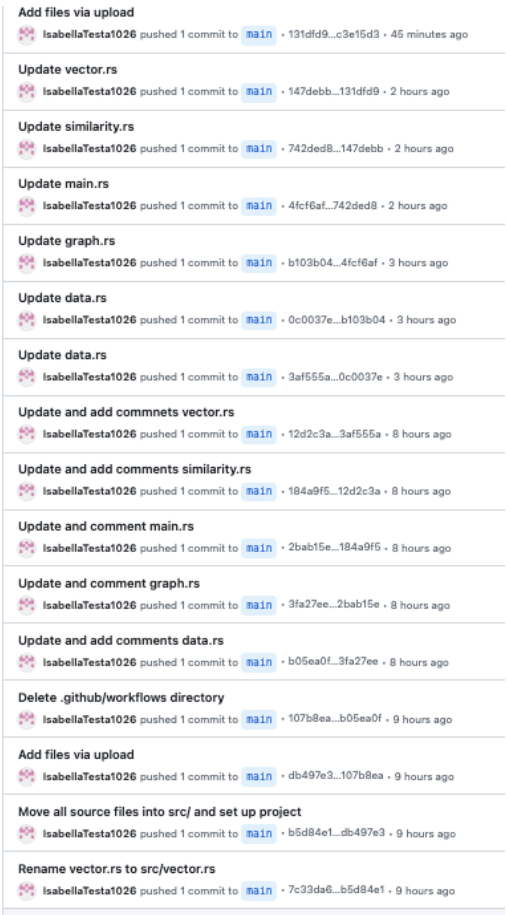
A. Project Overview

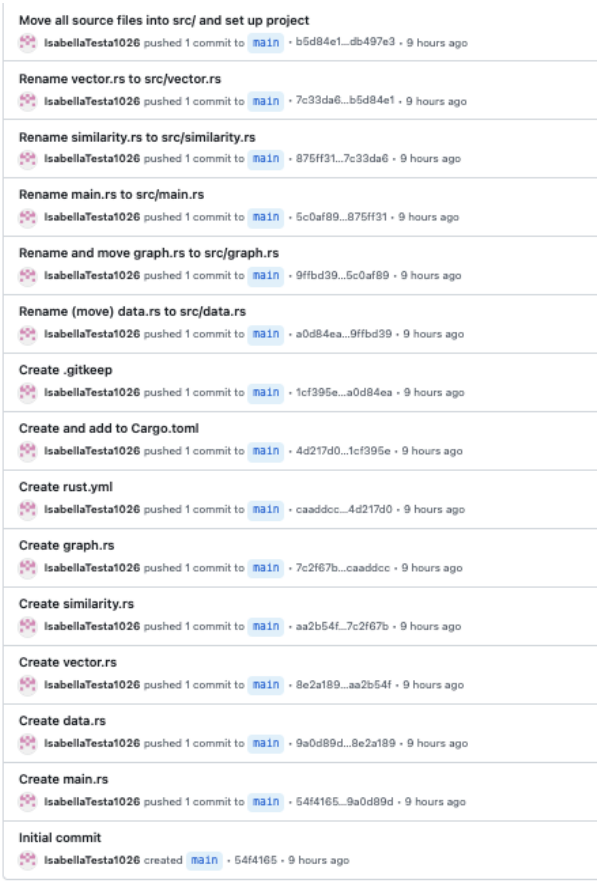
Goal:

My project group compares and analyzes attendance trends in school districts in Connecticut in order to find inequities among student groups, watch for these individual patterns, and track over a period of a few years. I wanted to see the similarities and differences among districts and how specified groups impacted these patterns.

Data Set: Linked in the GitHub repo.

Note: I did not realize I could commit in GitHub's terminal, so I did them all within the repo directly.





B. Data Processing

In order to load the data into Rust, I used a standard CSV file as well as Serde for deserialization to reconstruct the data structure to fit Rust formatting. I then defined RawDistrictRecord struct to match the raw CSV structure. So each part of the struct matched the columns within the actual CSV file. I then transformed this into a cleaned DistrictRecord struct, which included the rates_by_year field stored as a HashMap<String, f64>. During the cleaning process, I removed percentages from attendance rates and converted the strings into floating-point numbers in hopes of having the exact data for my calculations. I then grouped records beginning with the student group and then the district. I ended by constructing feature vectors for each district, ensuring the values were sorted by year.

C. Code Structure

My code is split into 4 content modules and 1 main; I did this for organizational purposes.

Module 1: Data

Module Purpose:
The data module is responsible for importing, cleaning, and structuring district-level attendance data from a CSV file. It defines data structures to represent both the raw and cleaned formats of the dataset and provides a function to convert the raw CSV entries into a Rust-friendly format.

Imports and Setup:
The code begins by importing the necessary library features. It brings in HashMap for organizing attendance rates by year and Error for handling potential failures when reading the file. It also uses the serde crate for deserializing the CSV rows into Rust structs, and csv::ReaderBuilder to configure how the CSV file is read.*

RawDistrictRecord Struct:
This struct maps directly to the raw CSV format, using #[derive(Deserialize)] to convert rows of the file into Rust values. Each field is matched to its corresponding column name in the original CSV using #[serde(rename = "...")]. All attendance rates are stored as strings initially to retain formatting (such as % symbols) for later cleaning, as I wanted to ensure exact values.

DistrictRecord Struct:
This is the cleaned and structured version of the data. Instead of keeping rates as strings, the DistrictRecord converts them into type f64, storing them in a HashMap where the

keys are year strings and the values are the corresponding attendance rates. This makes it easier to compute and analyze trends later on.

load_data Function:
This function performs the core data loading task. It reads the CSV from a given file path using ReaderBuilder, loops through each row, and deserializes it into a RawDistrictRecord. For each row, it processes the attendance rate fields by removing any % symbols and converting them to f64. These cleaned values are stored in a rates_by_year map, which is then grouped into a DistrictRecord. All such records are collected into a vector and returned. If any error occurs (like a missing file or parsing failure), it's wrapped in a boxed Error type for error handling.

Module 2: Vector

Module Purpose:
The vector module processes the newly cleaned attendance data to create numerical vectors grouped by student group and district. These vectors will help to calculate similarity between districts and build graphs based on those similarities.

Imports and Dependencies:
The module starts by importing standard data structures from the Rust standard library: HashMap to associate groups and districts with their respective vectors, and BTreeSet to automatically sort year keys**. It also imports the DistrictRecord struct from the data module, which provides the cleaned attendance data used as input.

build_grouped_feature_vectors Function:
This function transforms slices of DistrictRecord entries into a nested hash map. The outer HashMap maps student groups (like "All Students") to another HashMap, which maps each district name to a Vec<f64>, which represents attendance rates over time. The function first gathers all unique years from the dataset into a BTreeSet, ensuring the years are sorted. Then, for each record, it checks if an entry for the student group and district already exists; if not, it initializes one with a vector of zeros. Finally, it fills in the attendance rates in the correct year-based index using the sorted year order.

Module 3: Similarity

Module Purpose:
This module defines two computational functions to compute similarity metrics between vectors, specifically, it calculates Euclidean distance and Manhattan distance. These functions are what check for similarity based on their attendance patterns over time, and will help other tasks like graph construction and clustering.

I wanted to use Euclidean distance to capture the straight-line difference between districts, as it would help with the more similar data, but I realized that there may be outliers that it will not represent as well (or overly represent). So I wanted to have a method to compare it to. So I used Manhattan distance to measure the sum of absolute differences, treating all deviations equally. This helps see the validity of my results.

euclidean_distance Function:
This function computes the Euclidean distance between two equal-length vectors of f64 values. It begins by asserting that the two vectors are of the same length to avoid runtime errors. Then, for each pair of elements in the vectors, it calculates the squared difference and finds the result. After the loop, it returns the square root of the total, yielding the final Euclidean distance.

manhattan_distance Function:
This function computes the Manhattan distance between two equal-length vectors. It checks that the input slices are the same length, then sums up the absolute differences between each corresponding pair of values. The result is a total "path" distance, like navigating a grid-based city. This metric is useful for capturing overall differences without amplifying outliers as strongly as Euclidean distance does.

Module 4: Graph

Module Purpose:
This module builds undirected similarity graphs between school districts based on their attendance patterns. It supports constructing graphs using the two types of distance measures I am using to compare: Euclidean and Manhattan. It uses these measures to define edges between districts whose attendance vectors are sufficiently similar.

Imports and Dependencies:
The module imports Graph and Undirected from the petgraph crate to create graph structures**. It also uses HashMap to map district names to graph node indices. Additionally, it brings in the euclidean_distance and manhattan_distance functions from the similarity module, which are necessary for determining whether a connection (edge) should be made between any two districts.

build_euclidean_graph Function:
This function builds an undirected graph where each district is a node, and edges represent districts whose Euclidean distance in attendance patterns is below a defined max_distance threshold. First, all district names are added as nodes in the graph. Then, for each unique pair of districts, the function computes the Euclidean distance between their feature vectors. If the distance is within the threshold, an edge is added between the

nodes with the distance as the edge weight. This allows the graph to represent "closeness" in attendance trends.

build_manhattan_graph Function:
This function mirrors the structure of build_euclidean_graph, but uses the Manhattan distance metric instead. It adds all districts as nodes and compares each unique pair. If the total absolute difference between their feature vectors (i.e., the Manhattan distance) is within the threshold, an edge is created.

Main:

Module Purpose:
The main module will be where the program loads attendance data, computes pairwise similarity between districts for each student group using Euclidean and Manhattan distances, and then builds and summarizes similarity graphs.

Imports and Module Declarations:
The code begins by declaring the submodules and importing specific functions from each. These imports allow the main function to access data loading (load_data), feature vector construction (build_grouped_feature_vectors), distance computations (euclidean_distance and manhattan_distance), and graph-building utilities (build_euclidean_graph and build_manhattan_graph).

Step 1–2: Load and Structure Data:
The program first calls load_data to import the cleaned attendance CSV file into structured Rust objects. Then, build_grouped_feature_vectors groups this data by student group and district, returning a nested map where each district has a sorted vector of attendance rates for analysis.

Step 3: Print Group Names:
All unique student groups (13) found in the dataset are extracted and printed.

Step 4–6: Distance Calculations Between Districts:
For each student group, the program iterates through all unique district pairs and computes both Euclidean and Manhattan distances between their attendance vectors. These metrics quantify the similarity (or dissimilarity) between districts' attendance patterns over time.

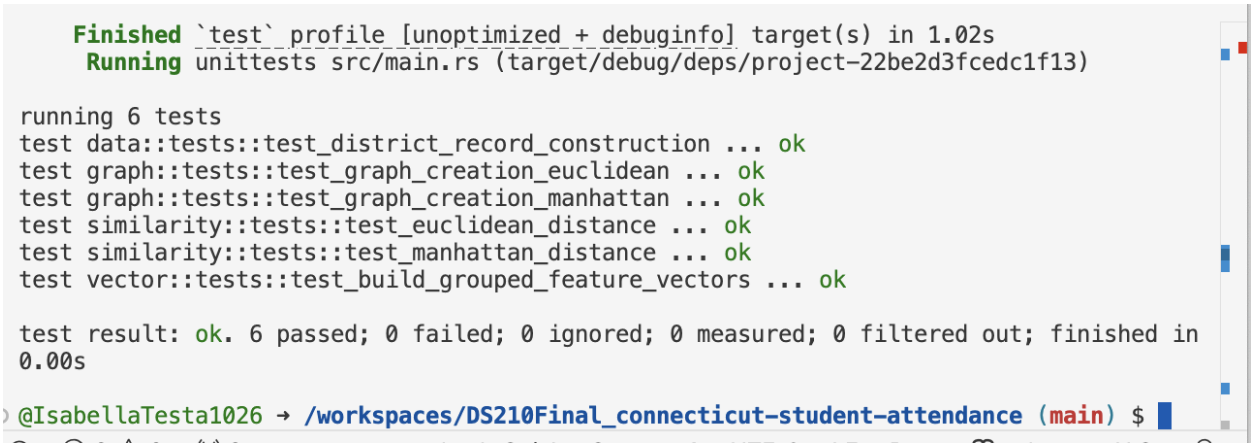
Step 7–9: Identify Most Similar and Dissimilar Pairs:
The computed distances are sorted in ascending order. The top 5 district pairs with the smallest Euclidean distances are considered the most similar and are printed first. Then,

the bottom 5 (with the largest Euclidean distances) are printed as the most dissimilar. For each, the actual vectors are shown to give context.

Step 10: Compute and Print Averages:
The program calculates the average Euclidean and Manhattan distances for each student group. This gives a quick summary of overall similarity trends within the group, helping to compare group-level variability.

Step 11–12: Graph Construction and Summary:
I end the code by constructing two undirected graphs per group. One uses Euclidean distance, the other uses Manhattan distance, with a distance threshold of 0.05. The number of nodes and edges in each graph is printed, providing insight into the network density and how tightly districts are connected under each similarity.

D. Tests



Unit tests are included to check functionality within each module. In the data module, a test confirms a DistrictRecord is correctly constructed and holds the right attendance values. In the similarity module, tests verify that distance calculations give expected results for known input vectors, ensuring, for example, that the Euclidean distance between identical vectors is zero. The graph module includes tests to confirm small sets of vectors produce the correct number of graph nodes and edges when passed to the graph-building functions. When cargo test is run, all tests pass successfully, outputting the above results.

E. Results

Section of final output:

Group: Hispanic/Latino of any race

Top 5 most similar (lowest Euclidean distance) districts:
1. Danbury School District <--> New Milford School District
Euclidean: 0.002784
Manhattan: 0.005300
  Danbury School District   => [0.9502, 0.9323, 0.9195, 0.9246]
  New Milford School District => [0.9511, 0.9332, 0.9177, 0.9229]

2. Newtown School District <--> Granby School District
Euclidean: 0.003114
Manhattan: 0.004600
  Newtown School District   => [0.9572, 0.9612, 0.9398, 0.9395]
  Granby School District   => [0.9576, 0.9613, 0.9411, 0.9367]

3. Odyssey Community School District <--> Regional School District 05
Euclidean: 0.003554
Manhattan: 0.005300
  Odyssey Community School District => [0.9507, 0.9391, 0.9298, 0.939]
  Regional School District 05 => [0.9509, 0.938, 0.9331, 0.9397]

4. Wilton School District <--> Darien School District
Euclidean: 0.003667
Manhattan: 0.005300
  Wilton School District   => [0.959, 0.9773, 0.9516, 0.9494]
  Darien School District   => [0.9594, 0.9739, 0.9514, 0.9507]

5. East Hampton School District <--> North Haven School District
Euclidean: 0.003708
Manhattan: 0.006100
  East Hampton School District => [0.9554, 0.9543, 0.9418, 0.9375]
  North Haven School District => [0.9521, 0.9534, 0.9431, 0.9369]

Top 5 most dissimilar (highest Euclidean distance) districts:
1. Deep River School District <--> Essex School District
Euclidean: 1.676877
Manhattan: 2.917500
  Deep River School District => [0.0, 0.0, 0.0, 0.9316]
  Essex School District   => [0.9736, 0.9761, 0.9545, 0.9449]

2. Stamford Charter School for Excellence District <--> Essex School District
Euclidean: 1.676828

Manhattan: 2.907800
  Stamford Charter School for Excellence District => [0.0, 0.0, 0.0, 0.9485]
  Essex School District   => [0.9736, 0.9761, 0.9545, 0.9449]

3. Deep River School District <--> New Canaan School District
Euclidean: 1.672988
Manhattan: 2.917800
  Deep River School District => [0.0, 0.0, 0.0, 0.9316]
  New Canaan School District => [0.9633, 0.9758, 0.9583, 0.952]

4. Stamford Charter School for Excellence District <--> New Canaan School District
Euclidean: 1.672867
Manhattan: 2.900900
  Stamford Charter School for Excellence District => [0.0, 0.0, 0.0, 0.9485]
  New Canaan School District => [0.9633, 0.9758, 0.9583, 0.952]

5. Deep River School District <--> Weston School District
Euclidean: 1.668268
Manhattan: 2.905700
  Deep River School District => [0.0, 0.0, 0.0, 0.9316]
  Weston School District   => [0.9558, 0.9732, 0.9603, 0.948]

Average Euclidean distance for group 'Hispanic/Latino of any race': 0.304330
Average Manhattan distance for group 'Hispanic/Latino of any race': 0.429559

Euclidean graph for group 'Hispanic/Latino of any race' has 170 nodes and 5246 edges
Manhattan graph for group 'Hispanic/Latino of any race' has 170 nodes and 2340 edges

Due to the length of the results; the entirety of the output can be found in a text file in the repo named: Attendence_data_Results.txt

Interpretation:

The overall results of my project show patterns of district similarity within specific groups as a result of the small Euclidean and Manhattan distances, indicating similar environment, policy, etc that would lead to similar attendance outcomes for these groups of students. On the other hand, the most dissimilar pairs had very high distances, with some over 1.6 for Euclidean and nearly 3.0 for Manhattan. This indicates some extreme attendance disparities in the data, which could indicate improper data or questionable environments in these districts.

The graph structure (node and edge counts) confirms density differences for different groups. For example, the “All Students” group had very dense graphs (over 12,000 edges for Euclidean), while marginalized groups like “Students Experiencing Homelessness” had very sparse connectivity (only 11 and 3 edges), which could be interpreted as seeing the inequality of care that is put when it comes to acquiring their data. Using both Euclidean and Manhattan distance showed that while rankings of similarity were often similar, the edge density and sensitivity to outliers differed. Euclidean distances exaggerated large differences due to squaring, whereas Manhattan distances produced more moderate differences, especially in sparsely distributed groups.

Overall, the outputs contextualize which districts may benefit from collaboration in improving their attendance for these marginalized groups or shift their current policy approaches to do so.

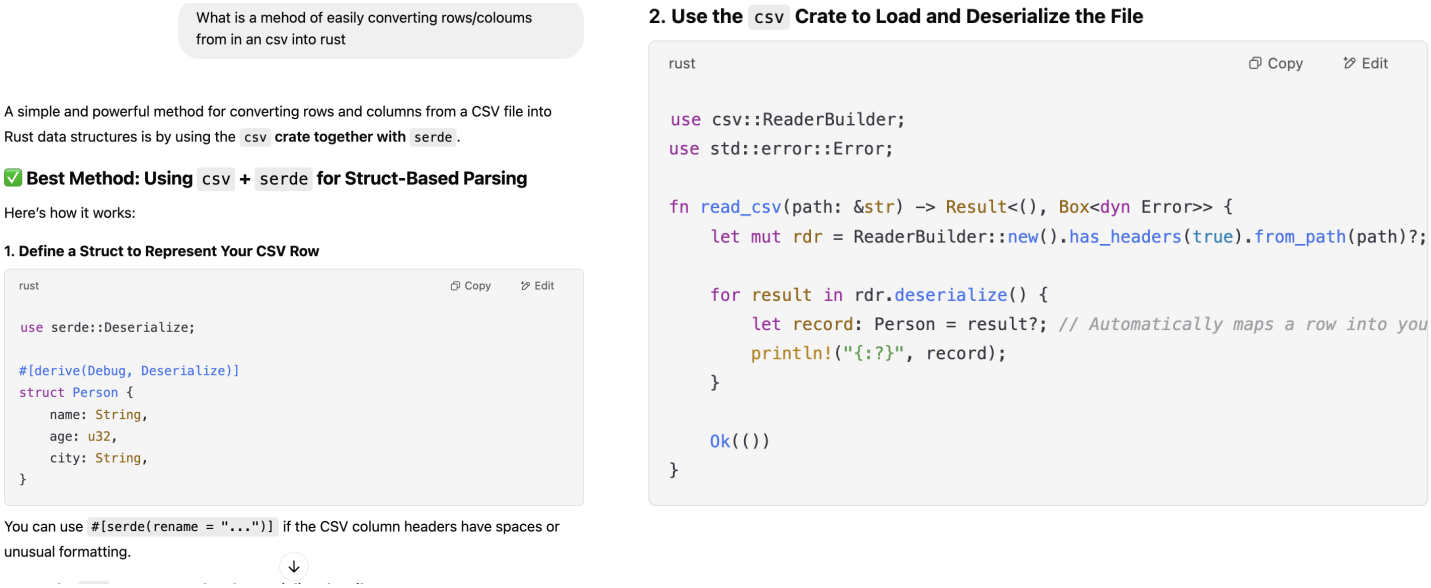
F. Usage Instructions

There is no required user input in the code. However, to run it successfully, the program expects the CSV file to be present and the column headers to match the expected group names exactly. The runtime is approximately 0.02 seconds.

G. AI-Assistance Disclosure and Other Citations

*Serde Crate

I asked Chat for help cleaning up my data and converting the csv into rust as I was unsure how to go about doing so.



What is a mehod of easily converting rows/coloums from in an csv into rust

A simple and powerful method for converting rows and columns from a CSV file into Rust data structures is by using the `csv` **crate together with** `serde`.

✅ **Best Method: Using** `csv` **+** `serde` **for Struct-Based Parsing**

Here's how it works:

1. Define a Struct to Represent Your CSV Row

rust Copy Edit

```rust
use serde::Deserialize;

#[derive(Debug, Deserialize)]
struct Person {
    name: String,
    age: u32,
    city: String,
}
```

You can use `#[serde(rename = "...")]` if the CSV column headers have spaces or unusual formatting.

2. Use the `csv` **Crate to Load and Deserialize the File**

rust Copy Edit

```rust
use csv::ReaderBuilder;
use std::error::Error;

fn read_csv(path: &str) -> Result<(), Box<dyn Error>> {
    let mut rdr = ReaderBuilder::new().has_headers(true).from_path(path)?;

    for result in rdr.deserialize() {
        let record: Person = result?; // Automatically maps a row into you
        println!("{:?}", record);
    }

    Ok(())
}
```

I continued to do my own research of how to use serde as well which led me to this article: https://medium.com/@joshmo_dev/using-serde-in-rust-dc620ce4beff

Explanation: Serde is a tool for serializing (turning Rust data into text formats like CSV or JSON) and deserializing (turning text files like CSV into Rust data structures). In this project, I used Serde for deserialization. it takes the raw text data from a CSV file and automatically converts it into well-structured and typed Rust data, such as a struct. This means I can work with grouped, typed data in Rust without manually parsing strings. To use it, you import Serde with use serde::Deserialize and define a struct that matches the column names (or groups) in the data file you're converting. Serde then handles the rest.

**BTreeSet

I knew I wanted to use a binary tree to effectively store my data, but was confused on how to do so in Rust effectively. I used these sites mainly as examples (ended up using the first one):
https://doc.rust-lang.org/std/collections/struct.BTreeSet.html

https://doc.rust-lang.org/std/collections/struct.BTreeMap.html

Explanation: I used a BTreeSet to sort the attendance data across multiple years (2020–2023) in the correct order without having to worry about duplicates. It ensured that all the years were stored once and in order, which made it easier to access and organize the data consistently later when building the attendance vectors.