William Parker

Final Project Writeup

Overview

The objective of this project is to analyze a lot of data on suicide rates in the United States of America over the past 70 years or so in a meaningful way. The data is broken down in many different ways, based on number adjustments and/or demographic categories such as just sex, sex and race, sex and race and age, or other such combinations. Some of these subcategories of the data have far more data to work with, so I focused on a subgroup with over 1000 singular points of data to work with. The main output to reach is a visual representation of the graph structure created from this subset. For the visual implementation, I used Python, and I used Rust to read the initial data set, parse it, select our subgroup, create the graph with inlaid logic for weighted edges, and export it as a meaningful CSV file that the Python code could read in to create the visual graph representation.

How To Use

To use this code, simple run the rust code first by running “cargo run”, which should read in the appropriate data, parse it, create a graph of connected nodes, and then export the data as a CSV file of connected nodes with edge weights. Then, run the Python code via “python .\visual.py”, and it will create and export a graph as a png in the repo.

Code Explanations

Analysis.rs - This code implements the **Label Propagation Algorithm (LPA)** for graph-based clustering, where each node in a graph is iteratively assigned a label based on the most frequent labels of its neighboring nodes, taking edge weights into account. The graph is represented as an adjacency list, where each node has a list of connected neighbors and the associated edge weights. The most_frequent_label function determines the most common label among a node's neighbors, weighted by edge weights. It first accumulates the weights for each label using a HashMap, where the key is the label, and the value is the cumulative weight. The labels with the highest cumulative weight are returned, ensuring that the most influential (or frequently occurring) labels among neighbors are considered. The propagate_labels function applies this logic to propagate labels across the graph. For each node, it identifies the most frequent label among its neighbors using most_frequent_label and checks if the label of the current node needs to be updated. If the label changes, the function flags this update and continues iterating. A clone of the labels is used to avoid modifying the labels map while iterating over it, which ensures consistent updates during a single iteration. The run_label_propagation function serves as the wrapper that drives the overall LPA process. It initializes each node's label as its own identifier, creating a one-to-one mapping between nodes and labels. Then, the algorithm iteratively propagates labels across the graph using the propagate_labels function until no labels are updated, signaling convergence. Finally, the function constructs a new graph where node names

and their connections are replaced by the final propagated labels, ensuring that nodes sharing the same label are grouped together. This implementation emphasizes simplicity and clarity by splitting the functionality into modular components: finding the most frequent label, propagating labels, and running the algorithm to completion. By using weighted edges, the algorithm considers the strength of connections between nodes, enabling more meaningful label assignments. The final graph returned is structurally identical to the input graph but with labels updated to reflect the clustering results, making it easy to export or analyze further. This approach was chosen for its efficiency, scalability, and suitability for community detection tasks in weighted graphs.

Data_loader.rs - The function load_data_from_csv reads data from a CSV file and converts each row into a Node object, returning a vector of these nodes. It begins by attempting to open the specified file path using csv::Reader, and if the file cannot be opened, it prints an error message and returns an empty vector to gracefully handle the failure. For each record (row) in the file, the function extracts fields corresponding to various attributes, such as unit, demographic information, year, age group, and an estimate value, and parses these fields into appropriate types like String, i32, or f64. If any parsing fails, the function falls back to default values to ensure robustness against malformed or incomplete data. A Node object is then constructed using the Node::new method and added to a vector for storage. In cases where a row cannot be read successfully, an error message is printed with the specific row number for debugging purposes, but the process continues to ensure that other valid rows are processed. Finally, the function returns the vector containing all the successfully created nodes. This implementation is designed to be robust, efficient, and modular, ensuring that raw CSV data can be seamlessly transformed into a structured format for use in further processing, such as building graphs or running clustering algorithms.

Graph_exporter.rs - The function export_graph_to_csv takes a graph represented as a HashMap of nodes and their connected edges with weights, and a string specifying the name of the output file. The function creates a CSV file where each row represents an edge in the graph, formatted as "Source,Target,Weight". It begins by attempting to create the specified output file using the File::create function and wraps the file handle in a BufWriter to enable efficient buffered writing. A header row, "Source,Target,Weight", is written first to indicate the structure of the CSV file. Then, the function iterates through the graph's adjacency list, where each key represents a node and its corresponding value is a list of tuples containing the connected target nodes and the associated edge weights. For each edge, a line is written to the file with the format: the source node, the target node, and the edge weight rounded to two decimal places. If any writing operation fails, the function will panic, ensuring that issues are caught early. After successfully processing all nodes and edges, a confirmation message is printed to indicate the file has been successfully exported. This function provides a straightforward way to convert a graph's

adjacency list representation into a standard CSV format, making the graph data easy to share, analyze, or visualize in external tools.

Graph.rs - This module provides a suite of functions for processing and analyzing a dataset of nodes and generating a weighted graph representation. The split_by_unit_num function is responsible for dividing a vector of Node objects into two groups based on the value of unit_num. Nodes with unit_num == 1 are placed into one group, and the rest into another. This separation allows for analyzing "crude" and "adjusted" metrics separately. The subdivide_by_stub_name_num function further splits a group of nodes into 12 distinct subcategories based on their stub_name_num values. A vector of 12 empty vectors is initialized to hold nodes corresponding to specific categories. Each node is assigned to one of these subgroups based on its stub_name_num. If a node has an invalid stub_name_num, a warning is printed to notify the user. The build_graph function constructs a graph from a vector of nodes. The graph is represented as a HashMap where each key is a unique node identifier (constructed using stub_label_num, year_num, and age_num), and each value is a vector of tuples containing connected node IDs and their corresponding edge weights. The function iterates through all pairs of nodes and calculates the edge weight between them if they share at least one common attribute (stub_label_num, year_num, or age_num). The weight is determined by normalizing the nodes' estimate values relative to the higher estimate of the pair, and then subtracting the absolute difference. Bidirectional edges are added for each pair of connected nodes, ensuring the graph is symmetric. A debug statement prints the total number of edges in the graph to provide insight into the graph's size. The sort_graph function is a utility designed to ensure consistent order in the graph's structure, making it easier to compare graphs in tests. It sorts the top-level keys of the graph and the edges within each node by their target node ID. This function is particularly useful for ensuring deterministic test results. A test case, test_build_graph, verifies the correctness of the build_graph function. It creates three sample nodes with predefined values and manually constructs the expected graph by calculating the edge weights between the nodes. The function is then invoked with the sample nodes, and the resulting graph is sorted and compared to the expected graph. By using the sort_graph function, both the actual and expected graphs are normalized to ensure a fair comparison. The test ensures that the build_graph function correctly identifies connections between nodes, calculates accurate edge weights, and constructs the graph structure as expected. If the graphs match, the test passes, confirming the accuracy of the implementation.

Main.rs - The main function serves as the central coordinator for processing a dataset of nodes, analyzing it, and exporting results as a CSV file. The process begins by specifying the file path to the input CSV data, "data.csv". The function first invokes load_data_from_csv to read and parse the file into a vector of Node objects. It checks if any data was loaded; if not, it prints an error message and exits, ensuring the program handles missing or incorrectly formatted files gracefully. If the file is successfully loaded, the total number of nodes is printed to provide

feedback to the user. The nodes are then split into two primary groups using the split_by_unit_num function. Group 1 contains nodes where unit_num == 1, and Group 2 contains nodes where unit_num != 1. This division allows for separating data into "crude" and "adjusted" metrics, as defined by the unit_num field. The sizes of both groups are printed to indicate how the data was partitioned. The program proceeds to further process Group 2, which is subdivided into 12 smaller groups (indexed from 0 to 11) based on the stub_name_num field. This is accomplished using the subdivide_by_stub_name_num function. The subdivision organizes the data into more granular categories, making it easier to analyze specific subsets of the data. The program then focuses on the 6th subgroup (index 5) within Group 2. It checks if this subgroup exists and is non-empty. If valid, the build_graph function is called to construct a weighted graph representation from the nodes in this subgroup. The graph is stored as a HashMap, where each key represents a node, and the corresponding values are vectors of connected nodes with edge weights. The number of nodes in the graph is printed to confirm successful graph creation. Once the graph is built, the program prepares to analyze it. Although the run_label_propagation function is commented out in the current version, it indicates that label propagation could be applied to further process or cluster the graph data. This step would help detect patterns or communities within the graph. Finally, the graph is exported to a CSV file named "graph_6.csv" using the export_graph_to_csv function. This function writes the graph edges, along with their weights, to the output file in a standardized format for easy viewing or further analysis. If the 6th subgroup is empty or does not exist, appropriate error messages are printed to inform the user. This ensures the program can handle missing or incomplete data without crashing. By organizing the process into clear steps—loading data, splitting and subdividing nodes, building the graph, and exporting results—the code maintains readability, modularity, and robustness. Each function performs a specific task, making it easy to debug or extend the program for additional analyses in the future.

Node.rs - This code defines a Node struct and its associated implementation, which serves as a container for demographic and statistical data. The Node struct has several fields that store different properties of the data, including unit, unit_num, stub_name, stub_label, year, age, and estimate. These fields are strongly typed to ensure type safety, with strings for categorical fields (e.g., unit, year) and numerical types (i32 and f64) for numerical data, such as unit_num and estimate. The struct uses the #[derive(Debug, Clone, PartialEq)] attribute to automatically implement the Debug, Clone, and PartialEq traits. This enables the struct to be printed in a human-readable format (via Debug), copied easily (Clone), and compared for equality (PartialEq), which is essential for testing. The Node::new method acts as a constructor to simplify the creation of Node instances. It takes the values for all the fields as arguments, converts string fields to String using .to_string(), and initializes the struct. This approach provides a clean and consistent way to instantiate nodes, ensuring that the input data is appropriately formatted. The test module (#[cfg(test)]) contains two unit tests to verify the correctness of the Node struct and its implementation. The first test, test_node_initialization,

creates a Node instance using the Node::new constructor and compares it against an explicitly defined expected instance of the Node struct. The assert_eq! macro is used to ensure the created instance matches the expected instance, validating that the constructor correctly initializes all fields. The second test, test_node_debug_format, checks the Debug output format of a Node instance. It uses the format!("{:?}", node) function to generate a string representation of the Node and compares it against the expected output string. This ensures that the Debug trait produces consistent and predictable output, which is useful for debugging and logging. By structuring the code with a clear definition of the Node struct, a robust constructor, and comprehensive unit tests, the implementation is both reliable and easy to maintain. The tests validate the essential functionalities of the struct, including initialization and debugging, ensuring that it behaves as expected under typical usage scenarios.

Visual.py - This code uses the networkx and matplotlib libraries to load, process, and visualize graph data stored in CSV files. The program is designed to load a graph from a CSV file, filter the edges, and export a high-resolution visualization of the graph. The load_graph_from_csv function reads the graph data from a CSV file, where each row represents an edge with a source node, a target node, and a weight. It adds these edges to a networkx graph object. During this process, the function tracks unique nodes and ensures no duplicate nodes are added. Any errors encountered while processing the rows are caught and displayed, ensuring the program can handle problematic data. The function outputs the number of nodes and edges in the graph, giving an overview of the loaded data. The export_filtered_graph function is responsible for visualizing the graph and saving the output as a PNG file. It uses networkx to draw nodes, edges, and labels of the graph. Nodes are displayed in light blue, while edges are gray with adjustable transparency (alpha) and line width. The graph layout is passed to this function as pos, ensuring consistent positioning of nodes. The function saves the graph as a high-resolution image with specified DPI (dots per inch) and tight padding to ensure clean output. Once the graph is saved, a confirmation message is printed. The export_graphs_from_files function ties everything together. It takes a CSV file path and an optional output prefix as inputs, loads the graph using load_graph_from_csv, and generates a consistent layout for the nodes using the spring layout algorithm (with a fixed random seed for reproducibility). The edges of the graph are extracted and passed to export_filtered_graph, which produces a visualization. The output file name is created dynamically using the provided prefix and the input CSV file name, ensuring clarity and uniqueness for each exported graph. Finally, the program's main block (if __name__ == "__main__") calls export_graphs_from_files for a specific graph file (graph_6.csv). Additional code is commented out, allowing for easy batch processing of multiple CSV files (e.g., graph_1.csv to graph_12.csv) by looping through a range. This structure ensures flexibility, enabling visualization of one or multiple graphs as needed. In summary, the program automates the loading, visualization, and exporting of graph data from CSV files. It is structured to handle errors gracefully, produce consistent layouts, and save high-quality visualizations, making it a robust solution for analyzing graph-based data.

Final Analysis

While I attempted to implement the LPA (label propagation algorithm) that would break down a graph into super nodes of the most commonly connected nodes, I could not quite get it to work in a meaningful way. I also tried several times to implement the Louvain algorithm to analyze clusters in the graph but ran into many problems and felt it was perhaps out of my scope with my current skills in Rust. In the end, I do have a meaningful visualization of the data which I feel can offer some insights. The final image of this graph depicts roughly 5 main subgroups, but notably some are closer to fragmenting further than others. These groups are by far defined by their age group, suggesting that one's age, or perhaps the generation they were born into, defines the odds of one's susceptibility to suicide more than anything else, which I find quite interesting considering other factors such as year of incident or sex and race were not the defining characteristic. I must acknowledge I feel there can be so much more work to be done here to fully understand the connections and relationships between all the data, but unfortunately I do not have an abundance of time to commit to this endeavor. I will lastly explain how to interpret the data. Each node represents a specific demographic, within a specific year, within a specific age group, and it is connected to any other node/data point that shares at least one of the same of these characteristics, and its edge weight is determined by how similar the suicide rates are on a normalized scale of 0-1. For example, the node "5.125-42.5" is read as a node using the 5th demographic categorization, in this instance "sex, age, and race" in which the 125 represents 1 for male, 2 for Black or African American, and 5 for the age group 65 years or older. The next number 42 is for the year 2018, and the final number is the age group which is 5 for 65 years or older. It is redundant in this case but in other categories, age is not in them and thus requires its own field. Knowing how to read each node can let you quickly identify the clusters we see in the final visualization.