Darren Budiyanto/ DS210 B1

DS210 Project Write-up

Project overview
The purpose of the project is to analyze the used car dataset as a graph to understand the relationships among car brands and their features. By using graph algorithms and centrality measures, the project aims to identify key nodes in the network—brands that hold significant influence in the dataset. This can provide insights into car brands and trends in the used car market that may not be immediately apparent from raw data. These insights can be used for further analysis or decision-making by car companies, automotive research firms, or online car-selling platforms. I used centrality measures because it quantifies the importance of nodes within a graph; closeness centrality indicates how quickly information from a given node can spread to other nodes (nodes with high closeness centrality represent brands that are well connected to other brands) and betweenness centrality measures the extent to which a node acts as a bridge between other nodes (high betweenness centrality represent car brands that are critical in connecting different groups of cars).

How to run the program
The ds210project folder contains all the files needed to run the program. Inside the src folder (within the ds210project folder), there are 4 Rust and 1 CSV file: used_car_dataset.csv, data_processing.rs, graph.rs, centrality.rs, and main.rs. Clone the repository locally and run it in terminal.

Used_car_dataset.csv
The dataset I chose was sourced from Kaggle. It contains information about used cars in the Indian market, comprising of 9,582 entries (rows) with 11 attributes (columns). The 11 attributes are:

- Brand: car manufacturer (e.g., Volkswagen, Maserati, Honda, Tata)
- Model: specific car model
- Year: manufacturing year of the vehicle
- Age: age of the vehicle in years
- kmDriven: total kilometers driven by the vehicle
- Transmission: type of transmission (manual or automatic)
- Owner: ownership status (first or second owner)
- FuelType: type of fuel (petrol, diesel, hybrid/CNG)
- PostedDate: the date the car listing was posted
- AdditionalInfo: extra details about the vehicle
- AskPrice: listed price in Indian Rupees

Data_processing.rs
The data_processing.rs module reads a CSV file, processes its contents, and converts it into a vector of HashMaps. Each HashMap represents a row of the CSV file, where the keys are the column header and the values are the corresponding cell entries for that row.

The function **process_dataset** first configures the CSV reader with headers enabled (using ReaderBuilder), then opens the file at the specified file_path for reading. The function checks if the CSV file can be successfully read using if let Ok(mut reader) = reader, extracts the first row (headers), and clones it for use as keys in the HashMap. The loops then reads each row from the CSV file, maps each field in a row to its corresponding header, storing them as key-value pairs in a HashMap and adds each

processed row to the dataset vector. If the file cannot be read, an error message is printed to the standard error output.

The function **test_dataset_processing** ensures that the process_dataset function works correctly and returns a non-empty dataset (using assert!). If the assertion fails, the test outputs an error message.

Graph.rs

The graph.rs module defines a graph data structure and includes methods for building graphs. It also contains the function to create subgraphs based on a target node.

The struct **Graph** represents the graph structure; nodes is a vector of strings where each string represents a unique node in the graph and edges is a vector of tuples (usize, usize) that define edges between nodes. The indices correspond to the positions of the nodes in the nodes vector. The method **new** creates and returns an empty Graph instance with no nodes or edges. The method **add_node** checks if the node already exists in the nodes vector; if not, it appends the node to nodes (to prevent duplicate nodes). The method **add_edge** creates an edge between two nodes. It first finds the indices of the two nodes in the nodes vector, then adds a tuple (index1, index2) to the edges vector. The method **nodes** and **edges** return a reference to the nodes and edges vector respectively to allow access to the list of nodes and edges without modifying the graph (getter methods). The **build_graph** function takes a dataset as input and creates a graph. First, it initializes an empty graph, iterates over the dataset (data), looks for the key "Brand" in each row. If the "Brand" key exists, its value is added to the graph as a node using graph.add_node(brand.clone()). Next, it clones the list of nodes for indexing, iterates over the indices of the nodes, and for each node, it creates an edge connecting it to the next node in the list (graph.add_edge(&nodes[i],&nodes[i + 1]);). The **build_subgraph** function extracts a subgraph centered around a target node. First, it searches the graph.nodes vector for the target_node, if found, it retrieves the index of the target node. Next, it adds the target node to the subgraph using subgraph.add_node(graph.nodes[index].clone()). It iterates through all edges in the main graph. If the start or end of an edge matches the target node's index, the other node in the edge is identified as a neighbor and adds the neighbor node and the edge to the subgraph.

The function **test_graph_construction** (ensures that the build_graph function can correctly handle a dataset) creates a dataset with one row containing a brand value, which is passed to the build_graph function. The test asserts there is exactly one node in the graph and no edges exist because there is only one node. The **test_add_node** function confirms that the add_node method works (nodes are added correctly, without duplication). It creates an empty graph, adds one node, and asserts that there is one node in the graph and the node value matches the added value. The **test_add_edge** function ensures that the add_edge method works (edges are added accurately and reference the correct nodes). It adds two nodes ("Toyota" and "Honda") to the graph and creates an edge between them; the test asserts that there is one edge in the graph and connects the correct nodes (indices 0 and 1). The **test_build_subgraph** function ensures that the build_subgraph function can correctly extract a subgraph. It constructs a graph with three nodes and two edges (connecting "Chevrolet" to "Jeep" and "Jaguar", extracts a subgraph centered on "Chevrolet" and asserts the subgraph contains all three nodes and both edges connected to "Chevrolet" are present.

Centrality.rs

The centrality.rs module computes graph centrality measures: closeness centrality and betweenness centrality and determines the nodes with the highest centrality scores.

The function **bfs_shortest_path** calculates the shortest path distances from a starting node to all other nodes using Breadth-First Search (BFS). Distances is a vector of size equal to the number of nodes, initialized to None. Each index represents a node, and its value will hold the shortest distance from the start node once calculated. Queue facilitates BFS traversal, initialized with the start node. Distances[start] is set to Some(0) since the distance from the starting node to itself is zero. The BFS loop continues until the queue is empty and current_distance retrieves the shortest distance to the current node (node), then the function iterates over each edge to check if the current node is part of the edge. If yes, it identifies the neighbor node. If the neighbor node hasn't been visited (distances[neighbor].is_none()), it updates its shortest distance to current_distance + 1 and adds it to the queue. The function finally returns the distances vector, where each index corresponds to a node, and its value represents the shortest path distance (Some) or None if the node is unreachable. The function **calculate_closeness_centrality** computes the closeness centrality for each node in the graph. It iterates over nodes, for each node i, bfs_shortest_path calculates shortest path distances from node i to all other nodes, reachable_distances filters out unreachable nodes and collects the distances to reachable ones, total_distance sums up the distances to all reachable nodes and reachable_countcounts how many nodes are reachable from i. If the total distance is non-zero and more than one node is reachable, it computes the closeness centrality using the formula: (reachable nodes – 1) / sum of shortest path distances. Otherwise, centrality is set to 0.0. It then scales all centrality values to a range of [0, 1] by dividing them by the maximum value. The function **calculate_betweenness_centrality** computes the betweenness centrality for each node (how often a node acts as a bridge along the shortest path between two other nodes). It does a BFS from each source node s, shortest_paths tracks the number of shortest paths to each node, dependencies tracks dependencies for centrality calculation, and predecessors keeps a list of predecessor nodes for each node. The source node s has exactly one shortest path to itself (shortest_paths[s] = 1.0). The function then loops the graph to compute shortest paths and track predecessors for dependency calculations. It processes nodes in reverse order using the stack, updates the dependencies of predecessors based on the shortest path ratios, adds the dependency contribution to the centrality score of each node, and divides each centrality score by the maximum possible value to scale results. The function **get_highest_centrality_indices** finds the index of the maximum closeness centrality value, repeats the process for betweenness centrality, and returns a tuple of indices for the nodes with the highest centrality scores.

The function **test_closeness_centrality** ensures that closeness centrality values are computed correctly for a graph with two connected nodes (asserts both nodes have non-zero centrality). The function **test_betweenness_centrality** ensures that betweenness centrality values are computed correctly (asserts that values are zero for nodes in a direct connection/ no intermediate nodes). The function **test_get_highest_centrality_indices** ensures that get_highest_centrality_indices can correctly identify the node with the highest centrality values in a graph with three nodes connected in a chain.

Main.rs

First, the program processes the "used_car_dataset.csv" dataset using the process_dataset function from the data_processing module. Using the processed data, the build_graph function constructs the main graph; nodes represent car brands and edges represent relationship between nodes. Next, it computes and prints the closeness centrality and betweenness centrality scores for each node using calculated_closeness_centrality and calculate_betweenness_centrality. The get_highest_centrality indices function identifies the nodes with the highest centrality values and prints the result. Then, the build_subgraph function extracts and prints a subgraph for nodes and edges related to "Chevrolet". Lastly, it prints the closeness and betweenness centrality scores for nodes in the subgraph.

Output

```
Centrality Measures Analysis Project
Closeness Centrality: [
    0.5128205128205128,
    0.5397727272727273,
    0.5680119581464873,
    0.5974842767295597,
    0.628099173553719,
    0.6597222222222222,
    0.692167577413479,
    0.7251908396946564,
    0.7584830339321358,
    0.7916666666666666,
    0.824295010845987,
    0.8558558558558558,
    0.8857808857808858,
    0.9134615384615383,
    0.9382716049382717,
    0.9595959595959594,
    0.9768637532133675,
    0.9895833333333333,
    0.9973753280839894,
    1.0,
    0.9973753280839894,
    0.9895833333333333,
    0.9768637532133675,
    0.9595959595959594,
    0.9382716049382717,
    0.9134615384615383,
    0.8857808857808858,
    0.8558558558558558,
    0.824295010845987,
    0.7916666666666666,
    0.7584830339321358,
    0.7251908396946564,
    0.692167577413479,
    0.6597222222222222,
    0.628099173553719,
    0.5974842767295597,
    0.5680119581464873,
    0.5397727272727273,
    0.5128205128205128,
]
```

```
Betweenness Centrality: [
    0.0,
    0.07965860597439545,
    0.12944523470839261,
    0.1763869132290185,
    0.22048364153627312,
    0.26173541963015645,
    0.30014224751066854,
    0.3357041251778094,
    0.3684210526315789,
    0.39829302987197723,
    0.42532005689900426,
    0.44950213371266,
    0.47083926031294454,
    0.48933143669985774,
    0.5049786628733998,
    0.5177809388335705,
    0.5277382645803699,
    0.534850640113798,
    0.5391180654338549,
    0.5405405405405406,
    0.5391180654338549,
    0.534850640113798,
    0.5277382645803699,
    0.5177809388335705,
    0.5049786628733998,
    0.48933143669985774,
    0.47083926031294454,
    0.44950213371266,
    0.42532005689900426,
    0.39829302987197723,
    0.3684210526315789,
    0.3357041251778094,
    0.30014224751066854,
    0.26173541963015645,
    0.22048364153627312,
    0.1763869132290185,
    0.12944523470839261,
    0.07965860597439545,
    0.0,
]
```

```
Node with highest closeness centrality: Chevrolet (Value: 1.0000)
Node with highest betweenness centrality: Chevrolet (Value: 0.5405)
Subgraph for Chevrolet: Graph { nodes: ["Chevrolet", "Jeep", "Jaguar"], edges: [(1, 0), (0, 2)] }
Closeness Centrality in subgraph: [1.0, 0.6666666666666666, 0.6666666666666666]
Betweenness Centrality in subgraph: [2.0, 0.0, 0.0]
```

The output above displays the closeness and betweenness centrality values for all car brands. The node with the highest closeness centrality and betweenness centrality is Chevrolet. The closeness centrality value of 1 suggests that it is the most accessible node in the graph, e.g. Chevrolet has the shortest average path length to all other nodes in the graph. This result matches my intuition that Chevrolet is well-connected or central in the network because of its prevalence in the dataset and relationships with other car brands. A betweenness centrality of 0.5405 also suggests that it serves as a bridge between other nodes. Having the highest betweenness centrality also suggests that many shortest paths between other nodes pass through Chevrolet, again reinforcing the importance in connecting different parts of the graph and my intuition that Chevrolet is a popular brand in the used car market and that it shares a lot of attributes (e.g. fuel type, transmission, ask price, age) to other car brands in the used car market.

When looking at the subgraph related to Chevrolet, the output shows that Chevrolet connects to Jeep (edge between nodes 0 and 1) and Chevrolet connects to Jaguar (edge between nodes 0 and 2). Chevrolet has a closeness centrality of 1.0 and both Jeep and Jaguar have a closeness centrality of 0.67. Chevrolet has a betweenness centrality of 2.0 and both Jeep and Jaguar have a betweenness centrality of 0. Similar to the main graph, Chevrolet has the highest closeness centrality, indicating that it has the shortest average distance to the other nodes in the subgraph. This occurs because Chevrolet is the only node that connects Jeep and Jaguar. Any shortest path between Jeep and Jaguar must pass through Chevrolet, hence Chevrolet is a critical bridge connecting them. Therefore, the centrality measures analysis in both the main graph and subgraph conclude that Chevrolet plays a central role as a hub, maintaining connectivity and influence within the network.