A. Project Overview
What question are you answering?
The project I worked on shows how products are related based on their monthly demand patterns, and what we can learn from those relationships using graph algorithms. By transforming product demand data into a similarity graph, I was aiming to identify central items and be able to discover close connected groups, and understand how fragmented or cohesive the product space is.

What dataset are you using and where is it from?
I used the abc_xyz_dataset.csv it was from kaggledata
link:https://www.kaggle.com/datasets/shahriarkabir/abc-xyz-inventory-classification-dataset . It contains 1000 products, each with monthly demand figures across 12 months. Every row includes a product ID, name, category, and demand data. This dataset was well suited for similarity-based analysis, especially using cosine similarity. It was small enough to process efficiently but large enough to explore meaningful structure in a graph.

B. Data Processing
How did you load it into Rust?
I used Rust's standard library, specifically BufReader and split, to read the CSV file line by line. Each line was analysed into an Item struct containing the product ID and a vector of 12 demand values.

Did you apply any cleaning or transformations?
I skipped malformed or incomplete rows during CSV analysis. Aside from that, no extra cleaning was needed because the dataset was already formatted consistently. The main transformation involved turning each demand row into a normalized demand vector for cosine similarity comparison.

C. Code Structure
Modules
graph.rs: Builds a similarity graph using cosine similarity and creates k-nearest neighbor edges (top 3 similar items).

shortest_path.rs: Implements Breadth-First Search (BFS) to compute shortest paths between nodes.

centrality.rs: Calculates closeness centrality using BFS.

clustering.rs: Counts the number of connected components using BFS.

densest.rs: Implements a greedy 2-approximation algorithm to find the densest subgraph.

Key Functions
calc_cosine_similarity(a: &Vec<f64>, b: &Vec<f64>) -> f64

Purpose: Calculates how similar two product demand vectors are using cosine similarity, which is the backbone for building edges in the similarity graph.

Inputs: Two vectors a and b of equal length, each representing numerical features (e.g., monthly demand).

Outputs: A floating-point number between 0.0 and 1.0 representing the cosine similarity.

Logic: Computes the dot product of a and b, divides by the product of their norms. If either vector is a zero vector, returns 0.0 to avoid division by zero.

2. build_graph(items: &HashMap<String, Vec<f64>>, k: usize) -> HashMap<String, Vec<String>>
Purpose: Constructs a k-NN similarity graph by connecting each item to its top-k most similar items.

Inputs: A hash map of item IDs to feature vectors, and an integer k for the number of neighbors.

Outputs: A graph represented as an adjacency list (hash map of item IDs to lists of neighbor IDs).

Logic: For each item, computes cosine similarity to all others, sorts them by score, and retains the top-k as neighbors. Builds an undirected graph.

3. analyze_paths(graph: &Graph)
Purpose: Samples pairs of nodes and computes average shortest path lengths using BFS to explore connectivity.

Inputs: The similarity graph as an adjacency list.

Outputs: Printed average shortest path distance based on sampled pairs.

Logic: Randomly selects start nodes and performs BFS to gather distances to all reachable nodes. Aggregates results to compute the average path length.

4. compute_closeness(graph: &Graph)
Purpose: Computes closeness centrality for each node and prints the top 5 with the highest values.

Inputs: The similarity graph.

Outputs: Console output of nodes ranked by centrality score.

Logic: For each node, performs BFS to all reachable nodes and calculates centrality as the inverse of the sum of shortest path distances. Normalizes by node count.

5. count_clusters(graph: &Graph)
Purpose: Counts how many disconnected clusters exist in the graph.

Inputs: The graph as an adjacency list.

Outputs: Prints the number of connected components.

Logic: Uses DFS or BFS to explore unvisited nodes and marks each component. Increments a counter per connected group.

6. approximate_densest(graph: &Graph)
Purpose: Approximates the densest subgraph by using a 2-approximation greedy method.

Inputs: The full similarity graph.

Outputs: Prints the best density found during iterative node removals.

Logic: Repeatedly removes the node with the lowest degree, recalculates graph density (edges ÷ nodes), and tracks the highest density achieved before the graph is fully removed.

Main Workflow
The program first reads the dataset and constructs the graph. Then, it samples nodes to compute average shortest path length. Calculates closeness centrality scores. Counts the number of disconnected clusters. Runs densest subgraph approximation to find the linked items.

D. Tests
How did you test it?
I began testing the project by manually verifying correctness through sample runs and console outputs. This involved checking the size of the graph, confirming the expected number of edges, and ensuring that the BFS traversal returned the correct distances between nodes. To validate the correctness of my similarity logic, I implemented a standalone test for the cosine similarity function. Since this function plays an important role in building the similarity graph, it was important to verify its behavior with known inputs. I wrote a self contained unit test within the main.rs file under the #[cfg(test)] module. The test redefines the cosine similarity logic and checks that two identical vectors return a similarity score of exactly 1.0. I used assert! with a tolerance margin (1e-6) to account for floating-point precision.

What should each test ideally check?

- That the cosine similarity function accurately identifies similar vectors and correctly handles edge cases like zero vectors.
- That the BFS implementation traverses all reachable nodes from a given starting point and computes accurate shortest path distances.
- That closeness centrality scores and connected component counts align with expectations when tested on small, predictable graph structures.
- That the densest subgraph approximation correctly removes nodes with the lowest degree in each step and maintains the highest density seen throughout the process.

E. Results

```
Average shortest path length (sampled): 6.96
Top closeness centrality:
ITM_190: 0.0007
ITM_687: 0.0007
ITM_454: 0.0007
ITM_637: 0.0007
ITM_295: 0.0007
Number of connected clusters: 155
Approximate densest subgraph density: 3.00
```

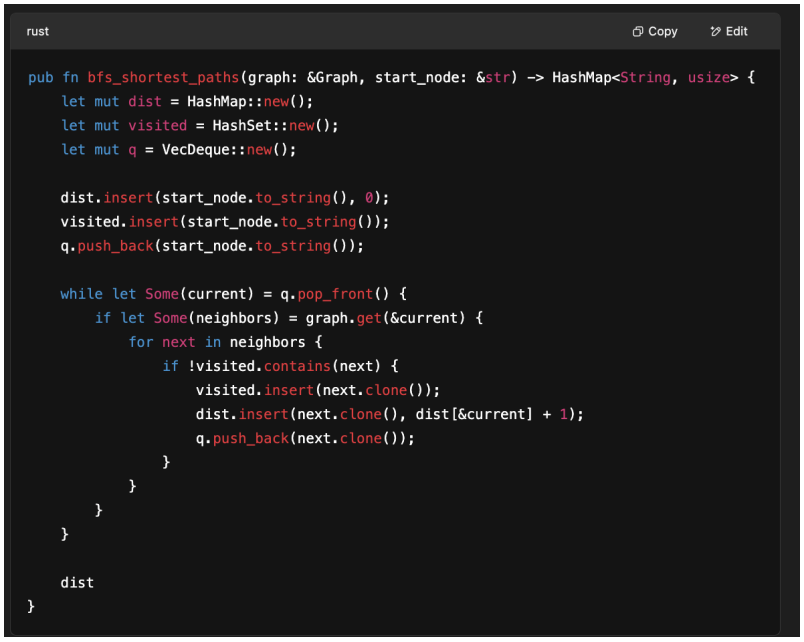
Interpretation:
All five top central products (ITM_190, ITM_687, ITM_454, ITM_637, and ITM_295) have the same closeness centrality score of 0.0007, which means they are equally close, on average, to all other products in the network. In other words, these items can reach the rest of the graph with fewer steps than most others, making them the most “central” or “influential” in terms of demand similarity. These could be general purpose or trend following products that exhibit behavior similar to a wide variety. The graph is split into 155 connected clusters, which shows that the overall product network is highly fragmented. Each cluster is a group of items that are connected through similarity, but not linked to products outside the group. This suggests many items in the dataset have unique or niche demand patterns that don’t align closely with others. The approximate densest subgraph has a density of 3.00, meaning that within the most tightly connected group, each product is connected to about 3 others, on average. This region likely represents a small set of products with very strong mutual similarity — for example, items that are commonly bought together or that share nearly identical demand patterns over time.

F. Usage Instructions
How to build and run the code:
Cargo run

G. AI Assistance Disclosure and Citations

Did you use AI? If so, how?

Throughout this project, I used ChatGPT to help me break down complex Rust concepts and build out the structure of my code. I got help generating reusable functions, designing modules, and understanding graph logic like BFS and centrality. One of the key areas where AI was helpful was in writing the cosine similarity function, which measures how similar two products are based on their monthly demand vectors. I used the code suggestion as a starting point and integrated it into my graph.rs module, adjusting it to fit the data types I used. I also got guidance on implementing BFS to compute distances in the graph. This was especially useful for calculating closeness centrality and finding connected components. I had to debug some ownership issues specific to Rust, and ChatGPT helped me understand how to use HashMap, HashSet, and VecDeque correctly in this context. Finally, I followed a template for a greedy approximation algorithm for densest subgraphs. Overall, I adapted it to my needs, and learned a lot in the process—especially about how to break down graph logic into modular, testable Rust functions.



rust Copy Edit

```rust
fn calc_cosine_similarity(vec1: &Vec<f64>, vec2: &Vec<f64>) -> f64 {
    let mut dot_product = 0.0;
    let mut norm1 = 0.0;
    let mut norm2 = 0.0;

    for i in 0..vec1.len() {
        dot_product += vec1[i] * vec2[i];
        norm1 += vec1[i].powi(2);
        norm2 += vec2[i].powi(2);
    }

    if norm1 == 0.0 || norm2 == 0.0 {
        return 0.0;
    }

    dot_product / (norm1.sqrt() * norm2.sqrt())
}
```