Written Report

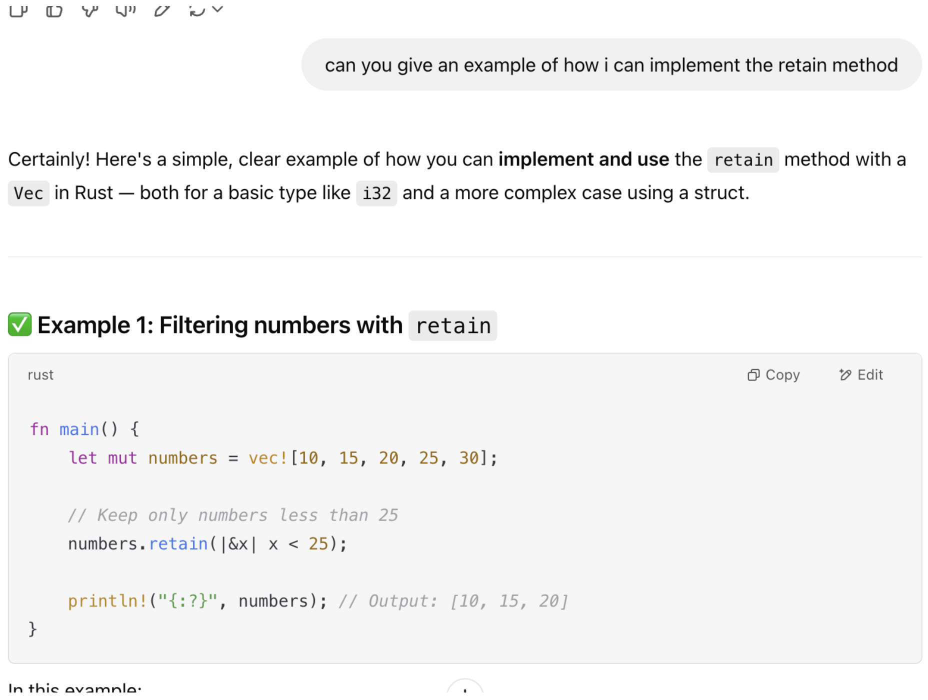
A. Project Overview

- **Goal**: The questions that I was interested in answering for this project was: what are the most common direct routes in the uber dataset? Which locations are most common for the categories business and personal. What is the shortest path between the most frequent routes? And finally, I was interested in some of the statistics of the data set such as mean, standard deviation, and the most number of hops in the location graph.

- **Dataset**: Kaggle, 1156 https://www.kaggle.com/datasets/ruchikakumbhar/uber-dataset

B. Data Processing

- **Loading**
    - I made a function read_file(path: &str) to open the uber csv by using tsd::fs::File as well as io::BufReader
    - The function iterates through each line and splits on commas. The function extracts the Categories for travel that I am interested in Business and Person. I parsed the Categories into an Enum.
- Any **cleaning** or **transformations** applied.
    - Instead of cleaning the csv in excel, I used Rust and filtered out records where the start and stop columns were empty with "Unknown Location." I used chatgpt to help me implement code that can filter out the unknown locations.



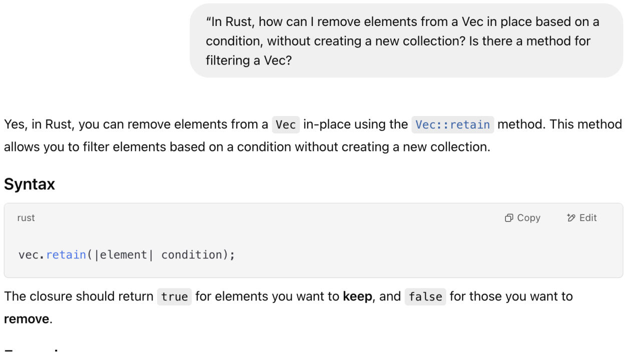
can you give an example of how i can implement the retain method

Certainly! Here's a simple, clear example of how you can **implement and use** the `retain` method with a `Vec` in Rust — both for a basic type like `i32` and a more complex case using a struct.

✅ Example 1: Filtering numbers with `retain`

```rust
fn main() {
    let mut numbers = vec![10, 15, 20, 25, 30];

    // Keep only numbers less than 25
    numbers.retain(|&x| x < 25);

    println!("{:?}", numbers); // Output: [10, 15, 20]
}
```

In this example:



“In Rust, how can I remove elements from a Vec in place based on a condition, without creating a new collection? Is there a method for filtering a Vec?

Yes, in Rust, you can remove elements from a `Vec` in-place using the `Vec::retain` method. This method allows you to filter elements based on a condition without creating a new collection.

Syntax

```rust
vec.retain(|element| condition);
```

The closure should return `true` for elements you want to **keep**, and `false` for those you want to **remove**.

C. Code Structure

1. **Modules**

    - The first module I implemented is stats (stats.rs). This stats module contains some basic graph statistic operations:
        - Bfs: computes the shortest path distance (hops) from a given node to all other nodes in a directed graph.
        - Mean_distance: calculates the mean of all the distances found from BFS in a collection of vectors.
        - Std_dev: computes the standard deviation of the BFS searches
        - max_distance: finds the maximum BFS distance from the pairs

2. **Key Functions & Types (Structs, Enums, Traits, etc)**

    - **Enum Category**: creates a ride category with two variants (Personal and Business). The outputs are Category(Business or Personal)
    - **read_file(path: &str)**: the purpose is to read a CSV file as a vector of tuples with 3 elements: start location, stop location, category). It opens the file, splits the lines, and parses the string to enum.
        - **Output**: Vec<(String,String,Category)>
    - **unique_nodes(rides)**: collects the unique location names into a HashSet.
        - **Input:** &[(String,String,Category)]
        - **Output:** HashSet<String>
    - Adjacency_list: builds a directed adjacency list and returns a mapping of node-index. The function counts each pair in a HashMap, then sorts them, and finally truncates.
        - Input: rides, N: usize
        - Output: Vec<((String,String),usize)>
    - popular_hubs(rides): finds the busieset location independently for personal and business trips
        - Input: rides
        - Output: (String,String)

    - Shortest_path: finds the shortest path between two node index by using BFS and back pointers. It records prev[] vector, which has the length equal to the number of nodes in the graph during BFS, then rebuilds the path from end node to start node.
        - Input: adj, start: usize, end: usize
        - Output: Option<Vec<usize>>

3. **Main Workflow**

    - The program begins by reading and cleaning the csv file “UberData.csv”. It loads the start stop and category tuple. The main function then begins to build a graph. First it collects distinct pickup/drop off locations in a HashSet (unique_nodes). Then the set is sorted as a list so that each location has a fixed index. Then we loop over the rides for each pair of start and stop, and loop up the indices to add an edge in the adjacency list. The sorted list of names is kept in a Vector of strings. The next step is to identify the five most common trips using most_frerquent_pairs. Then, popular_hubs finds how often a location appears for the respected categories (Business and Person). Then the shortest_ppath prints out the smallest sequence of rides between two points, which is found in the locations list. Finally to understand the network connectivity, the program runs Breadth-First-Search from a given node, and calculates the statistics.

D. Tests

- `cargo test` **output** (paste logs or provide screenshots).

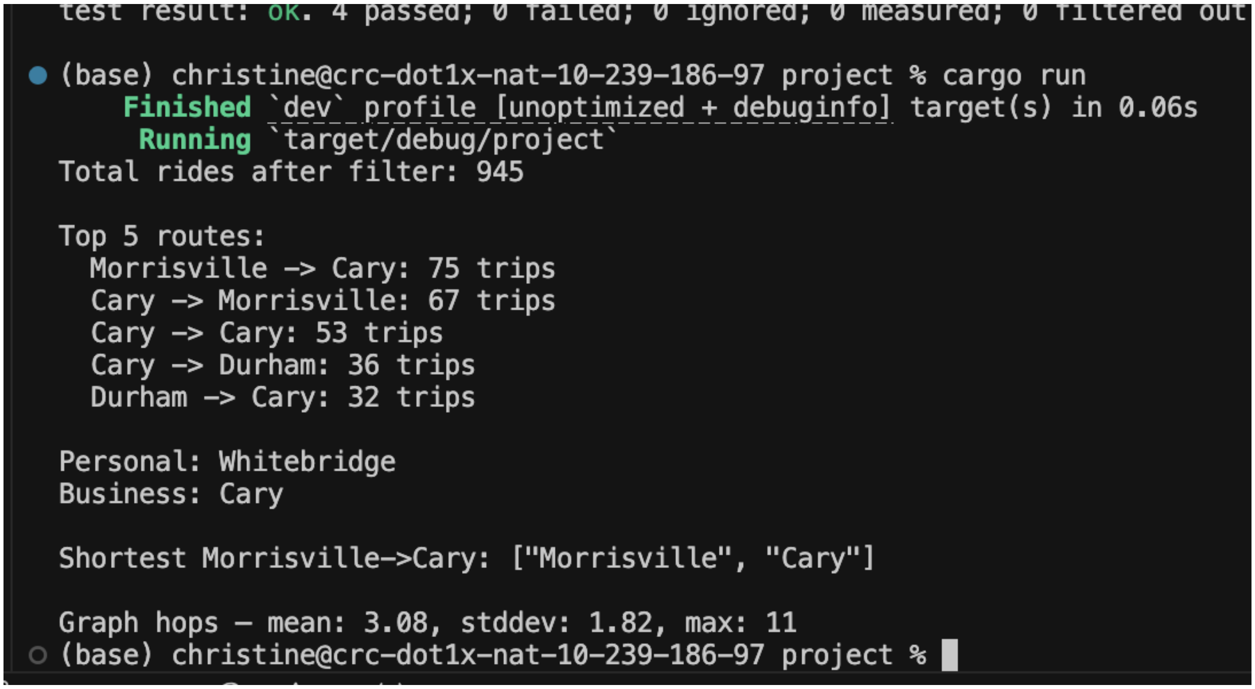
```
running 4 tests

test tests::test_unique_nodes ... ok

test tests::test_bfs_zeros ... ok

test tests::test_max_distance ... ok

test tests::test_most_frequent_pairs_counts ... ok



test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in
0.00s
```

- For each test: what it checks and why it matters.
    - test_unique_nodes: check that the unique nodes collect and sort all of the unique start and stop locations
    - test_most_frequent_pairs: this test identifies and orders the most frequent number of direct routes based off of the trip count

  - test_bfs_zeros: makes sure that every node has a distance of zero to itself when running BFS on the test cycle
  - test_max_distance: checks that the max_distance function outputs the longest shortest-path

E. Results

- All program outputs (screenshots or pasted).



```
  test result: ok. 4 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

● (base) christine@crc-dot1x-nat-10-239-186-97 project % cargo run
      Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.06s
       Running `target/debug/project`
  Total rides after filter: 945

  Top 5 routes:
    Morrisville -> Cary: 75 trips
    Cary -> Morrisville: 67 trips
    Cary -> Cary: 53 trips
    Cary -> Durham: 36 trips
    Durham -> Cary: 32 trips

  Personal: Whitebridge
  Business: Cary

  Shortest Morrisville->Cary: ["Morrisville", "Cary"]

  Graph hops — mean: 3.08, stddev: 1.82, max: 11
○ (base) christine@crc-dot1x-nat-10-239-186-97 project %
```

- The most frequent Direct routes are Morrizville to cary, cary to morrisville, cary to cary, cary to durham, and durham to cary. THe busiest hub for personal trips is Whitebridge, and Vary for business trips.

F. Usage Instructions

- In the terminal
  - cargo build – release
  - Cargo run - release
  - Cargo test
- Expected run time: 0.06s