Introduction
The data set I will be using is the "Gym Members Exercise Dataset" (https://www.kaggle.com/datasets/valakhorasani/gym-members-exercise-dataset/data). This data set is interesting to me because I enjoy going the gym and is an active gym member that exercises about 4 days a week. Going to the gym has become a habit and a passion for me, hence I am eager to apply the skills I have learned in data science to analyze patterns that might exist in the gym. The main problem that I will be analyzing is how to identify the best/optimal gym buddy in the gym for every gym member.

Code Logic Explanation
First, I created a struct called Members which acts as the logical foundation for my whole code. The Member struct in the code is designed to represent a gym member in the csv file, encapsulating their attributes like age, gender, weight, height, workout type, workout frequency, and health metrics such as BPM, calories burned, water intake, and BMI. Each attribute is assigned a variable type, such as gender is string type, workout frequency is an unsigned integer, and weight is a float. Furthermore, the last line (connections: HashSet<usize>) of code in the Member struct is used to visualize the relationship between gym members in a graph-like structure. That is, each Member is seen as a node in the graph, and the connections field holds the IDs of other members who are considered "connected" to the member based on their similarity score exceeding a predefined threshold later on in the main code.

Then I used impl to implement methods and functions for the Member Struct. The first function, new(), is used as a constructor for creating a new Member instance once called. The function has two inputs: first, "id" is used as an identifier to assign each member with an integer starting from the first line of the csv table; second, "attribute" is used as a vector that contains string values parsed from a single line in the csv file, representing the different variables of the table. The Self struct is used for data cleaning the csv file, which I approached by parsing and unwrapping the numerical data in the file. Parse() is used to convert the strings from the csv to another variable type, and unwrap() allows me to extract the non-string value outputted by the Ok function in parse, making sure that the data is cleaned to its appropriate type assigned in the Member struct. For the variables that are already strings, I simply used a to_string() method to convert the &str into an independent ownership string.

The read_csv function is used to read the data in the csv file and convert each row into a new Member struct. The overall logic and format of the code follows the read_csv function taught in the lecture. The function initializes an empty Vec<Member> to store the resulting member records. As it iterates through the lines, it skips the first line (the header row) and splits each subsequent line into fields using split(','). These fields, representing attributes of a Member, are passed to new(), which parses and organizes them into a structured format. Each created Member is pushed into the Vec<Member>. Finally, the function returns the vector, which contains all the parsed Member instances.

Next, I wrote a calculate_similarity function, which is really the backbone of my quantitative calculations. The function takes in two gym members as input and outputs a score that measures the similarity between the two members. I first assigned weights (added to 100) to each attribute according to how significant each one is to determining gym buddies. For instance, the workout type would have the most weight because people usually become buddies with someone doing the same type of workout, and variables like bpm, weight, and water intake would have less significance on determining whether one sees another as a gym buddy. Then, I initialized a score variable to keep track of the score for each member. Moving on, I assigned the absolute difference for each variable of each member to be input member 1's variable value – input member 2's variable value. Then, I used the formula of 1/(1+difference) to come up with a normalized score for each variable difference between the members and add it to the score variable. By using 1/(1+difference), I not only normalize the score between 0 to 1, but I can also ensure that a smaller difference results in a higher score (smaller denominator value), and vice versa. For the differences of variables like weight, calories, and age, since the difference was too large, I divided it by 10 or 1000 to make the score cleaner and more normalized. Moreover, I will multiply the score by the weighting factor for each variable assigned above. For the categorical data, I simply add the weighting factor if the two variable between the two members equal to each other, otherwise nothing will be added to score (illustrated by the if statement logic).

Finally, the find_gym_buddies function will take in account the similarity scores of each member and determine the best gym buddy for a specific member. The function takes in a member input from the Member struct and a similarity_threshold which acts as the determining point of deciding whether two members are connected enough to become potential gym buddies. The function first uses a nested loop because of the pairwise comparison, iterating through all members and assign a best_match and highest_score tracking variable, then if the similarity score between two members exceeds the similarity threshold, then the two members are considered compatible and are connected by adding each other's IDs to their respective connections sets (in the Member struct). The function tracks the best match for each member by keeping track of the highest similarity score encountered during the iteration. For each member, the ID of the member with the highest similarity score is stored in a best_buddies hash map. The function ensures that members are not matched with themselves, and only those with similarity scores above the threshold are considered as potential gym buddies. At the end, the function returns a map that associates each member with their best gym buddy.

In the end, the main function reads the file, and assigns members to the corresponding rows of the file. Then I set the similarity_threshold to 75, which states that if similarity is above 75%, then the two members could potentially become gym buddies. Moreover, I inputted all the members and the similarity_threshold into the find_gym_buddies function. At last, I iterate through all the members and let the best gym buddy to be the best gym buddy's ID for a specific member from the best_buddies hash map and provides a default value of 0 if no match is found. The code then outputs Member #x's best gym buddy is Member #y, and has #z connections. The connections here are members within that specific member's connection hash map, representing the degree distribution between it and other nodes.

```
     Running `/Users/michaelzhu/ds210_final_project/code/target/debug/code`
Member 0's best gym buddy is Member 316, and has 27 connections.
Member 1's best gym buddy is Member 722, and has 44 connections.
Member 2's best gym buddy is Member 813, and has 55 connections.
Member 3's best gym buddy is Member 148, and has 51 connections.
Member 4's best gym buddy is Member 647, and has 51 connections.
Member 5's best gym buddy is Member 840, and has 27 connections.
Member 6's best gym buddy is Member 73, and has 45 connections.
Member 7's best gym buddy is Member 882, and has 46 connections.
Member 8's best gym buddy is Member 844, and has 65 connections.
Member 9's best gym buddy is Member 884, and has 45 connections.
Member 10's best gym buddy is Member 503, and has 41 connections.
Member 11's best gym buddy is Member 633, and has 42 connections.
Member 12's best gym buddy is Member 714, and has 45 connections.
```

References

Article

1. https://www.muscleandfitness.com/workouts/workout-tips/7-rules-being-great-workout-partner/
2. https://www.hussle.com/blog/how-to-choose-the-perfect-gym-buddy/
3. https://perfectmanifesto.com/2020/11/08/what-makes-a-good-gym-buddy/

ChatGPT

specific method to convert a string to another variable type for a csv file in rust

To convert a string to another variable type in Rust, especially while processing CSV files, you can use the `.parse()` method. The `parse()` method is a general-purpose string-to-type conversion method that works for any type implementing the `FromStr` trait (e.g., `i32`, `f64`, etc.).

Here's a specific approach to handle this when working with CSV files:

how to get a value inside an Option in rust

In Rust, an `Option<T>` represents an optional value that can either be `Some(T)` (containing a value of type `T`) or `None` (representing the absence of a value). There are several ways to extract the value from an `Option`, depending on how you want to handle the `None` case.

---

1. Using `.unwrap()`

Extracts the value if it is `Some`, but panics if it is `None`.

rust Copy code

```rust
let some_value = Some(42);
let value = some_value.unwrap(); // Works because it is Some(42)
println!("Value: {}", value);

let none_value: Option<i32> = None;
// let value = none_value.unwrap(); // Panics because it is None
```

Using $\frac{1}{1+\text{difference}}$ to compute a normalized score is a common technique to scale a difference into a range, typically $(0, 1]$. Here's how it works:

Formula

$$\text{Normalized Score} = \frac{1}{1 + |A - B|}$$

- $|A - B|$: The absolute difference between two values A and B.
- The score is inversely proportional to the difference: the larger the difference, the smaller the normalized score.
- This formula naturally handles unbounded differences without requiring predefined bounds.

how to add data into a hashmap in rust

To add data to a `HashMap` in Rust, you can use the `.insert(key, value)` method. This method allows you to insert a key-value pair into the `HashMap`. If the key already exists, it updates the value associated with the key and returns the old value.

Steps to Add Data to a HashMap

1. Import the `HashMap` type from Rust's standard library.
2. Create a mutable `HashMap` to allow insertion of elements.
3. Use the `.insert(key, value)` method to add key-value pairs.

Example Code

Basic Usage:

rust Copy code

```rust
use std::collections::HashMap;

fn main() {
    // Create a mutable HashMap
    let mut map = HashMap::new();

    // Add key-value pairs to the HashMap
    map.insert("one", 1);
    map.insert("two", 2);

    // Insert a new key-value pair
    let old_value = map.insert("one", 10); // Overwrites the value for key "one"

    // Check what was replaced
    println!("Old value: {:?}", old_value); // Output: Some(1)
    println!("HashMap: {:?}", map);         // Output: {"one": 10, "two": 2}
}
```

walk me through how to find the largest value in rust

Finding the largest value in Rust depends on the type of collection you're working with. Here's a detailed walkthrough of common scenarios:

---

1. Largest Value in a Vector

The `Vec` type in Rust is a dynamic array, and you can use the `.iter()` method with `.max()` to find the largest value.

Example:

rust Copy code

```rust
fn main() {
    let numbers = vec![10, 20, 5, 30, 15];

    // Find the largest value using .max()
    if let Some(&max) = numbers.iter().max() {
        println!("The largest value is: {}", max); // Output: 30
    } else {
        println!("The vector is empty.");
    }
}
```

Walk me through the logic in a paragraph to use nodes and hash maps to find an optimal value in rust code

Using nodes and hash maps to find an optimal value in Rust typically involves representing a problem as a graph or tree and leveraging the fast lookup capabilities of a `HashMap` for efficient computations. Here's the logic:

Each node in the structure can represent an entity, state, or value, and edges between nodes indicate relationships or transitions. A `HashMap` is used to associate each node (or its identifier) with additional information, such as its current optimal value, visited status, or neighbors. During traversal (e.g., breadth-first search, depth-first search, or Dijkstra's algorithm), the `HashMap` enables efficient lookups and updates as you explore nodes. You start by initializing the map with known values (e.g., start node value as 0) and update entries as you traverse, recalculating the optimal value for each node based on its relationships or costs stored in edges. The process continues until all relevant nodes are visited, and the optimal value can be determined by querying the `HashMap` for the desired node or nodes. This approach is particularly useful for solving optimization problems like shortest paths, minimum spanning trees, or dynamic programming on graphs.

how to calculate degree distribution using rust

In graph theory, the **degree distribution** of a graph is a statistical measure that describes the frequency of degrees (the number of edges incident to a node) of nodes in the graph. To calculate the degree distribution in Rust, you need to:

1. **Represent the Graph**: You can represent the graph using an adjacency list, which stores a list of neighbors for each node.
2. **Calculate the Degrees**: The degree of a node is simply the length of its adjacency list (the number of neighbors it has).
3. **Calculate the Frequency of Degrees**: A degree distribution is a mapping of degree values (number of neighbors) to their frequencies (how many nodes have that degree).

Step-by-Step Example

Let's walk through how to compute the degree distribution in Rust:

1. Define the Graph Representation

We will use a `HashMap` where the key is a node and the value is a `Vec` representing its neighbors.

2. Calculate the Degree of Each Node

The degree of a node is just the length of its list of neighbors.

3. Count the Frequency of Each Degree

You can use another `HashMap` to store the frequencies of the degrees.