

## A. Project Overview

- **Goal:** The goal was to look into the demand fluctuations for the bike sharing system and how the trends change depending on the time-series. Moreover, applying BFS and DFS algorithms, the predicted bike rental demand according to historical data given weather conditions were found.
- **Dataset:** Seoul Bike Sharing Demand from UCI
  - <https://archive.ics.uci.edu/dataset/560/seoul+bike+sharing+demand>

## B. Data Processing

- **How it was loaded:** The csv file was uploaded to the rust project, and then the function was used to read the file into vector.
- **Cleaning:** The dataset was looked into for rooms for pre-cleaning procedure, but the data was already cleaned with no missing values and redundancy.

## C. Code Structure

### 1. Modules (5 separate modules were utilized)

- a. **common.rs:** the purpose was to define data structures, structs, enums, and types utilized across different modules
- b. **data.rs:** initializes the dataframe by loading the csv file and turning it into the vector of raw records
- c. **main.rs:** makes use of modules created and displays some key statistics regarding bike rentals
- d. **model.rs:** specifies logic used to classify the weather conditions into different groups, which will be used for search algorithms
- e. **search.rs:** implements bfs and dfs algorithms to predict bike rental count given weather conditions

The reason for utilizing five modules was to organize codes for their use. It prevents confusion.

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

#### **common.rs**

- **pub struct RawRecord**
  - Purpose: represents each record in the dataset with all the characteristics for every row entry
- **pub enum Weather, pub enum TemperatureCategory, pub enum PrecipitationIntensity**
  - Purpose: initializes possible weather labels such as “sunny, foggy, rainy, snowy” or “hot, moderate, cold”
- **pub struct Conditions**
  - Purpose: lists out traits that will be inputted by the user

- **pub struct DaySummary**
  - Purpose: represents aggregated data for a single calendar day. This will become handy during search procedure as the DaySummary of the closest match will be returned

#### **data.rs**

- **pub fn load\_data(path: &str) -> Result<Vec<RawRecord>, Box<dyn Error>>**
  - Purpose: loads in the csv file by taking in the path and outputting the vector of raw records
  - The output is the vector of raw records.
  - Core logic and key components: The for loop iterates through the raw records(rows) and then for each row, splits the date apart to specify the date/month/year structure. Then once it is checked that the raw record represents a functioning day, it is added to the dataset

#### **main.rs**

- **fn user\_input() -> Conditions**
  - Purpose: reads user input and matches with the enum and struct defined earlier on
  - Outputs the Conditions

#### **model.rs**

- **pub fn summarize\_daily\_totals(data: &[RawRecord]) -> Vec<DaySummary>**
  - This function takes in data (raw record) as the input and then returns the vector after summarizing daily totals
  - This step is crucial because the original data has separate rows for each hour, but by getting daily totals, we look at daily bike demands

#### **search.rs**

- **fn are\_similar(a: &DaySummary, b: &DaySummary) -> bool**
  - Purpose: checks if two days are similar enough to connect in the graph
  - Returns boolean depending on the result
- **pub fn build\_graph(days: &[DaySummary]) -> Graph**
  - The function for calculating similarity score is outlined. If the query and the day have essentially different values, the score goes up
  - The higher the score is, the more that value is dissimilar from what we're interested in
- **fn similarity\_score()**
  - Purpose: This function utilizes bfs algorithm, and we look for the min similarity score after iteration
- **pub fn bfs\_closest()**

- Purpose: searches through the graph with BFS and finds the closest match
  - Outputs the closest match
  - Core logic and key components: while labeling visited nodes, it compares the current best score to the score of the current node. We look for the minimum score after all the iteration.
- **pub fn dfs\_closest()**
    - Purpose: searches through the graph with DFS and finds the closest match
    - Outputs the closest match
    - Core logic and key components: while labeling visited nodes, it compares the current best score to the score of the current node. We look for the minimum score after all the iteration.

### 3. Main Workflow (fn main())

- 1) The main function first calls the load\_data function under data.rs to load the bike csv data into the scope
- 2) The function daily\_summaries is called to turn the hourly entries into per-day summaries
- 3) The function build\_graph under search.rs is called to build graph by connecting days with nodes, which will then be used for BFS and DFS
- 4) Variables season\_totals, weather\_totals, temp\_totals, and rain\_totals get initialized as hashmaps to be used to count rentals corresponding to each condition. For instance, each day entry gets classified into spring, summer, fall, or winter and then in the season\_totals, the total rented bikes for that same day get added to corresponding season
- 5) Then, the main function prints out totals for each season, weather condition, temperature group, and rain intensity
- 6) The variable query is defined to take in user input.
- 7) By calling the BFS and DFS closest distance algorithms, the closest match for the inputted weather conditions is printed with its historical bike rental counts.

## D. Tests

### ● Test Outputs

```

● (base) hyelynnkim@crc-dot1x-nat-10-239-149-104 finalproject % cargo test
  Compiling finalproject v0.1.0 (/Users/hyelynnkim/Desktop/finalproject)
  Finished `test` profile [unoptimized + debuginfo] target(s) in 3.04s
  Running unittests src/main.rs (target/debug/deps/finalproject-b567022fa49c6b2e)

running 2 tests
test search::test_bfs_finds_matching_day ... ok
test data::tests::test_load_data_success ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out; finished in 0.00s

```

### ● Explanations

- **Test 1:** The test creates an arbitrary csv file and loads the data. Looking at the first record, it checks through `assert_eq!` that each variable matches the expected.
- **Test 2:** The test aims to see if given two day summaries, the query correctly identifies the one that is closer

## E. Results

- **All outputs:**

```
(base) hyelynnkim@crc-dot1x-nat-10-239-149-104 finalproject % cargo run
Compiling finalproject v0.1.0 (/Users/hyelynnkim/Desktop/finalproject)
Finished `dev` profile [unoptimized + debuginfo] target(s) in 2.50s
Running `target/debug/finalproject`
[Seasonal average bike demand]
Spring: 17910 bikes
Summer: 24818 bikes
Fall: 22099 bikes
Winter: 5413 bikes

[Average daily demand by Weather condition]
Rainy: 15854 bikes
Sunny: 18640 bikes
Foggy: 15478 bikes
Snowy: 5756 bikes

[Average daily demand by temperature category]
Cold: 10016 bikes
Hot: 22647 bikes
Moderate: 24172 bikes

Use conditions for prediction:

Enter month (ex. 1-12): 2
Enter weather (sunny, rainy, snowy, foggy): snowy
Was it light or heavy snow? (light/heavy): heavy
Enter temperature (hot, moderate, cold): cold

[Finding the closest match]
- Month: 2
- Weather: Snowy
- Temperature: Cold
- Precipitation: Heavy
[BFS] Closest match: 03/07/2018 - 29293 rentals
[DFS] Closest match: 03/07/2018 - 29293 rentals
```

- **Interpretation**

- South Korea has four distinct seasons. It is visible how the bike demand is the highest during summer and the lowest in winter. Moreover, the bike demand when the temperature is “cold” was lower by about 50% compared to other temperature categories. Combined, it suggests that weather and temperature play a huge role in influencing people’s choices whether to rent bikes or not. Yet, the result conveys that the rental demand still exists in winter, cold and snowy conditions, proving the area of profit even in those conditions.

- The bike rental demand was very high during summer, sunny and moderate weather conditions. It might be important for bike rental companies to develop ways to practice flexibility in providing the service.

## F. Usage Instructions

- **How to build and run:**
  - Run with the “cargo run” command.
- **Command line arguments:**
  - Once the code is run, it will ask for user input, which is split into multiple components.
  - First, enter in the range of 1-2.
  - Enter weather condition from the choices: sunny, foggy, rainy, snowy
  - If answered rainy or snowy to the previous question, you will be prompted to answer whether the intensity was light or heavy. Choose one of those.
  - Then, as the last component, answer whether the temperature was cold, hot or moderate.
  - Then, the code will run through search algorithms and produce the closest match for both BFS and DFS.
- **Expected runtime: 1-2 minute**

## G. AI guidance and appendix

- The code snippets are included below. I also used chatGPT to clean and organize my codes and debug.

### Project Structure

```
graphql Copy Edit

src/
├─ main.rs
├─ data.rs      # handles CSV parsing and preprocessing
├─ model.rs     # defines model logic & prediction interface
├─ search.rs    # implements BFS/DFS trend exploration
└─ types.rs     # defines enums and data structures
```

- For my initial setup, I provided my pseudo code to chatGPT by inputting a part of my project proposal. It gave me a recommendation of how to split up the modules. (I used the name common.rs instead of types.rs)
-

how to revise this code accordingly to the changes to bfs and dfs search mechanisms

To adapt your current `main.rs` to the **revised BFS/DFS search mechanisms** (which now take a start index, a graph structure, and the full day dataset), you'll need to do **three key changes**:

## ✅ Step-by-Step Changes

### ✅ 1. Replace `make_graph()` with `build_graph()`

In your `main.rs`, change:

```
rust                                                                    Copy Edit
let graph = make_graph(&daily_summaries);
```

to:



```

/// Real BFS traversal based on graph edges
pub fn bfs_closest(start: usize, graph: &Graph, days: &[DaySummary], query: &Conditions)
    let mut visited = HashSet::new();
    let mut queue = VecDeque::new();
    let mut best_match = None;
    let mut best_score = i32::MAX;

    queue.push_back(start);
    visited.insert(start);

    while let Some(node) = queue.pop_front() {
        let day = &days[node];
        let score = similarity_score(day, query);

        if score < best_score {
            best_score = score;
            best_match = Some(day.clone());
        }

        if let Some(neighbors) = graph.adj_list.get(&node) {
            for &neighbor in neighbors {
                if visited.insert(neighbor) {
                    queue.push_back(neighbor);
                }
            }
        }
    }

    best_match
}

```

- I initially wrongly integrated the BFS search algorithm because I only compared the distances instead of actually visiting nodes of the graph. However, the new algorithm correctly traverses through the graph using BFS by labeling visited nodes and using VecDeque. Still, for the visited nodes, the current best\_score is compared to the current score, and best\_match updates to the minimum value.
- The same thing was done for DFS but in the reverse order.

how to keep track of number of bike rentals given weather conditions using hashmap

To keep track of **total bike rentals by weather condition** using a `HashMap` in Rust, you'll:

1. Use the `Weather` enum as the key.
2. Store a `(sum, count)` tuple as the value — to optionally compute averages later.

## ✓ Example

```
rust                                                                    Copy Edit

use std::collections::HashMap;
use your_crate::common::Weather; // assuming Weather enum is defined

// Create the map
let mut weather_rentals: HashMap<Weather, (i32, usize)> = HashMap::new();

// Iterate over daily summaries (or RawRecord)
for day in &daily_summaries {
    weather_rentals
        .entry(day.weather.clone()) // clone if Weather doesn't implement Copy
        .and_modify(|e| {
            e.0 += day.total_rentals; // sum of rentals
            e.1 += 1;                 // count of days
        })
        .or_insert((day.total_rentals, 1));
}
```

- In order to keep track of the number of bike rentals for different conditions, I asked chatGPT to generate a code that utilizes Hashmap instead of a manual counting. If the weather condition already exists in the hashmap, it just adds the total rentals and if not, it inserts the `day.total_rentals` in the one place.
- This code snippet was used to do similar jobs for other conditions like Precipitation and Temperature.



## ✓ Sample build\_graph() Function

```
rust Copy Edit

pub fn build_graph(days: &[DaySummary]) -> Vec<GraphNode> {
    let mut graph = Vec::new();

    for (i, day) in days.iter().enumerate() {
        let mut neighbors = Vec::new();

        if i > 0 {
            neighbors.push(i - 1); // previous day
        }
        if i + 1 < days.len() {
            neighbors.push(i + 1); // next day
        }

        graph.push(GraphNode {
            id: i,
            day: day.clone(),
            neighbors,
        });
    }

    graph
}
```

- Input: build\_graph function sample
- This function iterates through graphs and then returns vectors of graph nodes. I took a different approach of making the use of the function are\_similar to check if the node can be placed there for every iteration.